

[KOMPENDIUM] 3D-Spiele- Programm

Das Kompendium

Die Reihe für umfassendes Computerwissen

Seit mehr als 20 Jahren begleiten die KOMPENDIEN aus dem Markt+Technik Verlag die Entwicklung des PCs. Mit ihren bis heute über 500 erschienenen Titeln deckt die Reihe jeden Aspekt der täglichen Arbeit am Computer ab. Die Kompetenz der Autoren sowie die Praxisnähe und die Qualität der Fachinformationen machen die Reihe zu einem verlässlichen Partner für alle, ob Einsteiger, Fortgeschrittene oder erfahrene Anwender.

Das KOMPENDIUM ist praktisches Nachschlagewerk, Lehr- und Handbuch zugleich. Auf bis zu 1.000 Seiten wird jedes Thema erschöpfend behandelt. Ein detailliertes Inhaltsverzeichnis und ein umfangreicher Index erschließen das Material. Durch den gezielten Zugriff auf die gesuchte Information hilft das KOMPENDIUM auch in scheinbar aussichtslosen Fällen unkompliziert und schnell weiter.

Praxisnahe Beispiele und eine klare Sprache sorgen dafür, dass bei allem technischen Anspruch und aller Präzision die Verständlichkeit nicht auf der Strecke bleibt.

Mehr als 5 Millionen Leser profitierten bisher von der Kompetenz der KOMPENDIEN.

**Unser Online-Tipp
für noch mehr Wissen ...**



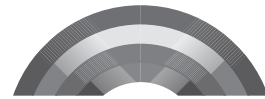
... aktuelles Fachwissen rund
um die Uhr — zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

3D-Spiele- Programmierung

Professionelle Entwicklung von
3D-Engines und -Spielen

STEFAN ZERBST OLIVER DÜVEL EIKE ANDERSON



Markt+Technik

(**KOMPENDIUM**)

Einführung | Arbeitsbuch | Nachschlagewerk

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <<http://dnb.ddb.de>> abrufbar.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:

Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

06 05 04

ISBN 3-8272-6400-6

© 2004 by Markt+Technik Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten

Coverkonzept: independent Medien-Design,
Widenmayerstraße 16, 80538 München

Coverlayout: Heinz H. Rauner, Gmund

Titelfoto: IFA-Bilderteam, Geysir, Yellowstone Nationalpark, Wyoming,
USA

Lektorat: Boris Karnikowski, bkarnikowski@pearson.de

Korrektur: Friederike Daenecke, Zülpich

Herstellung: Elisabeth Prümm, epruemm@pearson.de

Satz: reemers publishing services gmbh, Krefeld (www.reemers.de)

Druck und Verarbeitung: Bercker, Kevelaer

Printed in Germany

Im Überblick

	Vorwort	21
	Einleitung	25
Teil 1	Einführung in die Thematik	35
Kapitel 1	3D-Engines und Spieleprogrammierung	37
Kapitel 2	Design der ZFXEngine	65
Kapitel 3	Rahmenanwendung der ZFXEngine	73
Kapitel 4	Schnelle 3D-Mathematik	117
Teil 2	Rendern von Grafik	213
Kapitel 5	Materialien, Texturen und Transparenz	215
Kapitel 6	Das Render-Interface der ZFXEngine	259
Kapitel 7	3D-Pipeline und Shader	361
Kapitel 8	Skeletale Animation von Charakteren	411
Teil 3	Hilfsmodule für die Engine	469
Kapitel 9	Eingabe-Interface der ZFXEngine	471
Kapitel 10	Audio-Interface der ZFXEngine	505
Kapitel 11	Netzwerk-Interface der ZFXEngine	523
Kapitel 12	Timing und Bewegung in der ZFXEngine	589
Kapitel 13	Scene-Management	605

Teil 4 Schwarze Magie der Spieleprogrammierung..... 691

Kapitel 14 Computer Aided Design (CAD) 693

Kapitel 15 Pandoras Legacy 815

Kapitel 16 Scripting und Definition von NPC-Verhalten 877

Teil 5 Anhang 913

Anhang A Die Funktionen von ZBL/0 915

Anhang B CD-ROM und begleitendes Material zu diesem Buch..... 925

Anhang C Internetseiten rund um die Spieleentwicklung..... 927

Anhang D Epilog 929

Stichwortverzeichnis 931

Inhaltsverzeichnis

	Vorwort	21
	Einleitung	25
	Inhalt dieses Buches	25
	Was steht in diesem Buch?	25
	Was steht nicht in diesem Buch?	26
	Konzeptionelle Vorgehensweise	27
	Verwendete Werkzeuge	29
	Hardware-Voraussetzungen	30
	Was ist ZFX?	31
	Historie von ZFX	32
	Entwickler-Community ZFX	32
	Entwickler-Event zfxCON	33
Teil 1	Einführung in die Thematik	35
Kapitel 1	3D-Engines und Spieleprogrammierung	37
1.1	Modewort 3D-Engine	37
1.2	Eine Art von Magie	41
	Nostalgie und Neuzeit	41
	Der Zauberer von Oz	43
	... und die Verantwortung eines Gottes	44
1.3	Entstehungsprozess eines Spiels	45
	Spieleprogrammierung (Game-Programming)	47
	Spiele-Design (Game-Design)	48
	Spiel-Entwurf (Game-Proposal)	50

	Publisher, Veröffentlichung und Erlös	55
	Verkaufszahlen.	57
	Alternativen	58
1.4	Sage mir, mit wem du gehst	59
1.5	DirectX und OpenGL, Versionspolitik.	61
1.6	Warum schwer, wenn's auch einfach geht?	62
1.7	Auf los geht's los	63
Kapitel 2	Design der ZFXEngine	65
2.1	Anforderungen an die Engine	65
2.2	API-Unabhängigkeit durch Interface-Definitionen	67
2.3	Struktur der ZFXEngine	69
2.4	Komponenten der ZFXEngine	70
	ZFXRenderDevice Interface	70
	ZFXInputDevice Interface.	71
	ZFXNetworkDevice Interface	71
	ZFXAudioDevice Interface	71
	ZFX3D Bibliothek	72
	ZFXGeneral Bibliothek.	72
2.5	Ein Blick zurück, zwei Schritt nach vorn	72
Kapitel 3	Rahmenanwendung der ZFXEngine	73
3.1	Begriffsbestimmung Interface	73
3.2	Unser Interface	75
3.3	Der Arbeitsbereich für unsere Implementierung	77
	ZFXRenderer, eine statische Bibliothek als Manager	79
	ZFXD3D, eine dynamische Bibliothek als Render-Device	80
	ZFXRenderDevice, ein Interface als abstrakte Klasse.	82
3.4	Implementierung der statischen Bibliothek.	84
3.5	Implementierung der dynamischen Bibliothek	89
	Exportierte Funktionen	91
	Komfort durch einen Dialog	93

	Initialisierung, Enumeration und Shutdown	99
	Zwischen Child-Windows wechseln	106
	Render-Funktionen	107
3.6	Testlauf der Implementierung	109
3.7	Ein Blick zurück, zwei Schritt nach vorn	115
Kapitel 4	Schnelle 3D-Mathematik	117
4.1	Schnell, schneller, am schnellsten	117
	Grundlagen der Assembler-Programmierung	119
	Einführung in SIMD	122
	Wie sag ich's meinem Compiler?	133
	Identifikation einer CPU	134
	Unterstützung für SSE zur Laufzeit überprüfen	140
4.2	Rechnen mit Vektoren	141
	Grundlegende (arithmetische) Operationen	144
	Komplexere Operationen mit SSE-Unterstützung	147
4.3	Rechnen mit Matrizen	153
	Grundlegende Operationen	155
4.4	Rechnen mit Strahlen	158
	Grundlegende Operationen	159
	Kollision mit Dreiecken	161
	Kollision mit Ebenen	162
	Kollision mit Bounding-Boxen	164
4.5	Rechnen mit Ebenen	169
	Grundlegende Operationen	171
	Kollision mit Dreiecken	172
	Kollision zwischen Ebenen	173
	Kollision mit Bounding-Boxen	174
4.6	Rechnen mit AABB und OBB	176
	Grundlegende Operationen und Culling	178
	Kollision mit Dreiecken	182
	Kollision zweier orientierter Boxen	185

	Ebenen einer AABB	185
	Strahl in AABB	186
4.7	Rechnen mit Polygonen	186
	Grundlegende Operationen	187
	Punkte für das Polygon festlegen	189
	Clippen eines Polygons	192
	Culling mit Bounding-Boxen	199
	Kollision mit Strahlen	201
4.8	Rechnen mit Quaternionen	203
	Einführung in den 4D-Raum	203
	Grundlegende Operationen	206
	Multiplikation zweier Quaternionen	207
	Konstruktion aus Euler-Winkeln	208
	Rotationsmatrix zu einem Quaternion	209
4.9	Ein Blick zurück, zwei Schritt nach vorn	210
Teil 2	Rendern von Grafik	213
Kapitel 5	Materialien, Texturen und Transparenz	215
5.1	Mittleres Management	215
5.2	Eine Klasse für Skins	219
	Texturen	219
	Licht und Material	220
	Grundlegende Strukturen	224
	Interface-Definition für einen Skin-Manager	230
	Skin-Manager des Direct3D-Renders	233
	Farben und Materialien vergleichen	237
	Skins austeilen	238
5.3	Skins und Materialien aufnehmen	239
5.4	Texturen aufnehmen	241
	Texturen zu den Skins hinzufügen	242
	Grafikdateien als Texturen laden	245

5.5	Transparenz der Texturen einstellen	249
	Color-Keys über Alpha Channels	252
	Allgemeine Transparenz über Alpha Channels	254
5.6	Ein Blick zurück, zwei Schritt nach vorn	256
Kapitel 6	Das Render-Interface der ZFXEngine	259
6.1	Simplizität versus Flexibilität	259
6.2	Projekteinstellungen	260
6.3	Sicht und Projektion	262
	Multiple Stages	264
	Viewports, Viewmatrizen und das Frustum	266
	Orthogonale Projektion	273
	Perspektivische Projektion	275
	Aktivieren von Sicht und Projektion	278
	Koordinatenumrechnung 2D zu 3D und zurück	282
	Resümee: Sicht und Projektion	286
	Festlegen der Welttransformation	287
6.4	Vertex-Strukturen	288
6.5	Shader-Support	290
	Notwendige Vorbereitungen	291
	Vertex-Shader	295
	Pixel-Shader	301
6.6	Aktivierung von Renderstates	302
6.7	Effizientes Rendern von grafischen Primitiven	307
	Grundlagen zu Hardware und Performance	307
	Caching beim Rendern	310
	Statische vs. dynamische Vertex- und Index-Buffer	311
	Interface-Definition für einen Vertex-Cache-Manager	313
	Vertex-Cache-Objekt	315
	Vertex-Cache-Manager	328
6.8	Rendern von Text, Punkten und Linien	344
	Fonts anlegen und Text rendern	344

6.9	Punktlisten rendern	348
	Linienlisten rendern	350
6.10	Darstellung einer Szene	352
6.11	Demo-Applikation zur Anwendung der DLL	355
	Multiple 3D-Child-Windows mit multiplen Viewports	355
	Einfacher Geometrie-Loader	358
6.12	Ein Blick zurück, zwei Schritt nach vorn	359
Kapitel 7	3D-Pipeline und Shader	361
7.1	Grundlagen von Shadern	361
	3D-Pipeline	362
	CPU-lastig versus GPU-lastig	364
	Vertex-Manipulation über Vertex-Shader	366
	Pixel-Manipulation über Pixel-Shader	368
7.2	Shader-Techniken und Beispiele	369
	Demo 1: Basistransformationen	370
	Demo 2: Single-Pass-Multitexturing	378
	Demo 3: Directional Lighting per Pixel	380
	Demo 4: Per-Pixel-Omni-Lights	383
	Demo 5: Graustufenfilter	393
	Demo 6: Bump-Mapping	394
7.3	Ein Blick zurück, zwei Schritt nach vorn	410
Kapitel 8	Skeletale Animation von Charakteren	411
8.1	Eine Revolution?	411
	Der Siegeszug	413
8.2	Das Modellformat CBF	418
	Was ist ein Chunk?	419
	Einen Chunk auslesen (GetNextChunk)	421
	Unsere Hauptmethode	421
	Den Kopf einlesen (ReadHeader)	422
	Einlesen der Vertices (ReadVertices)	424
	Triangle-Information einlesen (ReadFaces)	427

	Das Netz (ReadMesh)	428
	Auf das Material kommt es an (ReadMaterial)	429
	Die Joints, bitte (ReadJoints)	431
	Der Hauptjoint (ReadJoint_Main)	434
	Die Rotation (ReadJoint_KeyFrame_Rot)	434
	Die Position (ReadJoint_KeyFrame_Pos)	436
	Sei animiert (ReadAnimations)	437
	Passt es? (SetScaling)	439
8.3	Verarbeitung der Daten im Speicher	442
	Vorbereitung der Daten (Prepare)	442
	Skeletale Animation (SetupBones)	447
	Bewegung im Modell (Animation)	452
	Vorbereitung ist alles (AnimationPrepare)	454
	Meine Position (AnimationVertices)	459
8.4	Updaten und Nutzen des Modells	462
	Aktueller Stand (Update)	462
	Präsentationstermin (Render)	463
	Bis auf die Knochen runter (RenderBones)	464
	Voll normal (RenderNormals)	466
8.5	Ein Blick zurück, zwei Schritt nach vorn	468
Teil 3	Hilfsmodule für die Engine	469
Kapitel 9	Eingabe-Interface der ZFXEngine	471
9.1	Kurz und schmerzlos	471
	Altbekanntes Interface-Design	472
	Interface-Definition für eine Eingabe-Klasse	473
9.2	Basisklasse für DirectInput-Devices	474
	Erstellen und Freigeben des Objekts	475
	Inbetriebnahme	476
	Abfrage des Inputs	478
9.3	Ran an die Tasten	481
	ZFXKeyboard-Klasse	482
	Initialisierung und Freigabe	482

	Update	483
	Abfrage des Inputs	484
9.4	Die Rattenfänger von Redmond	485
	ZFXMouse-Klasse	485
	Initialisierung und Freigabe	486
	Update	487
9.5	Kein Spaß ohne Joystick	490
	ZFXJoystick-Klasse	490
	Initialisierung und Freigabe	491
	Update	493
9.6	Implementierung des Interfaces	494
	Initialisierung und Freigabe	495
	Update	497
	Abfrage der Daten	498
9.7	Demo-Applikation zur Anwendung der DLL	499
9.8	Ein Blick zurück, zwei Schritt nach vorn	502
Kapitel 10	Audio-Interface der ZFXEngine	505
10.1	Kurz und schmerzlos	505
	Altbekanntes Interface-Design	506
	Interface-Definition für eine Audio-Klasse	507
10.2	Implementierung des Interfaces	508
	ZFXAudio Klasse	509
	Initialisierung und Freigabe	512
	Das Laden und Abspielen von Sounds	515
	Zuhörer und Quelle für 3D-Sound	520
10.3	Demo-Applikation zur Anwendung der DLL	522
10.4	Ein Blick zurück, zwei Schritt nach vorn	522
Kapitel 11	Netzwerk-Interface der ZFXEngine	523
11.1	Netzwerk-Spiele	523
	Session-basiert	524

	Persistente Welten	526
	LAG	527
11.2	Netzwerk-Architektur	528
	Peer-to-Peer	528
	Client-Server	528
11.3	Netzwerk-Technik	530
	Das OSI-Modell	530
	Protokolle	532
	APIs	533
11.4	Implementierung der Netzwerk-Bibliothek	535
	Altbekanntes Interface-Design	535
	Server versus Clients	536
	Pakete schnüren	536
	Warteschlangen	538
	Socket-Objekte	542
	Interface-Definition für eine Netzwerk-Klasse	557
	Windows Sockets-Kapselung	559
11.5	Demo-Applikation zur Anwendung der DLL	573
	Chatten über das Netzwerk	574
	Dateien versenden	579
11.6	Ein Blick zurück, zwei Schritt nach vorn	588
Kapitel 12	Timing und Bewegung in der ZFXEngine	589
12.1	Hilfsbibliothek ZFXGeneral	589
12.2	Verschiedene Kamera-Modi	590
	Freie Kamera	590
	1st-Person-Kamera	590
	3rd-Person Kamera	591
	Fixe Kamera	592
12.3	Bewegung durch ZFXMovementController	592
	Wozu einen Movement-Controller?	592
	Implementierung der Basisklasse	593

	Ableitung einer freien Kamera	596
	Ableitung einer 1st-Person-Kamera	598
12.4	Demo-Applikation zur Anwendung der Bibliothek	602
12.5	Ein Blick zurück, zwei Schritt nach vorn	602
Kapitel 13	Scene-Management	605
13.1	Sinn des Scene-Managements	605
13.2	Scene-Management-Techniken	607
	Keine Lösung ist auch eine Lösung	608
	Kontinuierliche und diskrete Detail-Level	609
	Quadtrees	611
	Octrees	621
	Binary Space Partitioning Trees	623
	Portal-Engines	634
	Potential Visibility Set	643
13.3	Implementierung eines BSP-Baums	645
	Klassendeklaration	647
	Erzeugen und Freigeben einer Instanz	650
	Rekursives Erstellen des Baums	650
	Auswahl des besten Splitters	654
	Durchlaufen des Baums	655
	Kollisionsabfragen	658
13.4	Implementierung eines Octrees	660
	Klassen-Deklaration	661
	Erzeugen und Freigeben einer Instanz	663
	Initialisieren eines Child-Nodes	664
	Rekursives Erstellen des Baums	665
	Polygonliste auf einen Node beschneiden	667
	Kollisionsabfragen im Octree	670
	Höhe des Spielers im Octree	672
	Durchlaufen des Baums	675

13.5	Demo-Applikation BSP-Tree und Octree	678
	Rendern von ZFXPolygon-Instanzen	679
	Laden der Level-Daten	681
	Berechnung eines Frames	684
	Sehenswertes in der Demo	687
13.6	Ein Blick zurück, zwei Schritt nach vorn	688
Teil 4	Schwarze Magie der Spieleprogrammierung	691
Kapitel 14	Computer Aided Design (CAD)	693
14.1	Anwendungen für CAD-Tools	693
	Ingenieur, Architekt, Spiele-Entwickler	693
	Level-Editing-Tools	694
14.2	Low-Polygon-Editor PanBox Edit	695
	Fähigkeiten des Tools	696
	WinAPI-Rahmenanwendung	698
14.3	Klassen-Design des Tools	698
	Genereller Aufbau eines Levels	699
	Die Grundlage allen Seins: CLevelObject	701
	Auf unterster Ebene: CPolygon	705
	Komplexe Modelle: CPolymesh	725
	Aufbruch in eine neue Welt: CPortal	744
	Es werde Licht: CLight	751
	Interaktive Objekte: CEntity	756
	Ich mach den Fisch: CSpawnPoint	761
	Lokales Management: CSector	762
	Alle zusammen: CLevel	789
	In der engeren Wahl: CSelectionBuffer	793
14.4	Ausgewählte Aspekte des GUI	805
	Klassendeklaration	806
	Wichtige Attribute	809
	Update-Methode	811
	Ein Polygon erstellen	812

14.5	Ein Blick zurück, zwei Schritt nach vorn	814
Kapitel 15	Pandoras Legacy	815
15.1	Der Deathmatch-Ego-Shooter Pandoras Legacy	815
	Einfaches Game-Design	816
	Altbekannte Klassen neu aufgelegt	817
15.2	Im Schatten unser selbst	818
	Verschiedene Verfahren zum Schattenwurf	819
	Zur Theorie der Shadow-Volumes.	820
	Implementierung von Shadow-Volumes.	827
	Erstellen des Shadow-Volumes	831
15.3	Einen Level laden	840
	Benötigte Hilfsdatenstrukturen	841
	Lade-Methode in CGameLevel	842
	Lade-Methode in CGameSector	843
	Verbindungen zwischen den Portalen herstellen.	853
15.4	Einen Level rendern	855
	Rendern der Geometrie	856
	Die Schatten im Level rendern	863
15.5	Integration von Characters	865
	CGameCharacter, die Basisklasse für Spieler und NPCs	866
	Netzwerknachrichten von und für Characters	868
15.6	CGame, die Klasse für das Spiel	870
	Update des Spiels.	870
	Aufgaben für das Netzwerk	872
15.7	Ein Blick zurück, zwei Schritt nach vorn	874
Kapitel 16	Scripting und Definition von NPC-Verhalten	877
16.1	Kontrolle von NPCs.	877
16.2	ZBL/0 ZFX Bot Language – die Scriptsprache für ZFX-Bots	878
16.3	ZBL-API – Integrieren der ZBL/0-Virtual Machine in eigene Projekte	879
	Mehr über das ZBL-API-Interface	879

	Mehr über die Virtual Machine	883
16.4	ZBL/0 Toolkit – die ZBL-Entwicklungsumgebung	892
16.5	ZBL/0-Bot-Design – Entwickeln von Bots mit der ZFX-Bot-Language	894
	Die Scriptsprache ZBL/0 (Syntax)	894
	EZBL/0-Standardbefehle und Anweisungen	898
	Die Entwicklung eines Game-Bots mit ZBL/0	901
16.6	Ein Blick zurück, zwei Schritt nach vorn	910
Teil 5	Anhang	913
Anhang A	Die Funktionen von ZBL/0	915
A.1	House-Keeping Funktionen zur generellen Botsteuerung	915
A.2	Modifikatoren zur Problemspezifikation	917
A.3	Game-Bot Kontrollfunktionen	918
A.4	Game-Bot Sensor-Funktionen	921
A.5	Andere Funktionen in ZBL/0	923
Anhang B	CD-ROM und begleitendes Material zu diesem Buch	925
Anhang C	Internetseiten rund um die Spieleentwicklung	927
Anhang D	Epilog	929
	Stichwortverzeichnis	931

Vorwort

*»In a world designed by the men in grey who decide how we live
in brief, there's a master-plan for the company men from the
cradle to the company-grave.«*

(Die Toten Hosen; Here comes Alex)

Betrachtet man seine eigenen Werke in der Retrospektive, wie ich das hier gerade an einem Dienstagabend um kurz vor 23 Uhr nach dem allwöchentlichen Volleyballtraining tue, so wird man in der Regel eines feststellen: Man ist nicht vollends zufrieden mit dem Werk. Hätte man nur noch einen Tag mehr Zeit oder nur zehn Seiten mehr Platz gehabt – dann hätte das auch nichts geändert. Und das ist auch gut so. Wenn man das Gefühl hat, man hat etwas geschaffen, das nicht mehr verbessert werden kann, dann hat man während des Erschaffungsprozesses selbst nichts gelernt. Das Bestreben, ein hundertprozentig zufrieden stellendes Ergebnis zu erreichen, lässt sich mathematisch auch als die asymptotische Annäherung des Protagonisten an die Perfektion beschreiben. Oder um es ein wenig philosophischer mit den Worten von Michael Abrash zu sagen: *»Perfection is a moving target.«*

Aufbauend auf meiner Erfahrung mit vergangenen Werken und dem überwältigenden Feedback der Leser habe ich mich auf meiner eigenen Kurve ein wenig näher an meine subjektive Perfektion herangetastet. Gleichsam hat sich diese Perfektion aber durch meine im Verlauf dieses Projekts gewonnene Erfahrung von mir wegbewegt und lässt mich, dem Sisyphos gleich, fleißig weiterstreben, das unmögliche Ziel eines fernen Tages zu erreichen. So lange jedoch mögt ihr dieses Buch als das betrachten, was es wirklich ist: ein Buch. Ein umfassendes und komplexes Buch, wohl wahr. Aber dennoch ist es kein perfektes Buch. Es ist keine Bibel für angehende Spieleprogrammierer. Es handelt sich bei dem vorliegenden Werk um ein Buch, das euch viele moderne Thematiken der Spieleprogrammierung aufzeigen und diese implementieren wird. Ihr solltet das Buch daher mehr als eine Sammlung von Ideen und Vorschlägen für Implementierungsoptionen sehen denn als perfekte Step-by-Step-Anleitung für einen einzig richtigen Königsweg – den gibt es nämlich in der Programmierung von Computergrafik nicht. Spaß ist, was ihr draus macht.

Spaß hatte ich auch jede Menge, und zwar beim Schreiben dieses Buches. Viel Zeit hat es gekostet, und viele Dinge sind in dieser Zeit geschehen. Vor allem aber hat mein kleiner Tim in dieser Zeit das Licht der Welt erblickt und mir gezeigt, dass ein kleines Baby-Lächeln weitaus schöner ist als das am perfektesten gerenderte Bild einer 3D-Szene mit Per-Pixel-Lighting, Realtime-Shadows und Fullscreen-Glow-Effekt.

Neben dem kleinen Tim haben aber noch etliche andere Personen dazu beigetragen, dass aus diesem Buch das geworden ist, was ich mir darunter vorgestellt habe. An erster Stelle möchte ich mich dafür bei Oli bedanken, der das Kapitel über Character-Animation geschrieben hat und der mir einen Haufen Arbeit am Kapitel über Netzwerke abgenommen hat. Als Nächstes gilt mein Dank Eike, der in etlichen Nachtschichten die Script-Sprache für dieses Buch entworfen und implementiert und das Kapitel über gescriptete KI zu diesem Werk beigetragen hat. Auf Seiten von Markt+Technik danke ich insbesondere Herrn Marcus Beck und Herrn Boris Karnikowski, die sich immer unverzüglich und freundlich um meine Wünsche und Bedürfnisse gekümmert haben. Friedericke Daenecke war so freundlich, mein Manuskript aus einem Wust kommata- und bindestrich-freier Sätze von neuartigen, kreativen Permutations-Ideen für konservative Buchstabenkombinationen in ein lesbares Werk deutscher Sprache zu verwandeln. Dies tat sie mit viel Geduld und Humor, und daher werde ich mich hüten in meinem nächsten Spiel ein Alien zu programmieren, welches die Aufschrift »Korrektorin« auf der Jacke trägt. Was die technische Seite angeht, so muss ich mich bei Marco Kögler bedanken. Er hat mir immer sehr aufmerksam zugehört, wenn ich ihm von den Abläufen meiner Engine berichtet habe, und meine Ideen und Vorstellungen mit hoher fachlicher Kompetenz kommentiert. Den Toten Hosen danke ich für die Musik, die mich (nicht nur) während des Schreibens bei Laune hält, und Jacobs für den löslichen Kaffee, der das Programmieren überhaupt erst ermöglicht.:-)

Last but still not least danke ich Ulrike, die das hier möglich gemacht hat.

Stefan Zerbst

*»Tüchtigkeit ist weder Fähigkeit noch Begabung,
sondern Gewohnheit.«*

(Peter F. Drucker)

Heute ist es wichtig, Spiele zu haben, die den Spieler faszinieren. Dabei gehören natürlich solche elementaren Teile wie Netzwerk und Character-Animation zu den Standards im Spiel. Das Ziel ist dabei, so viel Interaktion mit der virtuellen Welt wie möglich zu visualisieren.

Mich persönlich hat die Animation von Charakteren in Spielen schon immer gefesselt. Sei es nun, dass sich Mitte der 80er Jahre kleine animierte Bilder mit dem Namen Sprites über den Screen geschoben haben oder dass sich heute hoch präzise animierte Charaktere durch virtuelle Welten bewegen.

Es hat viel Freude bereitet, an diesem Buch mitzuwirken, und ich hoffe, es macht euch genauso viel Spaß, es zu lesen.

Oliver Düvel

*»We didn't start the fire, it was always burning
since the world's been turning«*

(Billy Joel)

Seit *Quake* von *id Software* A.D. 1996 die Welt erschüttert hat, ist es Standard für First-Person-Shooter, eine eigene Scriptsprache für die Definition von Bot-Verhalten (oder sogar für die Steuerung der Engine selbst) in die Game-Engine zu integrieren. So ist es auch selbstverständlich, dass die Engine, die in diesem Buch entwickelt wird, eine Schnittstelle zu einer Scriptsprache bereitstellt, nämlich die ZFX-Bot-Language ZBL/0, oder einfach Sybil. In dem Kapitel zu ZBL/0 versuche ich, euch die Grundlagen der Entwicklung von Game-Bots unter der Anwendung von ZBL/0 nahezubringen. Nach einem kurzen Ausflug in die Theorie von Scriptsystemen wird die Sprache ZBL/0 selbst unter die Lupe genommen, und ihr findet mehrere praktische Beispiele zur Programmierung von Game-Bots. Mit diesen Kenntnissen und euren eigenen Ideen sollte es dann ein Leichtes für euch sein, die von euch geschaffenen Spielwelten mit selbst definierten Game-Bots zu bevölkern.

Eike Anderson

Einleitung

»Wir vermarkten nicht bereits entwickelte Produkte, sondern wir entwickeln einen Markt für Produkte, die wir herstellen.«

(Akio Morita)

Inhalt dieses Buches

Wer das liest, ist doof. Na gut, das nehme ich wieder zurück. Anhand der Differenz zwischen verkauften Büchern und den E-Mails mit Protesten gegen diesen Einleitungssatz kann ich aber diesmal leicht herausfinden, wer sich die Mühe gemacht hat, die Einleitung zu lesen.

Test

In den Einleitungen verschiedener Bücher stehen verschieden wichtige Sachen. Ich bemühe mich immer, in die Einleitung wichtige Informationen zu packen, die der Leser vor Beginn der Lektüre wissen sollte. Leider zeigt die Erfahrung aber, dass die Einleitung von vielen Lesern einfach übergangen wird. »*Ich dachte, da steht eh nur Blabla drin ...*« oder »*Ich wollte einfach gleich mit dem Programmieren loslegen...*« sind die Sätze, die man als Entschuldigung immer wieder hört, wenn man bei Nachfragen zu bestimmten Dingen einfach auf die entsprechende Seite seiner Einleitung verweist. Ich bin jedoch immer wieder voller Hoffnung, dass meine einleitenden Worte von den Lesern wahrgenommen werden. Ich für meinen Teil fand es jedenfalls immer recht interessant, was ein Autor über das Werk zu sagen hat, das er im Verlauf einiger Monate erschaffen hat.

*Sinn und Zweck
von Einleitungen*

Was steht in diesem Buch?

Die wichtigste Frage, die ich hier klären möchte, ist die Frage nach dem, was euch auf den nächsten neunhundert Seiten erwarten wird. Es liegt in der Natur des Menschen, neugierig zu sein, und diese Neugierde möchte ich hier zum einen befriedigen, und zum anderen möchte ich auch Lust auf mehr machen.

Wissen ist Macht

Dieses Buch soll es euch ermöglichen, moderne Computerspiele auf einem grafisch sehr hohen Niveau zu entwickeln. Um dieses Ziel zu erreichen, müssen wir zunächst ein paar kleinere Komplementärziele erreichen. Wir werden damit beginnen, uns mit reiner 3D-Grafik und 3D-Mathematik zu

*Von 3D-Grafik bis
zum Ego-Shooter*

befassen. Danach benötigen wir Fachwissen aus den verschiedensten Bereichen, beispielsweise die Abfrage von Input über Tastatur, Maus und Joystick oder das Versenden von Daten über ein lokales Netzwerk (LAN) oder das Internet. Wenn wir dies gemeistert haben, kümmern wir uns auch gleich noch um die Entwicklung von Tools, wie beispielsweise einem grafischen Level-Editor, ohne die man bei der Entwicklung eines Spiels nicht weit kommen wird.

*Abgerechnet wird
am Schluss*

Nachdem wir uns Stück für Stück durch all diese kleinen Themen des Buches gekämpft haben, erreichen wir schließlich das große Finale. Dann geht alles auf einmal sehr schnell. Ihr werdet sehen, wie plötzlich all die kleinen Teile, an denen wir vorher so hart gearbeitet haben, ineinander greifen. In einem ganz natürlichen Prozess werden sich unsere bis dahin entwickelten Module mit unserem neu erworbenen Wissen verbinden und tatsächlich einen Multiplayer-Ego-Shooter ergeben, in dem wir über ein Netzwerk mit unseren Freunden und Feinden um die Herrschaft in der virtuellen Welt kämpfen können.

Das gibt es hier

Dazu beinhaltet dieses Buch sämtlichen Quellcode, der für die einzelnen Module und Zwischenschritte nötig ist. Ihr werdet hier also *on-the-fly* lernen, Direct3D, DirectMusic, DirectInput und WinSock anzuwenden. Ebenso werdet ihr auch lernen, wie man Interface-Definitionen erstellt und diese dann implementiert. Man kann also sagen, dass dieses Buch euch alles beibringt, was nötig ist, um ein vergleichsweise umfangreiches Spiel zu entwickeln.

Was steht nicht in diesem Buch?

*Wir sind doch
keine Anfänger
mehr, oder?*

Es gibt aber auch ein paar Dinge, die nicht in diesem Buch stehen. Was ihr hier in den Händen haltet, ist ein Buch über die Entwicklung von Computerspielen für fortgeschrittene Programmierer. Es handelt sich explizit **nicht** um ein Buch, das euch beibringen soll, mit DirectX oder OpenGL zu programmieren. Ich setze hier voraus, dass der Leser wenigstens grundlegende Kenntnisse eines DirectX-Interfaces hat idealerweise natürlich des Direct3D-Interfaces. Es ist aber vollkommen ausreichend, wenn ihr schon einmal ein Direct3D-Programm erstellt habt, das ein texturiertes Rechteck auf den Bildschirm rendert. Dann seid ihr durchaus in der Lage, dem Code in diesem Buch aufmerksam zu folgen und ihn zu verstehen.

Nachsitzen

Wem diese Voraussetzung fehlt, der möge nun nicht verzweifeln. Wie bereits erwähnt, ist hier sämtlicher Code enthalten und gezeigt. Es wird nichts verschwiegen. Der Fokus der Erklärungen liegt jedoch bei den entsprechenden Methoden immer auf dem Design der Methode und ihrer Klasse, nicht auf der umfassenden Erläuterung des DirectX-Codes. Ein helles Köpfchen wird dennoch mit diesem Buch auskommen, denn auf der

Internetseite <http://www.zfx.info> befinden sich Online-Tutorials, die euch die Anwendung von Direct3D so weit nahe bringen, dass ihr fit genug werdet, um mit diesem Buch etwas anfangen zu können.

Es ist jedoch unerlässlich, dass der Leser dieses Buches fundierte Kenntnisse der Programmiersprache C++ hat. Es gibt hier keine Einführung in die Grundlagen der Programmierung an sich oder in die Sprache C++ im Speziellen. Wem diese Voraussetzung fehlt, der wird in diesem Buch hoffnungslos verloren sein. So viel kann ich jetzt schon versprechen. Man sollte sich also hier nicht selbst maßlos überfordern und das Buch am Ende frustriert in die Ecke stellen, sondern sich gegebenenfalls vorher ein Buch über die Programmierung mit C++ zulegen.

*C++ und WinAPI
müssen aber sein*

Und noch etwas sei hier klargestellt: Es gibt so viele verschiedene Arten von Computerspielen – von rein zweidimensionalen Spielen bis hin zur 3D-Grafik, und dort kennen wir Ego-Shooter, Adventures, Sportspiele, Simulationen und noch vieles mehr. Wir werden uns hier auf die Königsklasse beschränken, also 3D-Ego-Shooter im Stil von *Doom* und *Half Life*.

Fokus auf Indoor

Konzeptionelle Vorgehensweise

Das Konzept dieses Buches ist recht einfach erklärt. Das Buch gliedert sich in vier Teile, die von euch auch in der entsprechenden Reihenfolge bearbeitet werden sollten.

*Ein Schritt nach
dem anderen*

Der erste Teil beinhaltet ein paar Grundlagen, die als Vorwissen für die folgenden Kapitel notwendig sind. Er beginnt mit einem allgemeinen Einstieg in die Thematik der Spieleprogrammierung und geht dabei auch auf organisatorische und betriebswirtschaftliche Aspekte bei der Entwicklung von Computerspielen ein. Daran schließt sich ein Kapitel über den Aufbau der im Folgenden entwickelten Engine namens *ZFXEngine 2.0* an. Dieses Kapitel bietet euch eine Roadmap, wie das Design unserer Engine aussehen wird. Das dritte Kapitel befasst sich dann mit den Grundlagen der API-unabhängigen Programmierung von Interface-Klassen. Eines der Ziele dieses Buches ist die Entwicklung einer API-unabhängigen Engine, und dort werdet ihr lernen, wie man so etwas macht. In diesem Kapitel geht es das erste Mal auch an die eigentliche Programmierung. Das vierte Kapitel, das den ersten Teil abschließt, behandelt dann in einem Schnellkurs die notwendige 3D-Mathematik. Der Schwerpunkt liegt dabei auf der Anwendung schneller Operationen in möglichst umfassenden Klassen für Vektoren, Matrizen usw.

Erster Akt



Die Abkürzung API steht für Application Programming Interface und bezeichnet i.d.R. eine Bibliothek, die eine bestimmte Funktionalität implementiert und dem Anwender dieser Bibliothek ein Interface für die Arbeit zur Verfügung stellt. Beispiele für APIs sind die WinAPI, die die Zusammenarbeit mit dem Betriebssystem Windows ermöglicht, oder die Direct3D- und die OpenGL-API, die die Zusammenarbeit mit der Grafikkarte ermöglichen.

Zweiter Akt Im zweiten Teil des Buches geht es dann um das umfassendste Interface unserer Engine, und zwar um das Render-Device. Auch dieser Teil umfasst vier Kapitel, die sich alle mit verschiedenen Aspekten des Renderns von Grafik befassen. Es geht um das Laden von Texturen und Materialien, das Rendern von Dreiecken auf dem Bildschirm, die Verwendung von Vertex-Shadern und Pixel-Shadern und zu guter Letzt um das Animieren von 3D-Modellen. Das letzte Thema ist zwar nicht direkt mit dem Rendern von Grafik verbunden, passte aber thematisch am besten in diesen Teil.

Dritter Akt Der dritte Teil dient dann der Erweiterung unserer Engine um weitere dringend notwendige Module. Hier werden in jeweils einem Kapitel die drei Komponenten für Input, Audio und Netzwerk der *ZFXEngine 2.0* programmiert. Daran schließt sich ein Kapitel mit der Implementierung eines Timers und einer Kamerafunktionalität an, damit wir auch Bewegung in den Spieler bringen können. Zu guter Letzt bietet dieser Teil ein umfangreiches Kapitel über Scene-Management-Techniken, die dann zum Teil als zusätzliche Komponenten in unsere Engine integriert werden.

Vierter Akt Im vierten und letzten Teil des Buches geht es dann nicht mehr um die Entwicklung der Engine, sondern um ihre Anwendung in zwei umfassenden Life-Fire-Projekten. Das erste Projekt ist die Programmierung eines funktionalen Tools mit grafischer Benutzeroberfläche. Dieses Tool kann als Modell-Editor oder als vollwertiger Level-Editor verwendet werden. Das zweite Projekt ist dann ein Netzwerk-Deathmatch-Spiel, das ebenfalls unter Zuhilfenahme unserer Engine implementiert wird. Nach der Bearbeitung des vierten Teils werdet ihr nicht nur gesehen haben, wie leicht es einem unter Verwendung einer sinnvoll abstrahierten Engine fällt, ein Spiel zu programmieren. Man kann sich dabei einfach auf das Wesentliche konzentrieren und muss sich nicht mit Lowlevel- Funktionen zur Ausgabe von Grafik oder Abfrage des Joysticks herumärgern. Ihr werdet aber auch gesehen haben, dass eine sehr weit von einer API abstrahierende Engine gewisse Einschränkungen hat und dass auch der Engine in diesem Buch sicherlich einiges an Funktionalität fehlen wird, die ihr gerne hättet. Aber in den ersten drei Teilen des Buches werdet ihr genug gelernt haben, um diese Erweiterungen für eure eigene Version der Engine vornehmen zu können.

Verwendete Werkzeuge

Neben unserem Kopf, der das wichtigste Werkzeug für die Entwicklung von Computerspielen ist, benötigen wir insbesondere eine so genannte **IDE**, also ein Integrated Development Environment (dt. integrierte Entwicklungsumgebung). Diese Umgebung heißt so, weil sie verschiedene Programme integriert. Dies ist vornehmlich ein Texteditor mit Syntax-Highlighting für Programmiersprachen zusammen mit einem Linker und einem Compiler sowie einem Debugger. Eine der bekanntesten IDEs ist natürlich das *Visual Studio* von Microsoft. Abbildung E.1 zeigt einen Screenshot von *MS Visual C++ 6.0* in der *Standard Edition*. Dies ist die IDE, mit der ich die Beispiele für dieses Buch entwickelt habe. Genau eine solche IDE benötigt ihr auch, wenn ihr die Beispiele aus diesem Buch selbst nachvollziehen wollt. Die IDEs von Microsoft eignen sich natürlich am besten für die Arbeit mit *DirectX*, da dieses auch von Microsoft ist.

IDE

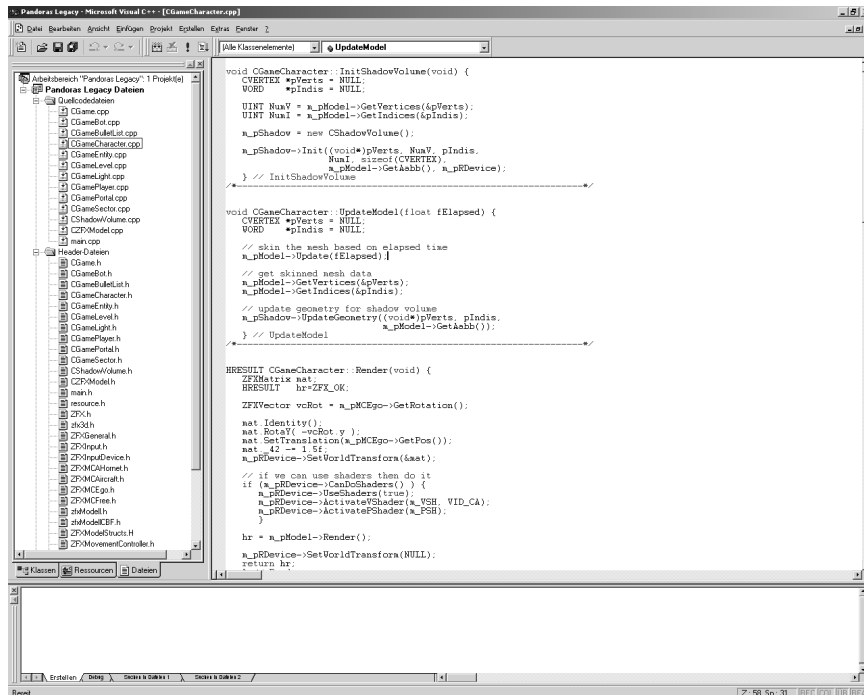


Abbildung E.1: Screenshot der Entwicklungsumgebung Microsoft Visual C++ 6.0 Standard Edition.

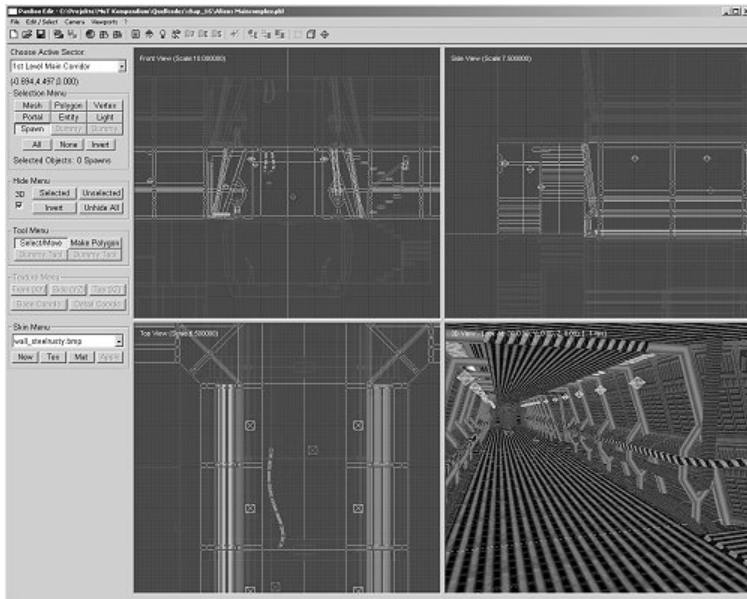
Wir werden zwar unsere Engine, die im Verlauf dieses Buches entstehen wird, API-unabhängig entwickeln. Trotzdem müssen wir diese unabhängigen Interface-Klassen mit mindestens einer (beliebigen) API implementieren. In diesem Buch verwende ich dafür *DirectX* in der Version 9. Ihr benötigt daher auch das *DirectX* SDK in der Version 9 oder aktueller. Damit ergibt sich auch die Voraussetzung für das Betriebssystem, denn *DirectX* 9 läuft nur auf Windows-Betriebssystemen in den Versionen 98SE, 2000, ME und XP.

DirectX 9 und
Windows

Modell-/Level-Editor

Mit der IDE und DirectX haben wir dann eigentlich schon alles zusammen, was wir benötigen. Aber es ist sicherlich auch reizvoll, wenn ihr beispielsweise eigene Level für das am Ende des Buches entwickelte Spiel basteln könntet. Dazu benötigt ihr einen entsprechenden Level-Editor. Und wie es der Zufall so will, bekommt ihr den auch frei Haus geliefert, nämlich auf der CD-ROM zu diesem Buch. Oder besser gesagt: Wir machen uns die Arbeit und programmieren diesen Editor einfach selbst. Abbildung E.2 zeigt einen Screenshot des Editors, den wir am Ende dieses Buches programmieren werden.

Abbildung E.2:
Screenshot des
Tools PanBox Edit



Hardware-Voraussetzungen

Gute Grafikkarte

Natürlich programmieren wir hier nicht mehr für Hardware-Relikte wie beispielsweise den TNT-Chip. Als Voraussetzung, um alle Beispiele dieses Buches auch ausführen zu können, braucht man unbedingt eine gute Grafikkarte. Als gut definiere ich dabei etwas in der Klasse einer *GeForce 3 Ti*-Karte. Unabdingbar ist aber auf alle Fälle, dass die Grafikkarte Vertex-Shader und Pixel-Shader jeweils in der Version 1.1 unterstützt. Ansonsten werden euch bei einigen der späteren Beispiele in diesem Buch viele Effekte fehlen, die insbesondere die Beleuchtung betreffen. Ansonsten kommt man mit einem Prozessor von mindestens 1 GHz und 256 Mbyte RAM sicherlich aus. Aber natürlich gilt: Je mehr man auffährt, desto besser ist das.

Was ist ZFX?

Im Verlauf dieses Buches werdet ihr immer wieder auf das Wort bzw. die drei Buchstaben **ZFX** stoßen. Viele Leute unterstellen diesen harmlosen drei kleinen Buchstaben immer wieder, ein Akronym zu sein. Dem ist aber nicht so. Es handelt sich dabei einfach um drei Buchstaben, die sich aus einer historischen Präfixbenennung der codierten Funktionen des Autors entwickelt hat. Doch was ist ZFX denn nun? Diese Bezeichnung steht für das Team von Personen, die sich hinter der Entwickler-Internetseite <http://www.zfx.info> verbergen. Dies sind im Moment Oliver Düvel, Eike Anderson, Steffen Engel und meine Wenigkeit – Stefan Zerbst.

Das wohl meist-gebrauchte Kürzel in diesem Buch

Herzlich Willkommen
Du willst, daß Du von 3D Computergrafik besessen bist, wenn Du draußen spazieren gehst, zu den Bäumen aufblickst und Du denkst: "Wow! Ist das eine geile Aufführung..."

Letzten 3 News
Grossteil der Gamer sind Frauen. (28.08.2003, 13:31 Uhr von Swordfighter)
Leut der Reuters Presseagentur hat eine aktuelle Studie festgestellt... (Quelle: Link)
Softimagine Xot nicht-kommerzielle Version (27.08.2003, 14:50 Uhr von Swordfighter)
Sofimagine, einer der weltweit bekanntesten Hersteller von 3D Anim. (Quelle: Link)
Lara Croft - Flash Games (27.08.2003, 13:11 Uhr von Swordfighter)
Als Werbung fuer den britischen Energiedrink Lucozade hat der Hersteller eine Webseite mit on-line... (Link)
Ältere News

DirectX
Buch Tip für directx9 (Heute, 07:23:11 Uhr von Stefan Zerbst)
DirectDraw7 - viele kleine ode... (Heute, 00:04:29 Uhr von Timetvisten)
CaptureStateBlock (29.08.2003, 23:41:51 Uhr von Richard Schubert)

OpenGL
Alpha-Blending + Farbe immer g... (Heute, 00:56:07 Uhr von Nameless Fox)
keine Anzeige im Ortho Modus (29.08.2003, 21:06:58 Uhr von BlzMat)
Teil Textur Kacheln (29.08.2003, 17:37:37 Uhr von Fastack)

Allgemeine Programmierung
[Engine Design] Input Interfac... (Heute, 00:23:03 Uhr von Harvick)
Callstack herausbekommen (Heute, 00:21:00 Uhr von Cygon)
Faszination Log (29.08.2003, 23:52:11 Uhr von smurfer)

Spezielle Programmierung
Parallele-Schnittstelle (29.08.2003, 20:22:15 Uhr von Richard Schubert)
BIOS Einstellung auslesen (29.08.2003, 19:02:53 Uhr von TheDiCodes)
x86 Assembler-Bücher Raum... (28.08.2003, 23:16:55 Uhr von iNcAnE)

Community-Engine
Chatsitzung vom 24.08.2003 (25.08.2003, 11:51:00 Uhr von kimmi)
Chatsitzung am Sonntag, den 17... (16.08.2003, 15:46:54 Uhr von Odinkrieger)
Webseitenupdate --- Was muss r... (13.08.2003, 21:23:22 Uhr von kimmi)

Off-Topic
2 Monitore an einer GraKa (Heute, 05:21:42 Uhr von scalas)
OrcWar Test PreAlpha Release 1... (Heute, 01:45:30 Uhr von Hygma)
Ich gebs auf ... (Heute, 01:27:16 Uhr von ActionHeidi)

Besucher auf dem Board: 8
Momentan sind 5 Mitglieder und 3 Gäste online.
@uzing16, scalar, phil, wendy, Godhand, Stefan Zerbst

Umfraage
Hast du schon mal eine 3D Engine programmiert?
Ich arbeite gerade drann. (76) (35%)
Ja. (48) (22%)
Nein. (46) (21%)
Nein, ist aber geplant. (45) (21%)
Gesamt: 215

Abbildung E.3:
Die Internetseite von ZFX (<http://www.zfx.info>)

Wir von ZFX beschäftigen uns mit der hobbymäßigen Entwicklung von Computerspielen, wobei wir einen besonderen Schwerpunkt auf die kostenfrei zugängliche Ausbildung von zumeist noch jugendlichen Nachwuchs-Programmierern legen. Im Rahmen dieser Hilfestellung für die oftmals abfällig »Noobs« oder »Newbies« genannten Nachwuchstalente ist unter anderem dieses Buch hier entstanden, das den interessierten Nachwuchs-Programmierern als einführende Quelle in die Thematik der Entwicklung von Computerspielen dienen soll.

www.zfx.info

Historie von ZFX

Wie alles anfang

Im Juni 1999 wurde erstmals die Internetseite von ZFX, damals als persönliche Internetseite meiner Wenigkeit, auf den Servern der TU Braunschweig online geschaltet. Sie offerierte den Besuchern der Seite eine Fachartikelserie über die Programmierung von 3D-Computergrafik für den Entertainment-Bereich. Dies führte ohne jede Werbemaßnahme nur knapp ein Dreivierteljahr später zu einem konstanten Besucherstrom von 1000 Zugriffen pro Monat, die durch Suchmaschinen auf das Angebot aufmerksam wurden. Das große Interesse an dieser Thematik mündete schließlich in der Veröffentlichung eines entsprechenden Fachbuches, das im Dezember 2000 erschien und somit das erste originär deutschsprachige Buch zu dieser Thematik war. Erst zu dieser Zeit wandelte sich dann auch die Internetseite von einem rein statischen Informationsangebot zu einer einfachen Form einer Virtuellen Community, eingeleitet im Januar 2001 durch die Online-Schaltung eines Forums als öffentliche asynchrone Möglichkeit der Kommunikation, die bisher nur auf E-Mails und einen unregelmäßigen Chat beschränkt war.

Freiheit der Information

Neben dieser Erweiterung der Kommunikationsmöglichkeiten wurde der Content der Seite ständig erneuert und erweitert. Im Juni 2002 erschien dann eine Fortsetzung zu dem ersten Buch, parallel zu einer technischen und inhaltlichen Überarbeitung der Seite, um den gestiegenen Anforderungen durch die stetig wachsende Community zu entsprechen. Der große Erfolg der gesamten Internetseite und die Entwicklung hin zu einer echten Virtuellen Community war nur deshalb möglich, weil die Internetseite vom ersten Online-Gang an interessanten und vor allem einzigartigen, freien Content zu bieten hatte.

Entwickler-Community ZFX

Online-Wissensbasis

Der Kern von ZFX ist die *ZFX Community*. Dabei handelt es sich um viele Gleichgesinnte, die tagtäglich unsere Internetseite <http://www.zfx.info> besuchen. Auf dieser Internetseite bieten wir unter anderem Tutorials, Open Source und Screenshots von Hobby-Projekten an. Den eigentlichen Reiz der Seite machen jedoch die vielen thematisch verschiedenen Foren der Webseite aus. Dort diskutieren täglich um die 800 Besucher der Seite über alle möglichen Themen und Fragestellungen rund um die Entwicklung von Computerspielen.

Jeder ist willkommen

Eine der großen Stärken von ZFX ist, dass die Community besonders anfängerfreundlich ist. Die mittlerweile weit über zweitausend Mitglieder der Community beantworten auch gern die Fragen von Neueinsteigern und reichen ihnen eine helfende Hand, wenn diese an für sie scheinbar unüberwindliche Hürden stoßen, die ein erfahrenerer Programmierer recht zügig überwinden kann. Wenn ihr diese Internetseite bisher noch nicht besucht habt, so seid ihr hiermit herzlich eingeladen. Der Besuch lohnt sich.

Entwickler-Event zfxCON

Eines der Ziele von ZFX ist es, dass wir alle voneinander lernen können. Wie sollte das wohl besser gehen, als auf einem Treffen, bei dem man im realen Leben von Angesicht zu Angesicht über die Entwicklung von Computerspielen schwatzen kann? Zu diesem Zweck findet einmal im Jahr die so genannte **zfxCON** statt. Dabei handelt es sich um ein Treffen von Mitgliedern der ZFX Community, auf dem verschiedene Speaker aus der Community Vorträge zu bestimmten Programmiertechniken und Spezialeffekten halten. Zu diesem Event ist jeder herzlich willkommen, der sich einfach nur für die Themen der Vorträge interessiert und daraus etwas lernen möchte oder der selbst einen Beitrag leisten kann. Aktuelle Informationen zu der zfxCON findet ihr auf der entsprechenden Internetseite <http://www.zfx-con.de>. Ich hoffe, wir sehen uns auf der nächsten zfxCON und wünsche euch nun viel Spaß bei der Lektüre dieses Buches. ;-)

*Event für Hobby-
Entwickler*

Teil 1 Einführung in die Thematik

Kapitel 1:	3D-Engines und Spieleprogrammierung	37
Kapitel 2:	Design der ZFXEngine	65
Kapitel 3:	Rahmenanwendung der ZFXEngine	73
Kapitel 4:	Schnelle 3D-Mathematik	117

1 3D-Engines und Spieleprogrammierung

» Wir würden es ganz gut finden, wenn man uns heute nur als Objekte der menschlichen Begierde betrachtet und nicht als Musiker.«
(Campino auf dem Frauenkonzert in Rottweil)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Was ist eine Engine?
- ➔ Einführung in die Welt der APIs
- ➔ Historisches über DirectX
- ➔ Betriebswirtschaftliche Aspekte der Spiele-Entwicklung

1.1 Modewort 3D-Engine

Die Bezeichnung *Engine*, oder gar *3D-Engine*, erfreute sich in den Kreisen der angehenden Spieleprogrammierer in den letzten Jahren immer größerer Beliebtheit. Doch nicht immer wussten diejenigen, die diese Begriffe verwendeten, was damit ursprünglich gemeint war oder auch heute noch gemeint sein sollte. Einleitend wollen wir also erst einmal die Frage klären, was man überhaupt unter dem Begriff *Engine* versteht. Schließlich wird es in diesem Buch ausschließlich darum gehen, eine vernünftige *Engine* für die Entwicklung von Computerspielen zu programmieren und diese dann natürlich auch sinnvoll einzusetzen.

Ey, Alter, ich hab da 'ne Engine

Aus dem Englischen übersetzt, bedeutet das Wörtchen *Engine* zunächst einmal nichts anderes als *Motor*. Im Kontext mit der Programmierung bezeichnen wir also die Teile unseres Programms, die eine gewisse Funktionalität des Programms quasi antreiben, als Motoren oder eben als *Engines*. Diese Begrifflichkeit ist hier nahe liegend, denn wenn wir den Schlüssel in das Zündschloss eines Autos stecken und ihn umdrehen, dann erwarten wir von der *Engine* des Autos, dass sie das Auto in Fahrbereitschaft versetzt und wir nur noch auf das Gaspedal treten müssen, damit die *Engine* Bewegungsenergie auf die Antriebsachse bringt und das Auto sich bewegt.

Ganz analog starten wir beispielsweise unsere 3D-Engine, indem wir die `Schlüssel_im_Zündschloss_umdrehen()` Funktion aufrufen. Dann erwarten wir von der 3D-Engine, dass sie die Grafikkarte in Ausgabebereitschaft versetzt. Jetzt können wir auf das virtuelle Gaspedal treten und unsere 3D-Modelle zur Grafikkarte pumpen. Die 3D-Engine hat dann dafür zu sorgen, dass diese 3D-Modelle auf dem zweidimensionalen Monitor erscheinen, und zwar *pronto*, also so performant wie möglich.

*Ein Irrtum kommt
selten allein*

So viel zum Grundprinzip. Nun zu den Missverständnissen. Heutzutage gibt es bereits einige Bibliotheken, die die grundlegendsten Funktionen einer 3D-Engine übernehmen können, und sogar solche, die schon umfassendere Funktionen implementieren. Als Beispiele sind hier `DirectGraphics`, `OpenGL`, `Java3D` oder `OpenInventor` zu nennen. Schließen wir die eher umfassenderen Ansätze einmal aus und konzentrieren wir uns auf die beiden großen APIs (Application Programming Interface) *DirectGraphics* und *OpenGL*, dann haben wir mehr oder weniger integrale Bestandteile der Grafikprogrammierung, die sich über Treiber und DLLs nahtlos in die Schnittstelle zwischen Grafikkarte (Hardware) und Betriebssystem bzw. Applikation (Software) einfügen.

Bei solchen APIs handelt es sich aber eben nicht um 3D-Engines. Vielmehr sind sie eigentlich nur dazu da, dreidimensionale Daten zu projizieren und mit schönen Farben versehen als zweidimensionale Pixeldaten auf dem Bildschirm anzeigen zu lassen. Man spricht hier besser von einer *3D-Pipeline*, durch die die dreidimensionalen Daten auf dem Weg vom Programm zum Monitor müssen. Zu oft entdeckt man jedoch im Internet kleine Progrämmchen, die gerade mal ein rotierendes Dreieck auf den Bildschirm bringen. Durch Verwendung einer entsprechenden API ist das mit wenigen Zeilen Quellcode möglich, wie wir schon bald sehen werden. Solche Progrämmchen nennen sich jedoch bereits stolz »3D-Engine« und haben kilometerlange Feature-Listen. Beispielsweise wird man Folgendes dort lesen können:

Meine 3D-Engine namens `MegaMonsterKiller3D` hat folgende Features:

- ➔ Hardware-Rasterisierung von Triangles
- ➔ Hardware-Transformation and Lightning
- ➔ Texture-Filtering und Antialiasing
- ➔ Texture-Blending von bis zu acht Texture-Stages usw.
- ➔ usw. ...

Da fragt man sich natürlich, wo all diese Features stecken und warum diese tolle Engine bisher nur ein Dreieck anzeigt. Nun, in ihrem Enthusiasmus verwechseln die meisten Einsteiger eine 3D-Engine mit der 3D-Pipeline. All diese schönen Features kann niemand dem Programm absprechen, aber sie

wurden nicht durch den Programmierer implementiert, sondern werden von der 3D-Pipeline-API (DirectGraphics oder OpenGL) zur Verfügung gestellt, ohne dass man dafür eine Zeile Code schreiben müsste. Ein Programm, das ein rotierendes Dreieck anzeigt, ist daher nicht als 3D-Engine zu bezeichnen, sondern allenfalls als kleines Testprogramm, wie man Daten über die 3D-Pipeline zur Anzeige am Monitor bringen kann.

Von dem allgemeinen Begriff Engine wird sicherlich jeder eine andere Vorstellung haben. Es gibt aber ein paar Dinge, die man einer Engine grundsätzlich anlasten können sollte. Die folgende Liste soll ein paar Aufgaben zeigen, die alle in den Verantwortungsbereich einer Engine fallen.

*Ja, und was ist
nun eine Engine?*

Eine Engine muss ...

- ➔ ... alle Daten in ihrem Aufgabenbereich verwalten.
- ➔ ... alle Daten entsprechend ihres Aufgabenbereichs verarbeiten.
- ➔ ... alle Daten nach der Bearbeitung an die entsprechenden nachgeordneten Kompetenzen weiterleiten, wenn das nötig ist.
- ➔ ... alle Daten zur Verwaltung und Bearbeitung von vorgelagerten Kompetenzen übernehmen.

Wie habe ich solche Definitionen beispielsweise in BWL- oder Marketing-Lehrbüchern immer gehasst, weil jeder Autor sich nach seinem Gutdünken eigene Definitionen ausdenkt, weil alle anderen 8.712.984 Definitionen, die es schon gibt, seiner Meinung nach gravierende Fehler enthalten. Und jetzt schreibe ich sie selbst! Also sehen wir die obige Liste nicht als Definition mit Anspruch auf alleinige Gültigkeit an. Betrachten wir sie eher als ganz allgemeine Beschreibung, die uns beim Verständnis der Arbeit einer Engine helfen soll. Damit das klappt, konkretisieren wir diese abstrakte Liste im Folgenden anhand des Beispiels einer Sound-Engine.

Ja, wir haben richtig gelesen. Es gibt den Begriff Engine eben nicht nur in Bezug auf 3D-Engines. Ebenso gut können wir eine Sound-Engine haben oder eine Input-Engine oder eine Netzwerk-Engine usw. Aber bleiben wir beim Sound. Wir schreiben also ein Programm in Form einer Bibliothek, die sich Sound-Engine nennt. Unser Programm muss in der Lage sein, der Sound-Engine gewisse Daten und Anweisungen zu übermitteln, – beispielsweise den Namen einer Sound-Datei zusammen mit der Anweisung, diese zu laden. Mehr möchte unser Programm mit dem Sound zunächst nicht zu tun haben, denn dafür ist ja die Engine da. Diese kümmert sich nun darum, dass sie den entsprechenden Sound lädt und in eigenen Datenstrukturen abspeichert und verwaltet. Ebenso obliegt es ihrer Verantwortung, diesen Sound an die nachgelagerte Instanz (z.B. eine Sound-API wie DirectAudio oder Windows-Multimedia) zum Abspielen weiterzureichen, wenn die vorgelagerte Instanz (beispielsweise unser Programm) die Anweisung dazu erteilt.

*Ein kleines
Beispiel*

*Und eine
3D-Engine?*

Dies war ein zugegebenermaßen recht simples Beispiel, das aber noch dadurch verfeinert werden könnte, dass alle geladenen Sounds in einem entsprechenden Memory-Manager verwaltet werden. Die Aufgaben einer Engine können aber auch weit komplexer ausfallen, wie es beispielsweise bei einer 3D-Engine der Fall ist. Hier ist die Aufgabe, dreidimensionale Daten in die 3D-Pipeline einer Grafikkarte zu schieben, schon lange in den Hintergrund gerückt. Vielmehr dient eine 3D-Engine dazu, Daten zu organisieren und zu berechnen. Wie wir später noch sehen werden, ist insbesondere das Scene-Management ein großer Bestandteil dieser Aufgabe. Idealerweise sendet der Benutzer sämtliche dreidimensionalen Daten einer darzustellenden Szene während der Initialisierung an die 3D-Engine. Deren Job ist es nun unter anderem, diese Daten zu strukturieren und zu organisieren. Die 3D-Pipeline, durch die unter anderem über die oben genannten APIs zugegriffen werden kann, wurde zwar mit der Weiterentwicklung der Grafikkarten immer besser und schneller, die entstehenden 3D-Programme wurden in der Regel jedoch kaum schneller. Warum? Nun, je mehr eine Grafikkarte leisten kann, desto mehr und desto realistischere Grafiken und Effekte möchten die Spieleentwickler in ihren Programmen darstellen.

*3D-Pipeline
versus 3D-Engine*

Doch auch die neueste und modernste Grafikkarte lässt sich relativ schnell sauber in die Knie zwingen, wenn man weiß wie. Oder besser gesagt: Wenn man nicht weiß, wie man es besser macht. Und dabei hilft einem auch die beste 3D-Pipeline einer DirectGraphics- oder OpenGL-API nichts. Die Aufgabe der 3D-Engine, also des Teils, den wir vor die 3D-Pipeline der API setzen, ist es hier, die ihr anvertrauten Daten möglichst schnell und genau so zu berechnen, dass auch nur diejenigen Daten in die 3D-Pipeline eingefüttert werden, die zurzeit benötigt werden. Ein kleines Beispiel: Wenn ein Spieler in einem Level steht, so ist mindestens all das nicht sichtbar, was sich hinter seinem Rücken befindet. Diese Daten müssen daher nicht durch die sehr enge 3D-Pipeline geprügelt werden.

Doch eine 3D-Engine hat natürlich noch mehr Aufgaben zu erfüllen als nur die Organisation und Anordnung der Daten. Durch die Einführung der so genannten Vertex-Shader und Pixel-Shader haben wir Grafik-Programmierer eine vollkommen neue Möglichkeit erhalten, über die 3D-Engine die 3D-Pipeline sehr flexibel selbst zu programmieren.

Resümee

Was genau man nun dem Funktionsumfang der 3D-Engine zuteilt oder in andere Mini-Engines abschiebt, das bleibt letzten Endes jedem selbst überlassen. Daher kann es auch keine einheitliche Definition einer 3D-Engine geben. Dennoch denke ich, insbesondere der Unterschied zwischen der 3D-Engine und der 3D-Pipeline ist hier klar geworden. Die 3D-Pipeline werden wir uns dann noch einmal intensiv anschauen, wenn wir in einem späteren Kapitel über die Vertex- und Pixel-Shader reden. Die 3D-Engine wird uns an vielen Stellen immer mal wieder über den Weg laufen, auch wenn man sie nicht so genau fassen kann. Im Verlauf dieses Buches werden wir die *ZFXEngine* entwickeln, die eine Engine ist, die wiederum verschiedene kleine Engines in sich

vereint – beispielsweise eine Render-Engine, eine Input-Engine und eine Sound-Engine. Abschließend sei noch ein wichtiges Kriterium für Engines erwähnt, an dem bereits viele so genannte Engines scheitern werden: *Eine Engine muss jederzeit von einem konkreten Spiel abgekoppelt und an ein anderes Spiel oder Software-Projekt angekoppelt werden können, ohne dass der Code der Engine dabei geändert werden müsste.*

1.2 Eine Art von Magie

Hm ... ein Kapitel gleich mit so einer Moralpredigt über die Nomenklatur von Engines und Pseudo-Engines zu beginnen ist anscheinend kein guter Anfang. Dennoch bin ich der Meinung, dass ein paar klärende Worte vorweg nicht schaden, und schließlich sollte jeder wissen, worüber wir in diesem Buch reden wollen.

Trotz all der abschreckenden Worte über die leichte Bezwingbarkeit einer modernen Grafikkarte kann man aber eine Sache nicht leugnen: Wenn man es sich nicht zur Aufgabe gemacht hat, die Grafikkarte in die Knie zu zwingen, sondern vielmehr ihre Sprache spricht und mit ihr zusammenarbeitet, dann kann man heutzutage Dinge auf den Bildschirm eines Computers zaubern, von denen vor gar nicht mal fünf, sechs Jahren noch niemand zu träumen gewagt hat. Und das alles zum Preis einer aktuell durchschnittlich guten Grafikkarte von um die 90 € die kaum größer als ein Taschenbuch ist. Vor noch nicht allzu langer Zeit kostete die entsprechende Hardware noch das Zehntausendfache und füllte ganze Räume aus.

Nostalgie und Neuzeit

An dieser Stelle ver falle ich gerne ins Schwärmen über das, was heutzutage möglich ist. Nur wer die Zeiten eines *Wing Commander 1* und *Doom* miterlebt hat, wo sich pixelige, kleine Sprites bei einer Bildschirmauflösung von 320x200 und 256 Farben in einer gefakten 3D-Welt über den Bildschirm quälten und dabei staunende Gesichter hervorriefen, kann dies nachvollziehen. Damals sorgten die Spiele nicht nur für Umsatz in ihrem eigenen Markt, sondern bescherten auch den Hardware-Herstellern steigende Umsätze. Nicht wenige Spieler kauften sich einzig und allein aus dem Grund neue Hardware, damit sie ein neues Spiel spielen konnten. Ob dies nun *Wing Commander III* oder *Quake II* hieß, spielt keine Rolle. Fakt ist, dass die Computerspiele viele Menschen derart verzauberten, dass sie mehrere hundert Mark in neue Prozessoren und zusätzlichen Speicher steckten. Dieser Boom im Markt ermöglichte es im Gegenzug, dass dort immer neuere und bessere Hardware in immer kürzeren Lebenszyklen entstand. Es entwickelte sich ein Trend hin zu besseren Prozessoren auf der Grafikkarte selbst. Und heutzutage ist es ohne weiteres möglich, einfach Tausende von Polygonen in die 3D-Pipeline einer Grafikkarte zu pumpen und diese in Echtzeit am Bildschirm mit multiplen Texturen anzeigen zu lassen.

*Früher war alles
besser ... ?*

Die Neuzeit

#Leider resultiert daraus aber auch das Verhalten von Neuligen, dies einfach so zu tun. Spätestens dann, wenn man einen ganzen Level und alle darin enthaltenen Objekte mit mehreren Texture-Stages, Echtzeit-Beleuchtung und allen anderen grafischen Effekten komplett zur Grafikkarte schiebt, weil »*die ja so schnell ist*«, wird sich diese mit einer kleinen Stichflamme verabschieden. Auch heute ist also durchaus das Mitdenken im Bereich des Entwurfs der Engine gefragt, auch wenn sich durch die neue Hardware viele Dinge um 180 Grad gedreht haben. Beispielsweise wird man heute eher Berechnungen einsparen und die moderne Grafikkarte ihren Job tun lassen, während man früher noch viel rechnete um die langsame Grafikkarte zu schonen. Nun gilt es jedoch, andere Fallstricke als früher zu beachten. Damals quälten sich die Programmierer damit, effiziente Algorithmen zu entwickeln, die die bei Rotationen notwendigen trigonometrischen Berechnungen beschleunigten oder die das Setzen eines Pixels auf der Grafikkarte möglichst ohne Overhead und Overdraw ermöglichten. Jeder Cycle der CPU war quasi bares Geld wert und musste dreimal überdacht werden, bevor man ihn ausgab. Heutzutage tendiert man dazu, CPU Cycles wie Rubel unter die Menge zu werfen, ohne merklichen Impact auf die Geschwindigkeit zu spüren. Zumindest so lange nicht, bis man eine Testumgebung mit einer entsprechend geeigneten Datenmenge zur Verfügung hat und nicht nur ein Modell mit viertausend Polygonen am Bildschirm rendert. Aber das ist nicht das Hauptproblem.

Die moderne Technik im Bereich der Computergrafik ist heutzutage so weit, dass man eine Art Arbeitsteilung im Computer eingeführt hat. Der Hauptprozessor (die CPU) ist dazu da, die Algorithmen zu berechnen, die der Programmierer entwirft. Daran hat sich nichts geändert. Nun ist aber der Prozessor der Grafikkarte selbst so weit entwickelt, dass er einen großen Batzen der Arbeit mit erledigen kann, zumindest was die Berechnung von Grafik und 3D-Modellen angeht. Die Herausforderung liegt heutzutage nicht mehr darin, möglichst viele Cycles der CPU zu sparen, sondern darin, die Arbeit so effizient wie möglich zwischen den beiden Prozessoren zu verteilen. Über die so genannten *Shader* ist es inzwischen auch endlich möglich, Programme zu schreiben, die direkt vom Prozessor der Grafikkarte bearbeitet werden. Der Trick dabei ist nun, dass man so wenig Traffic, also Datenverkehr, wie möglich zwischen den beiden Prozessoren hat. In den späteren Kapiteln werden wir das noch genauer erörtern. An dieser Stelle ist es aber wichtig einzusehen, dass mit der Entwicklung neuer Technik auch neue Technologien und Ansätze von den Programmierern gefordert werden.



Ich erinnere mich noch gut daran, wie ich die ersten Screenshots des im Verlauf dieses Buch entwickelten Spiels Pandoras Legacy auf der ZFX-Homepage¹ präsentierte und die ersten Facts preisgab. Allein dafür, dass ich keinen BSP mit PVS² verwendete, wurde ich fast gelyncht und musste lange argumentieren, um die Entscheidung zu rechtfertigen.

1 <http://www.zfx.info>

2 Binary Space Partitioning mit Potential Visibility Set

Ganz platt ausgedrückt würde man sagen: »Du musst deinen Feind kennen, um ihn zu besiegen.« Aber man kann es noch besser formulieren: »Wenn du deinen Feind nicht besiegen kannst, dann verbünde dich mit ihm.« Unser Feind ist die Hardware im Computer. Diese ist auf alle Fälle immer zu langsam für das, was ein Spieleprogrammierer gern alles umsetzen möchte. Betrachtet man die Hardware jedoch als seinen Freund und versucht man, sie näher kennen zu lernen und ihre internen Abläufe zu verstehen, dann wird man eher zum Ziel kommen als durch Rambo-Strategien. Das heißt: Wer Spiele programmieren will, der sollte sich auch in den einfachsten technischen Abläufen der Hardware gut auskennen. Wir werden im Verlauf dieses Buches natürlich die aktuellen Entwicklungen der Hardware anschauen und daraus einen entsprechenden Schluss bei der Auswahl an Algorithmen und Strategien für unsere Engine ziehen.

*Die Hardware –
dein bester
Kumpel!*

Der Zauberer von Oz ...

Wenn man sich mit einem offenen Geist an die Programmierung von 3D-Grafik macht, dann wird man unweigerlich diese Magie verspüren, die man heutzutage erzeugen kann. Es gibt Programmierer, die sind Programmierer. Und es gibt Programmierer, die sind Künstler oder auch Zauberer. Durch ihren Zauber erzeugen sie die Magie eines 3D-Spiels, in dem raffinierte Licht- und Schatteneffekte von einer atmosphärischen Soundkulisse dezent untermalt werden. Man sollte diesen Aspekt der Entwicklung eines Computerspiels nicht als lächerlich oder unwichtig abtun. Ganz im Gegenteil: Genau dieser Aspekt ist der Kern unseres Schaffens. Ein fertiges Computerspiel ist nicht einfach ein Produkt, das sich in Zahlen und Codes ausdrücken lässt, wenigstens nicht für den Programmierer und die potenziellen Spieler. Marketing- und Produkt-Manager des Publishers (der das Spiel nachher vermarktet) und des Distributionskanals mögen das (aus ihrer Perspektive zu Recht) anders sehen. Aber für den Entwickler sollte sein fertiges Spiel ein Kunstwerk darstellen. Ähnlich wie ein Maler können die Grafiker des Teams die Texturen für die Objekte im Spiel gestalten; ähnlich wie ein Bildhauer können die Modeller des Teams die Modelle für das Spiel erschaffen; ähnlich wie ein Musiker können die Sounddesigner des Teams die akustische Kulisse für das Spiel komponieren. Aber es gibt auch einen großen Unterschied zwischen einem herkömmlichen Kunstwerk und einem Computerspiel. Das Computerspiel ist eine *interaktive* Form der Kunst, wie es Richard Rouse III in seinem Buch über Game Design ausgedrückt hat.³

*Spiele-Entwick-
lung ist Kunst!*

Das fertige Werk umfasst dabei viele einzelne Bereiche, die aber erst zusammengekommen ein großes Ganzes ergeben: das fertige Kunstwerk. Die Aufgabe eines *Lead Designers* ist es beispielsweise, eben genau darauf zu achten, dass die einzelnen Teile sich zu einem Gesamtkunstwerk zusammenfügen lassen, ohne Stilbrüche zu erzeugen. Der Erfolg eines Spiels steht und

*Integrität des
Gesamtwerks und
Gameplay*

³ Game Design – Theory & Practice, Wordware, 2001

fällt auch gerade mit dem Gesamtdesign. Der unüberwindbare Unterschied zwischen dem hundersten Ego-Shooter und einem sehnüchtlig erwarteten *Doom III* ist ... Kunst! Und zwar die Kunst, all die einzelnen Elemente des gesamten Spiels zu einem einzigartigen Entertainment-Erlebnis zu vereinen.

Es ist Magie!

Ein Computerspiel ist aber noch viel mehr als das. Ähnlich wie ein beachtetes Kunstwerk übt es einen Zauber, eine magische Anziehungskraft auf die Spieler aus. Während »normale« Anwendungs-Software in 99,9% aller Fälle dazu entwickelt wird, als Werkzeug eine Lösung für bestimmte Problemstellungen im Arbeitsalltag zu bieten, haben Spiele eine ganz andere Zielsetzung. Wir wollen an dieser Stelle nicht in die sozialwissenschaftliche Analyse der Ursachen für die Faszination eines Computerspiels abdriften. Aber das deutlichste Unterscheidungsmerkmal eines Computerspiels von anderer Software ist unbestritten, dass ein Computerspiel der virtuelle Gegenpart eines Freizeitparks ist. Die Menschen gehen in einen Freizeitpark, um dort unterhalten zu werden. Sei dies nun bei einem *Magic Mountain-Ride* oder beim Snack in einem Themen-Restaurant im Stil der Hotels in Las Vegas. Die Menschen möchten dort einfach abschalten und sich verzaubern lassen. Genau denselben Zweck soll ein Computerspiel erfüllen. Es soll den Spieler für eine Zeit lang aus seinem Alltag reißen und ihn zum Geheimagenten machen oder zum Piloten eines Jets oder zu einem Rennfahrer. Er soll gefährliche Aufträge unter Einsatz seines virtuellen Lebens erfüllen und dabei die Welt oder die Jungfräulichkeit von ein paar Babes⁴ retten.

Blasphemie

In der von ihm erschaffenen Welt ist der Programmierer eine Art Gott, definiert als Bezeichnung für ein Wesen, das die Naturgesetze (in der virtuellen Welt) aufstellt. Soll ein Charakter in der virtuellen Welt ebenfalls einen Teil Magie beherrschen, oder soll die Schwerkraft von oben und nicht von unten kommen? Ob einen dunklen Dungeon voller zauberhafter Wesen, einen dunklen Hinterhof voller Yakuzzas in Tokio oder ein düsteres Raumschiff voller schleimiger Aliens – was auch immer man für eine Welt erschaffen möchte, heutzutage ist dies mit einem akzeptablen Grad an Realismus möglich.

... und die Verantwortung eines Gottes

*Computerspiele
und ihre Folgen*

Doch auch wenn sich das Schaffen eines Computerspiele-Entwicklers als Quasi-Gott in einer virtuellen Welt abspielt, hat es doch auch Auswirkungen auf die reale Welt. Aufgrund aktueller Entwicklungen kommt man heutzutage nicht umhin, sich mit gewisser Kritik an Computerspielen auseinander zu setzen. Ein Großteil der Computerspiele befasst sich in der Regel damit, vergangene Kriege nachzuspielen, fiktive Kriege in der virtuellen Welt zu erschaffen oder irgendeine Art von Waffe gegen irgendeine Art von Gegner einzusetzen. Ob dies nun ein Schwert ist oder eine Laserkanone, macht vom Prinzip her keinen Unterschied.

4 Man erinnere sich nur an Duke Nukem.

Computerspiele sind deshalb in die Kritik geraten, weil sie dem Spieler in der Regel die Möglichkeit geben, durch aktives Handeln Gewalt gegen andere auszuüben. Zwar handelt es sich dabei um virtuelle Gewalt gegen virtuelle andere, aber der Kritikpunkt bleibt bestehen. Genau hier ist nämlich der Unterschied zu der passiv aus Filmen konsumierten Gewalt. Es gibt nun Dutzende von verschiedenen Meinungen darüber, inwieweit ein Computerspiel die realen Handlungen des Spielers beeinflusst und ihn beispielsweise zu schrecklichen Amokläufen wie in den USA und nun auch in Deutschland motiviert. Für jede Meinung auf der Welt lassen sich mindestens ebenso viele Gutachten dafür wie auch dagegen erbringen, und dieses Buch ist nicht der richtige Platz, um dieses Thema zu diskutieren.

Fest steht aber auf alle Fälle, dass die Programmierer und Designer eines Computerspiels einen Einfluss auf die Spieler haben können. Wenn man sich also an die Entwicklung eines Computerspiels macht, dann sollte man die sich daraus ergebende Verantwortung immer im Hinterkopf halten. Ob der Realismus eines Spiels durch Effekte wie physikalisch korrekt berechnete Blutropfen aus einer Schusswunde merklich verbessert wird, so dass dieses Feature die Qualität des Spiels spürbar steigert, sollte man sich folglich gut überlegen. Ebenso ist es fraglich, ob die US-Armee der geeignete Publisher für ein (kostenloses) Kriegsspiel auf technisch hohem Niveau ist. Ob hier wirklich und ausschließlich der Spielspaß der Jugend ohne Hintergedanken befriedigt wird, sei dahingestellt.

Wie viel Blut muss sein?

Man sollte sich also immer darüber im Klaren sein, dass die Arbeit als Spiele-Entwickler eine Menge Verantwortung mit sich bringt: zum einen die Verantwortung, dem Käufer das Beste zu bieten, was mit jeweils aktueller Hardware machbar ist, zum anderen aber auch die Verantwortung, ein Spiel immer ein Spiel bleiben zu lassen.

1.3 Entstehungsprozess eines Spiels

Nachdem wir nun ein wenig über die Motive und die Motivationen gehört haben, die einen Entwickler bei der Erstellung eines Spiels antreiben oder wenigstens antreiben sollten, müssen wir uns noch ein wenig damit beschäftigen, wie ein Entstehungsprozess abläuft. Die Erschaffung eines Computerspiels ist ein langwieriger Prozess in verschiedenen Kompetenzbereichen, die von Einsteigern in die Branche schnell verwechselt oder gar nicht erst erkannt werden. In den folgenden Absätzen erläutere ich daher kurz ein paar Begriffe, die man als Entwickler von Entertainment-Software kennen sollte. Gleichzeitig werde ich den Inhalt dieses Buches auch entsprechend zu den einzelnen Bereichen abgrenzen.

Schöpfung ist ein Prozess

Wer vorhaben sollte, in das Geschäft Computerspiele-Entwicklung auch beruflich einzusteigen, der sollte die folgenden Absätze genau lesen, um einen ersten Eindruck von den Verflechtungen innerhalb des Entwicklungs-

Berufliche Perspektive

prozesses zu bekommen. Und man sollte sich dabei nichts vormachen. Auch wenn es um Spiele geht, so bewegen wir uns hier in einem ernsthaften Marktsegment, das ebenso hart umkämpft ist wie jedes andere lukrative Marktsegment, egal ob es sich dabei um Anwendungssoftware, Autos, Seife oder Müllbeutel handelt. Von erfolgreichen Computerspielen werden durchschnittlich einige hunderttausend Exemplare weltweit verkauft, sehr erfolgreiche Titel kommen gar über die Millionengrenze, und die absoluten Spitzentitel verkaufen sich mehrere Millionen mal.⁵ Hier sei nur auf John Carmack von *id Software* verwiesen, der bekanntermaßen das Ferrari-Sammeln als Hobby für sich entdeckt hat. Die kommerzielle Entwicklung eines Computerspiels ist also ein Business wie jedes andere. Nur vielleicht mit dem Unterschied, dass die Arbeit an dem Produkt ein wesentlich kreativerer Prozess ist als bei Seifen oder Müllbeuteln.

*Computerspiele-
Entwicklung*

Daher halte ich es an dieser Stelle für sinnvoll und auch interessant, wenn ich ein wenig mehr über den gesamten Prozess der Entwicklung eines Spiels aus kommerzieller Sicht schreibe, quasi vom Beginn der Arbeit an einem Titel bis zu dem Moment, wo er dann im Laden steht und gekauft werden kann. Die folgenden Abschnitte beschäftigen sich mit diversen Einzel-Aspekten der Computerspiele-Entwicklung, allerdings in eher umgekehrter Reihenfolge, als sie eigentlich stattfinden. Das ist deshalb so, weil wir bei der Betrachtung der bekanntesten Dinge auf vertrautem Gelände anfangen und uns dann erst zu den Dingen vortasten, die für Einsteiger im Bereich der Spiele-Entwicklung sicherlich relativ neu sein werden. Allen diesen Aspekten ist gemeinsam, dass sie ein Teil des Gesamtprozesses namens Computerspiele-Entwicklung (engl. Game Development) sind, zum Teil aber fälschlicherweise als eigenständige Gesamtprozesse interpretiert werden.

*Job oder
Berufung?*

Eine andere Frage ist natürlich auch noch, mit welcher Vision man antritt, um Spiele-Entwickler zu werden. Hier gibt es natürlich mehrere Optionen. Die zwei wichtigsten sind wohl die folgenden:

- ➔ Anstellung als Programmierer, Grafiker, ... bei einem Team
- ➔ Gründung eines eigenen Teams

Beide Optionen haben ihre Vor- und Nachteile. Wenn man lediglich eine Anstellung sucht, wird man sich in einem stagnierenden Markt gegen eine große Zahl von Konkurrenten durchsetzen müssen. Gleichsam läuft man natürlich immer Gefahr, wieder auf den Markt freigestellt zu werden, wenn das Geschäft des Entwickler-Teams sich nicht wie erwartet entwickelt und es Personal abbaut. Wenn man sich entscheidet, ein eigenes Entwickler-Team zu gründen, so trägt man logischerweise selbst die gesamte Verantwortung. Man muss einerseits das Projekt an sich voranbringen, andererseits muss man

⁵ Diese Werte beziehen sich auf international erfolgreiche Titel. Produktionen des deutschen Marktes müssen sich mit geringeren Zahlen zufrieden geben.

fähige Leute finden, die sich der eigenen Vision zu einem konkreten Projekt aus eigenem Antrieb heraus hoch motiviert anschließen, und zu guter Letzt muss man noch einen geeigneten Publisher oder eine andere Finanzierungsquelle finden. Man sollte dabei nicht das betriebswirtschaftliche Wissen unterschätzen, was dazu notwendig ist. Nur ein guter Programmierer zu sein hilft da nicht weiter. Aber gehen wir der Reihe nach vor.

Spieleprogrammierung (Game-Programming)

Eines der größten Missverständnisse bei Computerspielen ist vielleicht die Annahme, dass die Programmierung des Spiels den wichtigsten und größten Teil der Arbeit ausmache. Hier hat sicherlich jeder das Bild des langhaarigen Freaks vor Augen, der mit einer glühenden Zigarette im Mundwinkel und einer Tasse schwarzen Kaffees um drei Uhr morgens mit blutunterlaufenen Augen vor dem Monitor sitzt und Algorithmen in die Tastatur hackt.

Der Freak

Zugegeben, nicht der langhaarige Freak ist dabei das Missverständnis, sondern die Wichtigkeit, die der Programmierung zugemessen wird. Naturgemäß nimmt die Programmierung natürlich einen großen Teil der Arbeit in Anspruch, denn schließlich entsteht das fertige Produkt größtenteils aus dem programmierten Code. Doch was genau hacken die Freaks da in ihre Tastaturen? Das, was sie sich spontan ausdenken? Oder das, was sie denken was cool rüberkommt? Nein, mit Sicherheit nicht. Programmiert wird dort nämlich genau das, was das Design-Dokument (dazu kommen wir gleich) vorschreibt. Die Programmierung eines Computerspiels ist so betrachtet eigentlich der langweiligste Aspekt bei so einem Projekt. Schließlich ist es letztendlich nicht viel mehr, als das sture Herunterprogrammieren einer vorgegebenen Aufgabenliste.

Der Code

Natürlich ist es nicht ganz so schrecklich, wie das jetzt vielleicht klingt. Ich wollte es nur bewusst so hart formulieren, damit niemand auf die Idee kommt, man entwickle ein Spiel, indem man sich an die Tastatur setzt und losprogrammiert. Natürlich stellt die technische Umsetzung der geforderten Features einen Programmierer immer wieder vor Herausforderungen bei der Implementierung. Auch an dieser Stelle des Entwicklungsprozesses sind Kreativität und Feedback für das gesamte Projekt notwendig und überlebenswichtig. Darüber hinaus sollte man aber immer im Hinterkopf behalten, dass die Programmierung nur ein Teilaspekt der Entwicklung eines Computerspiels ist. In diesem Buch wird es vorwiegend um die Programmierung gehen, auch wenn ich versuche, hier und da den Fokus etwas zu erweitern. Daraus kann man aber auch ableiten, dass die Fähigkeit, gut zu programmieren, zum Grundhandwerkszeug eines Computerspiele-Entwicklers gehört – oder wie die Mathematiker sagen würden: Es ist eine notwendige, aber *nicht* hinreichende Bedingung. Die Programmierung sollte man im Schlaf beherrschen, damit man den Kopf für die Dinge frei hat, die ein Computerspiel von einer anderen Software-Applikation unterscheiden.

Die Umsetzung

Das Buch

Mit diesem Buch haben wir uns das ehrgeizige Ziel gesetzt, die entsprechenden Kenntnisse im Bereich der Spieleprogrammierung vermitteln zu wollen. Eine notwendige Voraussetzung ist beispielsweise, dass man bereits des Programmierens mächtig ist. Und damit meine ich insbesondere nicht die Kenntnisse einer bestimmten Programmiersprache. Eine solche kann man sich sehr schnell aneignen, wenn man programmieren kann. Das Programmieren an sich ist eine Fähigkeit, die nicht an eine Sprache wie *Visual Basic*, *Java* oder *C++* gebunden ist. Wer programmieren kann, der kann das in jeder Sprache, zu der man ihm eine Referenz gibt. Die beste Programmiersprache ist immer noch Pseudo-Code. Aber natürlich ist es für das Erlernen der Spieleprogrammierung (mit diesem Buch) hilfreich, wenn man gute Kenntnisse der Sprache *C++* und der *WinAPI* hat. Die Spieleprogrammierung dreht sich dann eher darum, wie man Multimedia-APIs sinnvoll verwendet und wie man seinen Code effizient aufbaut.

*Lead Programmer/
Technical Director*

Für jeden Subkomplex in einem größeren Team (nicht selten werden Computerspiele in Teams von zehn bis zu dreißig Personen entwickelt) gibt es eine Führungsposition für das Unterteam, beispielsweise für die Programmierung, für die Grafik und natürlich für das Design. Die Aufgabe des *Lead Programmers* oder auch *Technical Directors* ist dabei natürlich die Koordination aller Programmierer des Teams, über die er entsprechend Weisungsbefugnis haben muss. Das bedeutet, dass er beispielsweise in implementierungstechnischen Fragen im Streitfall das letzte Wort hat und die entsprechenden Subaufgaben aus der Gesamtaufgabe für die Programmierer ableitet sowie den Programmierern zuweist.

Spiele-Design (Game-Design)

Design-Dokument

Das Spiele-Design (engl. Game Design) ist nun quasi alles an einem Computerspiel, was nicht mit der Programmierung zu tun hat. Bei der Entwicklung eines Computerspiels ist einer der ersten Schritte, dass man ein so genanntes *Design-Dokument* erstellt. Dieses Dokument ist das Herz und die Seele des gesamten Projekts und stellt sozusagen das fertige Computerspiel in schriftlicher Form dar. Es beinhaltet alle Aspekte des Spiels und auch erste Skizzen und Grafiken der vorkommenden Locations und Charaktere. Zusätzlich findet man dort die Storyline, Cut Scenes, eine Beschreibung der Benutzer-Interfaces⁶, die möglichen Aktionen, die der Spieler ausführen kann, und so weiter. Um es auf einen Punkt zu bringen: Würde man dieses Design-Dokument einem vollkommen projektfremden Team von Programmierern, Grafikern und Modellern auf den Tisch legen, dann müssten diese in der Lage sein das Spiel, genau so umzusetzen, wie es das ursprüngliche Team kann.⁷

6 Letztens wurde ich doch glatt von einem Professor belehrt, dass man aus Respekt vor den Menschen von einem »Benutzungs-Interface« spricht. Aber ich finde wer die Zeit hat, sich über solche Dinge Gedanken zu machen, der hat einfach nichts besseres zu tun.

7 Das ist natürlich nur bedingt möglich, da so ein Projekt in der Regel von der Kreativität und gemeinsamen Vision der konkreten Teammitglieder getragen wird.

Die Erstellung eines solchen Design-Dokuments ist eine intensive Arbeit, die bis zu einem Jahr Zeit in Anspruch nehmen kann. In dieser Zeit wird allerdings nicht nur dieses Dokument erstellt, sondern auch parallel Marktforschung betrieben und ein Publisher gesucht. Dennoch ist die Erstellung des Designs ein Schritt, der vor der eigentlichen Programmierung abgeschlossen sein sollte – spätere Änderungen während des Prozesses mal außen vor gelassen.

Damit ist auch klar, was die Aufgabe des Lead Designers ist. Diesem obliegt die Verantwortung, in Zusammenarbeit mit dem gesamten Team das Game-Design festzulegen. Dazu gehört beispielsweise auch die Atmosphäre des Spiels, die durch die akustischen Effekte und die Texturen und Grafiken vermittelt wird. Der Lead Designer muss dabei nicht notwendigerweise Ahnung vom Programmieren haben. Seine Hauptarbeit besteht darin, dafür Sorge zu tragen, dass alle optischen und akustischen Elemente des Projekts konsistent zur Atmosphäre des Projekts passen. Auf Gamasutra findet sich ein interessanter Post-Mortem-Artikel von Warren Spector über die Entwicklung des Spiels *Deus Ex*.⁸

Lead Designer

Da das Spiel über die Grenzen der konventionellen Genres weit hinausgehen sollte und da es zwei kompetente Personen für den Posten des Lead Designers gab, wurden kurzerhand zwei Design-Teams gegründet, jedes mit seinem eigenen Lead Designer. Um es gleich vorweg zu nehmen, dieser Teil des Post Mortem steht in der Rubrik *What went wrong*. Die unterschiedlichen Ansichten und Philosophien zweier Lead Designer führten zu Spannungen in den Teams und nicht zu den erhofften Synergie-Effekten. Bezeichnenderweise wurden die Teams auch *Team 1* und *Team A* genannt, da kein Team gern die Nummer 2 oder B gewesen wäre. Letzten Endes musste das Projekt dann nach vielen Monaten auf ein Team und einen einzigen verantwortlichen Lead Designer mit endgültiger Entscheidungskompetenz beim Design umgestellt werden. Man kann sich, glaube ich, gut vorstellen, dass dies sowohl zu technischen wie auch temporalen, aber insbesondere auch sozialen Problemen führen kann.

Als Lead Designer kann man diejenige Person im Projektteam bezeichnen, die bei Streitfragen und Differenzen über Design und Konzept des Spiels die endgültige Entscheidungskompetenz hat. Dabei muss es sich nicht zwangsläufig um eine entsprechend bezeichnete Stelle handeln, die eigens eingerichtet wurde.



Aus dieser Geschichte kann man das Resümee ziehen, dass eine Art diktatorische Demokratie beim Game-Design – und damit für das gesamte Projektteam – die sinnvollste Form der Teamleitung ist. An dem Schöpfungsprozess des Designs sollten natürlich alle Teammitglieder beteiligt werden, um das

*Diktatorische
Demokratie*

⁸ http://www.gamasutra.com/features/20001206/spector_01.htm

kreative Potenzial des Teams voll nutzen zu können.⁹ Sollte es aber zu Differenzen kommen, so sollte es immer genau eine Person geben, die die endgültige Entscheidung über eine Streitfrage trifft. Diese Kompetenz muss natürlich im Team vorher bekannt sein.

Umsetzung des Designs

Mit dem Begriff GameDesign bezeichnet man also den kompletten Entwurf und die detaillierte Planung des Spiels. Das »fertige« Design-Dokument, das, wie bereits erwähnt, in Teamarbeit erstellt wird, wird dann an die einzelnen Teammitglieder zur Bearbeitung weitergereicht. Die Programmierer implementieren die Engine (falls diese nicht extern über eine Lizenz erworben wird) und den Game-Code, die Grafiker (engl. Artists) erstellen die Grafiken und Texturen für das Spiel, die Modeller und Level-Designer erzeugen die Modelle und die Level für das Spiel usw. Dabei sollte man immer beachten, dass das Game-Design ein dynamischer Prozess ist, der von dem Feedback während der gesamten Entwicklungszeit lebt. Ein Design-Dokument sollte nie als fixe Konstante angesehen werden, sondern als Stand eines Projekts zu einem konkreten Zeitpunkt. Während der gesamten Entwicklungsdauer werden immer wieder Änderungen und Ergänzungen am Design notwendig sein. Die Vision des Projekts sollte dadurch aber nicht mehr verändert werden, schließlich wurde das Projekt als Vision zuvor (unter anderem auch basierend auf dem Design-Dokument) an einen Publisher »verkauft« (dazu gleich mehr). Über das Game-Design an sich gäbe es noch weitaus mehr zu sagen, daher verweise ich an dieser Stelle auf das sehr gute Buch von Richard Rouse III.¹⁰

Spiel-Entwurf (Game-Proposal)

Vorsicht Falle

Der komplette Entwurf für ein Computerspiel sollte nicht mit dem Design-Dokument verwechselt werden. Er ist viel umfangreicher und beschreibt zusätzlich zu dem Inhalt des Design-Dokuments weitere Details bezüglich des Projekts. Nun mag man sich fragen welche Details das Design-Dokument noch nicht beschreibt. Die Antwort auf diese Frage ist simpel zu beantworten. Während sich das Design-Dokument zwar sehr intensiv mit dem Spiel an sich beschäftigt, so sagt es doch nichts über die »Umwelt« des Spiels aus. Einer der wichtigsten Aspekte bei der Evaluation einer neuen Produktidee vor dem Beginn der Produktion ist aber eine genaue Analyse der Umwelt, in der sich das Produkt später bewegen soll.

Der Spiel-Entwurf ist das Dokument, mit dem sich ein Entwickler-Team bei einem Publisher bewirbt, um ihn für die Veröffentlichung und damit auch die Finanzierung des Titels zu gewinnen. Dementsprechend wichtig ist die gewissenhafte Erstellung des Dokuments, das später auch Bestandteil des Vertrags zwischen dem Team und dem Publisher wird.

⁹ Dabei ist vorausgesetzt, dass alle Mitglieder des Teams aus Überzeugung und im Glauben an die Vision des Projekts arbeiten und dass diese Vision bei allen Teammitgliedern grob übereinstimmt.

¹⁰ Game Design – Theory & Practice; Wordware Publishing Inc., 2001

Ebenso wie für das Design-Dokument gibt es auch für den Entwurf keine Vorlage, wie man das Dokument gestalten sollte und welche Form es haben soll. Ebenso wenig kann man sich auf die genaue Länge eines solchen Dokuments festlegen. Umfangreiche Projekte brauchen mehrere hundert Seiten Text, während kleinere oder simplere Projekte mit weniger auskommen. Dennoch ist es natürlich für Neueinsteiger wünschenswert zu sehen, wie so ein Dokument aussehen *könnte*. Allen, die sich nicht nur mit der Programmierung, sondern auch mit der Entwicklung eines Computerspiels beschäftigen wollen, kann ich daher das Buch von Luke Ahearn empfehlen.¹¹ Dort findet man eine sehr gute Einführung in entsprechende Themen wie beispielsweise Game-Proposal und Design-Dokument sowie grundlegende betriebswirtschaftliche Techniken und Marketing-Basiswissen.

In seinem Buch nennt Luke Ahearn die folgenden Komponenten, die Bestandteil eines Game-Proposal sein sollten:

- ➔ Game Treatment
- ➔ Competitive Analysis
- ➔ Design Document
- ➔ Team Introduction
- ➔ Budget
- ➔ Schedule

Als Erstes fällt hier natürlich auf, dass das Design-Dokument ein Bestandteil des gesamten Proposals ist. Wie bereits erwähnt, ist das Proposal ja sozusagen der gesamte Projektplan, und das Design-Dokument als komplette Beschreibung des zu entwickelnden Spiels ist damit natürlich logischerweise ein Teil davon. Schauen wir uns nun also kurz an, was mit den anderen Teilen gemeint ist.

Game Treatment

Das *Game Treatment* (dt. Behandlung des Spiels) ist sozusagen die *Executive Summary* (dt. Zusammenfassung für den Manager) des gesamten Dokuments. Ein verantwortlicher Manager (nicht nur im Bereich der Spiele-Entwicklung) hat in der Regel einen sehr langen und arbeitsreichen Tag und hetzt von Termin zu Termin. Von unten kommen aus der Hierarchie scheinbar Tausende von Entscheidungsanfragen, Wünschen und abzuarbeitenden Dokumenten. Der Manager muss innerhalb kürzester Zeit die ankommenden Arbeiten nach Priorität einordnen: sehr Wichtiges sofort, weniger Wichtiges später und Unwichtiges in den Mülleimer. Damit ein Manager eine solche Entscheidung bei umfangreichen Dokumenten treffen kann, beinhalten diese eine Executive Summary die die wichtigsten Aussagen und

¹¹ Designing 3D Games That Sell; Charles River Media Inc, 2001

Inhalte des Dokuments zusammenfasst. Basierend auf dieser Summary entscheidet der Manager, welche Priorität er diesem Dokument beimisst. Das Game Treatment sollte also auf einer oder zwei Seiten in aller Kürze das geplante Spiel und den weiteren Inhalte des Proposals zusammenfassen.

Competitive Analysis

Die Competitive Analysis beinhaltet die Marktforschungsaktivitäten, die das Team bisher unternommen hat. Hier ist es für den Publisher insbesondere wichtig zu erkennen, inwieweit sich das geplante Spiel von der Konkurrenz unterscheiden wird (Differenzierungsmerkmale). Sind nicht genug Unterschiede zu vergleichbaren Spielen gegeben, so wird es dem Publisher sehr schwer fallen, dieses Spiel im Markt zu platzieren und so zu bewerben, dass es überhaupt Interesse bei den potenziellen Käufern weckt. Ein weiterer wichtiger Punkt ist aber auch, ob es im Markt überhaupt Platz für ein solches Spiel gibt. Ein guter Startpunkt sind dabei die aktuellen Verkaufscharts der letzten Monate. Aus diesen lässt sich beispielsweise ein Rückschluss darauf ziehen, welche Genres bei den Spielern zurzeit sehr beliebt sind. Ein gutes Computerspiel sollte innovativ sein und sich in vielen Details von anderen unterscheiden. Aber ein hoch technisierter Sidescroller unterscheidet sich dann eben doch bei der aktuellen Nachfrage *zu sehr* von den Mitbewerbern im Markt, um ein profitables Verkaufsvolumen zu prognostizieren, wenn beispielsweise Ego-Shooter, Roleplaying Games (RGP) und Handelsimulationen die Charts anführen. Der Publisher möchte des Weiteren auch wissen, welche vergleichbaren Titel kürzlich erschienen sind (oder demnächst erscheinen) und wie deren Marktdurchdringung aussah. Die Competitive Analysis muss also letzten Endes nicht viel mehr darstellen, als die fundiert begründete Evaluation der Chancen, die das Spiel, das ihr entwickeln wollt, im Markt haben wird.

Team Introduction

Der Name *Team Introduction* spricht ja eigentlich für sich selbst. Hier stellt sich das Team kurz vor und verweist auf seine bisherige Erfahrung im Bereich der Spiele-Entwicklung sowie auf vorhandene Referenzen und Portfolios. Schließlich interessiert den Publisher auch, mit wem er es zu tun bekommt, falls er sich auf das Projekt einlässt. Falls ein Team hier noch wenig vorzuweisen haben sollte, empfiehlt sich auch ein Verweis auf die bisherige Ausbildung und den Background und die Erfahrung im Programmieren sowie – was nicht unterschätzt werden sollte, auch wenn es zunächst ein Schmunzeln hervorrufen wird – die Erfahrung im Spielen von Computerspielen. Ein guter Programmierer aus dem Bereich der Applikationsentwicklung, der Computerspiele nur über die Schulter seines Sohnes mal kurz gesehen hat, ist wohl kaum ein geeigneter Kandidat für die Entwicklung eines Computerspiels. Langfristiges eigenes Interesse an Computerspielen bringt überhaupt erst die nötige Erfahrung, um zu wissen, was zu einem guten Spiel gehört und was nicht. Die Mitglieder eines Teams müssen, wie

bereits oben erwähnt, nicht unbedingt alle Programmierer oder Modeller sein. Aber begeisterte Computerspieler sollten sie auf alle Fälle sein.

Budget

Bei einer Veranstaltung des Consulting-Unternehmens *Accenture* (damals noch *Anderson Consulting*) am Institut für Unternehmensführung der TU Braunschweig mussten wir als Studenten für einen fiktiven Kunden eine Beratung bezüglich eines fiktiven Projekts durchführen. Nachdem eine Gruppe ihr Projekt dem fiktiven Auftraggeber (einem *Accenture*-Berater) vorgestellt hatte, so reagierte dieser zunächst nur mit einer Frage auf die Präsentation: »Was wird es kosten?« Über diesen Punkt hatte sich keine Gruppe Gedanken gemacht, aber eben genau dieser Punkt interessiert einen verantwortlichen Manager als Erstes. Ein Entwicklerteam mag Feuer und Flamme für das geplante Projekt sein und viele kreative Visionen und Ideen bezüglich des Spiels haben. Für den Publisher ist aber primär erst mal nur wichtig: Was wird es kosten und wie viel wird es einspielen?

Die Manager-Frage!

Im *Budget*-Teil des Proposals erwartet der Publisher daher einen kompletten Finanzierungsplan bzw. eine Aufstellung aller anfallenden Kosten innerhalb des Entwicklungszeitraums – eben so, wie man es von einem Business-Plan her kennen sollte. Man erkennt bereits an dieser Stelle, dass man zu der Entwicklung eines Computerspiels viel mehr braucht als nur Kenntnisse einer Programmiersprache und viel Motivation. Vielmehr braucht man insbesondere auch Kenntnisse im Bereich Betriebswirtschaft, Marketing und Projektmanagement. Ein Budget enthält beispielsweise ganz banale Dinge wie etwa die Lohnkosten für alle beteiligten Teammitglieder über den geschätzten Entwicklungszeitraum. Auch Einmal-Kosten wie beispielsweise die Anschaffung von Lizenzen für Engines oder Software-Tools sind hier zu berücksichtigen. Aber natürlich gibt es auch andere Dinge, die man bei der Finanzierung leicht vergessen könnte. Beispielsweise anfallende Kosten für Büromieten, Büroausstattung und Ähnliches. Die Tabelle 1.1 zeigt ein sehr vereinfachtes Beispiel, wie ein solches Budget aussehen könnte.¹²

Posten	Kosten x Anzahl	Gesamtkosten
Projekt		
Projektleitung	€ 6.000 x 24	€ 144.000
Programmierung		
Programmierung (Lead)	€ 4.000 x 24	€ 96.000
Programmierer (Game)	€ 3.000 x 24	€ 72.000
Programmierung (Tools)	€ 3.000 x 24	€ 72.000

Tabelle 1.1:
Vereinfachtes
Beispielbudget für
ein kleines Projekt

12 Die Daten für die Gehälter der Teammitglieder basieren auf den Ergebnissen einer Umfrage, die Marc Kamradt bei deutschen Entwicklern durchgeführt hat. Aktuelle Daten sind immer auf der Internetseite <http://www.spieleentwickler.org> zu finden.

Tabelle 1.1:
Vereinfachtes
Beispielbudget für
ein kleines Projekt
(Forts.)

Posten	Kosten x Anzahl	Gesamtkosten
Grafik und Design		
Level-Designer (Lead)	€ 2.500 x 24	€ 60.000
Level-Designer	€ 2.000 x 24	€ 48.000
Grafiker	€ 2.500 x 24	€ 60.000
Modeller	€ 2.000 x 24	€ 48.000
Lizenzen und Software		
Discrete '3D Studio Max 5'	€ 5.000 x 3	€ 15.000
Id Software 'Quake II Engine'	€ 10.000 x 1	€ 10.000
Adobe 'Photoshop 7'	€ 1.200 x 1	€ 1.200
Miete und Ausstattung		
Büroräume (Miete)	€ 1.000 x 24	€ 24.000
Computer	€ 1.000 x 7	€ 7.000
Gesamt		€ 657.200

*Ein Wort zum
Beispiel*

Beachtet, dass das Team für das Projekt mit acht Leuten recht klein gehalten ist. Als Projektdauer werden hier 24 Monaten angesetzt, und als Haupttool wird die *Quake II*-Engine von *id Software* kalkuliert.¹³ Dabei handelt es sich schon um eine recht alte Technologie, eine Lizenz der neueren *Quake III*-Engine kostet jedoch einen gewissen Prozentsatz auf verkaufte Exemplare des damit entwickelten Titels, mindestens aber 250.000 US-\$. Für die Level-Designer und den Modeller spendieren wir hier je eine Lizenz eines 3D-Modellierungstools und eine Lizenz für ein Grafikprogramm für den Grafiker. Diese Aufstellung in Tabelle 1.1 ist jedoch mit Vorsicht zu genießen. Es handelt sich hier nur um die offensichtlichsten Posten, die man zu berücksichtigen hat, und das Budget ist bei weitem nicht vollständig. Insbesondere fehlen hier weitere Kosten, beispielsweise für Geschäftsreisen zu Verhandlungen mit dem Publisher, Kosten für eine Rechtsberatung bei den Vertragsentwürfen oder Kosten für die Erstellung von Soundeffekten und Cutscenes, zu entrichtende Steuern usw. Dieses Beispiel soll nur eine einfache Grundlage darstellen, damit ihr seht, in welche Richtung so ein Budget gehen wird.

Schedule

Neben dem finanziellen Fahrplan braucht man natürlich auch einen zeitlichen Fahrplan für das Projekt. Hier wird nicht nur festgehalten, wie lange das Projekt insgesamt dauern soll. Vielmehr handelt es sich dabei auch um eine genaue Definition von Milestones und Miniature Milestones sowie deren genaue Terminierung. Anhand dieser Daten kann der Publisher einerseits

¹³ <http://www.idsoftware.com/business/home/technology/>

abschätzen, wie viel Entwicklungszeit er einkalkulieren muss. Andererseits bietet ihm der Zeitplan aber auch eine Möglichkeit, während der Laufzeit des Projekts zu kontrollieren, inwieweit die Arbeit wirklich vorankommt. Verschiedene Verfahren, wie zum Beispiel die so genannten Netzplantechniken und Tools wie *MS Project* helfen einem bei der Visualisierung von Zeitplänen und Abhängigkeiten der Milestones.

Die Einschätzung der Arbeitszeit ist insbesondere für Anfänger sehr schwierig. Man sollte auch bereits bei seinen ersten Hobby-Projekten damit beginnen, vor dem Startschuss eine grobe Abschätzung in Arbeitsstunden zu machen, wie lange man für dieses und jenes brauchen wird. Im Nachhinein kontrolliert man sich dann selbst in seinen Leistungen, wobei man natürlich nicht schummeln sollte. So gewinnt man im Laufe der Zeit einen großen Schatz an Erfahrungswerten, die die einzige solide Grundlage für die Aufstellung von Projektplänen sind. Hier wird man in der Regel auch sehr schnell erkennen, dass man die benötigte Arbeitszeit zu Beginn um 50% bis 100% unterschätzt.

Weitere Elemente

Neben den oben genannten Dingen gehören aber noch andere Informationen in ein Game Proposal. Beispielsweise sind technische Spezifikationen, minimale Systemanforderungen, die Entwicklungsumgebung und ähnlich banale Dinge ebenfalls für den Publisher von Wichtigkeit, da diese auch im angepeilten Markt verfügbar sein müssen.

Publisher, Veröffentlichung und Erlös

Wie ein Computerspiel erstellt wird, haben wir nun geklärt. Dabei haben wir erst einmal mit der weit verbreiteten Vorstellung aufgeräumt, dass das Spiel eigentlich nur programmiert und die Grafik erstellt werden muss. Doch wie kommt ein Computerspiel eigentlich in das Regal eines Kaufhauses oder in das Sortiment eines Online-Versandhandels? Nun, mit Computerspielen verhält es sich ähnlich wie mit Büchern. Ein Autor schreibt ein Buch, so wie ein Team ein Computerspiel entwickelt. Bevor jedoch die Kernarbeit an dem Buch (Computerspiel) beginnt, also das eigentliche Schreiben (Programmieren), erstellt der Autor eine Gliederung und ein Konzept, die das geplante Buch grob skizzieren (das Design-Dokument). Dieses sendet er an in Frage kommende Verlage (Publisher), die eventuell vorher schon ihr Interesse bekundet haben. Ist man sich über das geplante Werk einig, werden die Konditionen für den Vertrag ausgehandelt, und der Autor beginnt mit dem Schreiben des Buches. Ist es fertig, dann geht es an den Verlag, der dann die Aufgabe hat, das Buch zu drucken und über Distributoren bzw. landesweit agierende Vertreter an Buchhändler heranzutragen, damit diese es in ihr Sortiment aufnehmen. Ebenso obliegt es dem Verlag, Werbung für das Buch auf Messen und in Veröffentlichungen zu machen und Lizenzen für eine Übersetzung zu vergeben.

*Wie kommt ein
Computerspiel ins
Regal?*

Bei einem Computerspiel verhält es sich recht ähnlich. Ein Team findet sich zusammen und entwickelt die Vision eines gemeinsamen Spiels. Dazu schreiben die Teammitglieder das oben erwähnte Design-Dokument und das Proposal für das geplante Spiel. Sobald diese Dokumente fertig sind und eventuell schon eine erste Tech-Demo des Projekts existiert, wendet sich das Team an verschiedene Publisher, die in Frage kommen. Dort stellen sie ihr Proposal vor und versuchen, den Publisher von ihrer Idee zu überzeugen.



Man sollte es vermeiden, erst in einem sehr späten Stadium eines Projekts einen Publisher zu suchen. Für gewöhnlich wird der Publisher eigene Wünsche und Ideen haben, die er in das Game-Design integriert sehen möchte, um das Projekt der Produktlinie des Publishers anzupassen. Daher sollten die Arbeiten noch nicht voll im Gange sein, um entsprechende Änderungen einflechten zu können.

Wenn man mit einem nahezu fertig gestellten Projekt zu einem Publisher geht, wird dieser das Gefühl haben, den Entwicklungsprozess nicht nach seinen Marktkenntnissen mitgeformt zu haben. Das wird dazu führen, dass die Chancen zur Marktdurchdringung durch den Publisher schlechter eingeschätzt werden können und daher eher vorsichtig kalkuliert werden. Das wirkt sich natürlich auch auf den Vertrag mit dem Entwicklerteam aus.

Verkaufserlöse

Hat man dann einen geeigneten Publisher gefunden, wird ein entsprechender Vertrag ausgehandelt. Ein Modell sieht dabei beispielsweise vor, dass die Entwickler prozentual an den Verkäufen beteiligt werden. Betrachtet man den Endpreis eines Computerspiels im Laden, so erhält der Distributionskanal (hier: der Einzelhandel) ca. 50% des erzielten Erlöses. Die andere Hälfte des Geldes fließt in die Kassen des Publishers. Dieser finanziert mit dem Geld beispielsweise die Werbung und natürlich insbesondere auch die Herstellung und den Vertrieb des Spiels. Das Entwicklerteam kann damit rechnen, zwischen 10 und 25% Anteil am Erlös des Publishers zu nehmen. Dies bezeichnet man als Marge (engl. Royalties). Bei einem Titel, der im Regal 50 € kostet, ist der Erlös des Publishers also ca. 25 €, wovon das Entwicklerteam ca. 3,75 € erhalten wird.

Vorauszahlung und Break Even

Weiter oben hatte ich aber schon erwähnt, dass ein Publisher ein solches Projekt finanziert, wenn er sich dafür entschieden hat. Das Entwicklerteam kann ja nicht zwei Jahre umsonst arbeiten und dann darauf warten, dass die Marge in seine Kassen fließt. Selbst wenn man für die Chance, ein Computerspiel entwickeln zu können, sterben würde – so wörtlich, dass man es auf's Verhungern ankommen lässt, nimmt man es dann doch meistens nicht. Und das ist auch gut so. Der Publisher finanziert das Team also vor. Er zahlt in der Regel nach den ausgehandelten Konditionen das Geld, das das Team im Game-Proposal im Budget kalkuliert hat. Bleiben wir mal bei dem vereinfachten Budgetbeispiel von oben, und setzen wir für ein Zweijahresprojekt mit einem kleinen Team von acht Personen Kosten in Höhe von

700.000 € an. Dieses Geld erhält das Team vom Publisher, und es wird dann beim Erscheinen des Titels mit der zu zahlenden Marge verrechnet.

Nehmen wir hier einmal die prognostizierten Verkaufszahlen von 150.000 Stück für einen erfolgreichen Titel. Damit erwirtschaftet der Handel einen Umsatz von ca. 7,5 Millionen Euro, über die gesamte Verkaufsdauer gerechnet. Davon gehen ca. 3,75 Millionen Euro Erlös direkt an den Publisher, der damit aber auch noch seine Ausgaben für die Produktion, das Marketing usw. abdecken muss. Das wiederum bedeutet für das Entwicklerteam Erlöse aus der Marge von ca. 570.000 €. Das klingt doch schon ganz gut, oder? Dabei sollte man allerdings nicht vergessen, dass man von dieser Summe noch das Geld abziehen muss, das einem der Publisher während der zwei Jahre Entwicklungszeit vorgeschossen hat. Damit erhält das Entwicklerteam also keine weiteren Zahlungen für das Projekt, und die Vorfinanzierung durch den Publisher hat dem Team sozusagen keinen Gewinn im eigentlichen Sinne eingebracht. Es wurden lediglich die laufenden Kosten für die Entwicklungszeit gedeckt. Das Team hingegen steht finanziell bestenfalls genauso da wie vor der Entwicklungszeit.

*Gewinn des
Teams*

Damit ist der Job des Spiele-Entwicklers ein Job wie jeder andere auch, jedenfalls vom finanziellen Aspekt her gesehen. Ein durchschnittlich erfolgreicher Titel führt nicht zu unendlichem finanziellen Segen. Eher im Gegenteil: Man verdient in dieser Branche für gewöhnlich weniger als bei einem vergleichbaren Job in der Applikationsentwicklung. Und ganz genauso wie jeden anderen Job kann man diesen auch wieder verlieren: nämlich dann, wenn ein Team zwar schon einen recht ordentlichen Titel herausgebracht hat, aber für sein nächstes Projekt keinen Publisher mehr findet. All dies sind Faktoren, die es bei der Auswahl der zukünftigen Profession zu bedenken gilt. Selbst ein sehr erfolgreiches Spiel ist kein Garant für irgendetwas, da man aus den erzielten Erlösen keine finanziellen Rücklagen bilden kann, mit denen man das Team selbst finanzieren könnte, bis ein Publisher gefunden ist. Wenn nach dem Abschluss eines Projekts nicht gleich ein weiterer Deal abgeschlossen werden kann, sinkt die Liquidität des Teams dramatisch ab, und der Geschäftsführer des Teams kann seine Mitarbeiter nicht mehr bezahlen.

Ein normaler Job

Verkaufszahlen

Wo wir gerade bei Zahlen sind, möchte ich noch ein paar Daten angeben, was die Verkaufszahlen von Computerspielen angeht. Wir betrachten hier primär den deutschen Markt, der mit ganz anderen Schwierigkeiten zu kämpfen hat als der US-amerikanische. Das Geschäft auf dem US-amerikanischen Markt ist zwar auch hart, aber allein durch die internationale Ausrichtung liegen dort die zu erwartenden Verkaufszahlen durchschnittlicher Titel im Bereich von 100.000 und mehr Exemplaren. Dadurch amortisiert sich die Investition eines Publishers in ein Projekt eines Entwicklungsteams

VUD

mit einer höheren Wahrscheinlichkeit als im deutschsprachigen Markt. Hierzulande beginnt man bei mageren Verkaufszahlen von ca. 40.000 bis 100.000 Stück. Der Verband der Unterhaltungssoftware Deutschland e.V. (VUD)¹⁴ vergibt beispielsweise Sales-Awards für die meistverkauften Computerspiele aus deutschen Ländern. Ab 100.000 Exemplaren erhält ein Titel den Gold-Award, ab 200.000 Titeln gibt es den Platin-Award und bei 400.000 Titeln logischerweise Doppel-Platin. Diese Zahlen lassen bereits erahnen, dass die entsprechenden Verkaufszahlen im deutschen Markt nicht wirklich einfach zu realisieren sind.

*Ausnahmen von
der Regel*

Natürlich gibt es auch Computerspiele aus deutschsprachiger Entwicklung, die sehr erfolgreich sind. Ein gutes Beispiel hierfür ist das Spiel *Anno 1503* des deutschen Publishers *Sunflowers*. Innerhalb weniger Tage nach der Veröffentlichung im Oktober 2002 erhielt das Spiel den Gold-Award, nach gut zwei Wochen waren bereits über 200.000 Einheiten verkauft. Es setzt damit die Erfolgsgeschichte des Vorgängers *Anno 1603* fort, der bereits Doppel-Platin erreicht hat. Eine solche Erfolgsstory ist also auch mit deutschsprachigen Produktionen möglich, auch wenn der internationale Markt in anderen Dimensionen rechnet. Dennoch bleiben solche Verkaufserfolge nur wirklich wenigen Spielen vorbehalten. Als deutsches Entwicklerteam sollte man also keine Rosinen im Kopf haben, was den finanziellen Erfolg eines Projekts angeht.

Alternativen

Kredite?

Nun stellt sich die Frage nach alternativen Strategien in der Entwicklung und Vermarktung von Computerspielen. Bei der aktuellen Marktlage, die ihr ebenfalls in den Veröffentlichungen des VUD nachlesen könnt, wird ein Team es sehr schwer haben, einen Publisher zu finden. Alternative Konzepte zur Finanzierung der Entwicklung sind also gefragt. Leider gibt es hier wenig gute Aussichten, denn die Vorfinanzierung durch einen anderen Geldgeber, wie beispielsweise eine Bank, ist in den seltensten Fällen möglich. Die zu entwickelnde Software wird von den Banken für gewöhnlich nicht als Sicherheit für einen Kredit akzeptiert. Selbst mit einem ausgereiften Business-Plan und einer realistischen Budget-Kalkulation lässt sich keine Bank überzeugen, da sie, falls ein solches Projekt scheitert, keine finanziell verwertbaren Komponenten aus dem aufgelösten Team ziehen kann. Im Gegensatz zu einem Startup-Unternehmen in einer anderen Branche, das für das Geld aus dem Kredit beispielsweise Anlagen und Maschinen beschafft, die einen finanziellen Gegenwert darstellen, wird das Geld des Kredites bei einem Entwicklerteam zu großen Teilen allein für Gehaltszahlungen ausgegeben, ohne durch finanziell greifbare Werte ersetzt zu werden.

¹⁴ <http://www.vud.de>

So oder so wäre ein Kredit jedenfalls nicht die beste Wahl, denn im Falle eines wirtschaftlichen Misserfolgs des Titels trägt hier der Entwickler das Risiko selbst und nicht mehr der Publisher. Wird nach der Veröffentlichung weniger Geld eingespielt, als die Entwicklung gekostet hat, dann steht das Team am Ende auch noch mit Schulden da.

Eine andere Alternative ist die Entwicklung eines Computerspiels als Hobby. Auch wenn man als Anfänger in dem Bereich gern sagt, ich komme mit so und so viel Euro im Monat aus, ich lebe ohne große Ansprüche, usw., sollte man das relativieren. Spätestens wenn man nicht mehr nur sich selbst versorgt, sondern eine Familie hat, ist man auf ein gewisses Einkommen angewiesen und kann sich nicht auf finanzielle Abenteuer einlassen, ohne seine Zukunft zu gefährden. In einer solchen Situation kann es durchaus ein erfolgreiches Konzept sein, ein Computerspiel als Hobby-Projekt zu entwickeln und dann kommerziell zu vertreiben.

Hobby?

Das Internet bietet mittlerweile genügend Möglichkeiten, dies erfolgreich zu tun. Aber auch hier sollte man sich bewusst sein, dass man mit Rosinen im Kopf allein nicht weit kommt. Insbesondere das Zusammenhalten und die Langzeitmotivation eines Teams bei einem derartigen Projekt stellen hier das Problem dar. Für ein solches Projekt benötigt man Leute, die wesentlich motivierter sind und dem Projekt absolute Treue halten. Diese sind nur schwer zu finden, und daher erscheint diese Alternative auch nur in Ausnahmesituationen gangbar. Hat man jedoch ein entsprechendes Team bereits um sich gesammelt und bereits mehrere Monate oder sogar Jahre hinweg zusammen an kleineren Dingen erfolgreich gearbeitet, so lohnt es sich durchaus, darüber nachzudenken.

Damit verlassen wir den Bereich der betriebswirtschaftlichen Betrachtung der Entwicklung eines Computerspiels. Der folgende Abschnitt setzt sich mit einigen der Tools auseinander, die ihr für die Entwicklung eines Computerspiels benötigen werdet.

Cut

1.4 Sage mir, mit wem du gehst ...

Weiter oben hatte ich bereits die beiden bedeutendsten APIs im Bereich der Grafik-Programmierung erwähnt. Es ist dies zum einen die DirectGraphics-Bibliothek als Bestandteil von DirectX und die OpenGL-Bibliothek. Beide APIs verwenden unterschiedliche Konzepte zur Umsetzung der Aufgabe; was jedoch die Strukturierung der Funktionsaufrufe angeht, näherte sich DirectX mit jeder erscheinenden Version immer ein Stück weiter an OpenGL an.

Der Hauptunterschied zwischen den beiden APIs besteht wohl nur noch darin, dass DirectX strikt objektorientiert aufgebaut ist. OpenGL hingegen ist im C-Stil rein funktional programmiert. Natürlich lässt sich OpenGL

Unterschiede

auch in eigene Klassen wrappen, aber dort arbeitet man dann nicht mit Objekten, sondern nur mit den reinen Funktionsaufrufen der API. In den DirectX-APIs hingegen arbeitet man immer mit Instanzen von Klassen, über die man die Funktionen aufruft.

Flame Wars

Diese Tatsache sollte man einfach so akzeptieren. Leider trifft man aber immer wieder auf Diskussionen in Online-Foren, die sich einzig und allein um die Frage drehen: Wer ist besser, wer ist schneller, wer ist höher und wer ist weiter – DirectX oder OpenGL? Geführt werden solche Diskussionen meistens auf der Basis von Unwissenheit und Argumenten, die man vom Bruder des Cousins der Schwester des Typen gehört hat, dessen Bruder mit dem Auto seines Großvaters neulich die Katze des Nachbarn überfahren hat. Zwischen OpenGL und DirectX gibt es trotz einiger Ähnlichkeit immer noch gewisse Unterschiede in der Handhabung, und wer ohne ein besseres Verständnis für die jeweilige API ein Programm von der einen API eins zu eins auf die andere API portiert, der wird in vielen Fällen auch gravierende Geschwindigkeitsunterschiede feststellen. Durch eine leichte Anpassung des Programms an die jeweilige API und nicht einfach nur durch ein blindes Umschreiben der Funktionsnamen wird man beide Versionen jedoch äquivalent schnell machen. Wer also in der Verwendung beider APIs ausreichend vorgebildet ist, für den wird es solche Flame Wars nie geben. Es gibt in dieser Hinsicht einfach kein Besser oder Schlechter. Man kann allenfalls rein subjektiv mit »Das mag ich mehr und das weniger« argumentieren.

*Und was man
immer wieder
gern vergisst ...*

Letzten Endes sind es nämlich insbesondere die Treiber für die Grafikkarte, die die Geschwindigkeit zweier Programme mit identischer Ausgabe auf dem Monitor bei den beiden verschiedenen APIs bestimmen. Ein Update der Treiber einer Grafikkarte auf die neuesten verfügbaren Versionen kann da manchmal Wunder wirken. Letzten Endes ist es also egal, welche der beiden großen Grafik-APIs man bemüht. Beide sind mehr oder weniger gleichwertig, was die Erlernbarkeit der gesamten Funktionalität (nicht nur der Initialisierung und des Renderns eines Dreiecks), die Geschwindigkeit und die Flexibilität angeht. OpenGL ist DirectX nur in einem Punkt klar voraus, und das ist die Verfügbarkeit der API auch unter Unix/Linux und nicht nur unter Windows. Doch diese Tatsache wird allzu leicht mit einer leichten und schnellen Portierung einer für diese API geschriebenen Software von dem einen Betriebssystem auf das andere verwechselt. Dabei stellt sich auch immer die Frage, wie sinnvoll es ist, ein Betriebssystem für Spiele zu unterstützen, das nur eine absolute Minderheit der Zielgruppe verwendet.

*Und mit wem
gehen wir?*

Mein Ziel ist es, in diesem Buch eine vollkommen grafik-API-unabhängige Engine zu entwickeln. Da stellt sich doch gleich die Frage, warum tue ich das obwohl es dafür eigentlich keinen zwingenden Grund gibt. Die Antwort ist einfach: Dieses Buch dient in erster Linie nicht dazu, ein konkretes Projekt zu implementieren, sondern dazu, die Programmierung einer Engine zu erlernen. Es macht auf alle Fälle Sinn, die Funktionen einer bestimmten API

so weit vom Rest des Programms zu trennen, dass man bei einem Update dieser API schnell zur neuen Version wechseln kann, ohne seinen gesamten Quellcode durchforsten und ändern zu müssen. Idealerweise verwendet man eine Grafik-API daher eben nicht dazu, überall im Programm bestimmte Funktionen dieser API aufzurufen, sondern man sollte die API wirklich nur dazu verwenden, um vom Programm erstellte Dreieckslisten über die Grafikkarte auf dem Monitor auszugeben. Man kann also die Funktionalität des Programms strikt von der Grafik-API trennen, so wie wir es hier tun werden. Eigentlich brauchen wir ja von der Grafik-API nichts weiter als den Funktionsaufruf `Rendere_Dreiecke()`.

Dass es ganz so einfach nicht ist, werden die folgenden Kapitel zeigen. Schließlich haben wir es noch mit Projektionsmatrizen, Texture-Stages und so weiter zu tun. Aber langer Rede kurzer Sinn: Ich werde in diesem Buch DirectX verwenden. Der Renderer, also der Teil unseres Programms, der DirectGraphics verwendet, wird aber so weit vom Rest des Programms getrennt sein, dass man ihn auch durch eine äquivalente OpenGL-Implementierung ersetzen kann.

1.5 DirectX und OpenGL, Versionspolitik

Es gibt noch einen Punkt, in dem sich DirectX und OpenGL voneinander unterscheiden. Das ist die Philosophie, mit der neue Features in die APIs integriert werden. Wann immer die Hardware, also die Grafikkarten, technische Neuerungen bietet, muss eine API ergänzt werden. Anderenfalls verliert sie ihre Aktualität und veraltet relativ schnell.

Bei DirectX geht man den Weg der neuen Versionen. Ungefähr alle fünfzehn Monate erscheint eine neue Version dieser API, die manchmal mehr und manchmal weniger Änderungen mit sich bringt. Als Grundsatz gilt hier, dass man immer mit der neuesten Version dieser API arbeiten sollte. Andererseits ist es bei einem sehr weit fortgeschrittenen Projekt auch nicht immer sinnvoll, kurz vor dem Release wichtige Komponenten der Software, wie beispielsweise den Renderer, noch mal anzufassen und umzuschreiben. Bei dem Wechsel von DirectX 7 auf DirectX 8 gab es insbesondere im Bereich der Grafik einen sehr radikalen Umbruch durch den Wegfall der 2D-Komponente DirectDraw.¹⁵ Der Wechsel von DirectX 8 auf DirectX 9 hingegen brachte nur wenige minimale Änderungen in der Interface-Struktur mit sich.

DirectX

Bei OpenGL ist die Strategie vollkommen anders. Neue Versionen von OpenGL gibt es nur in sehr großen Zeitabständen von mehreren Jahren. Die Anhänger dieser API führen das gern auf die Tatsache zurück, dass die Spe-

OpenGL

¹⁵ Die damals aktuellen Grafikkarten emulierten die 2D-Funktionalität von DirectDraw sowieso nur noch über echte 3D-Grafik-Objekte, was entsprechend langsamer war, als auch 2D-Grafik direkt über 3D zu implementieren.

zifikationen der API von Anfang an entsprechend gründlich geplant wurden. Das ist aber nicht ganz korrekt, denn technischen Neuerungen wird in OpenGL auf eine andere Art und Weise Rechnung getragen. Es gibt hier die so genannten *Extensions*, die über ein entsprechendes Treiber-Update umgesetzt werden. Der Programmierer verwendet weiterhin die aktuelle OpenGL-Version, kann aber die Grafikkarte abfragen, was für Extensions sie bietet. Ist eine entsprechende Extension vorhanden, beispielsweise für Vertex-Shader, so kann der Programmierer Vertex-Shader über diese Extension verwenden. Wird jedoch eine neue Version von OpenGL herausgebracht, dann basiert diese hauptsächlich darauf, dass ältere Extensions in den Standard übernommen werden, da sie mittlerweile auf jeder neuen Grafikkarte verfügbar sind.

1.6 Warum schwer, wenn's auch einfach geht?

Hier kommt wieder ein allseits beliebtes Thema für »flamige« Diskussionen. Wir werden es uns in diesem Buch so schwer wie nur irgend möglich machen. Getreu dem Motto: »*Nur wer in der Hölle war, kann den Himmel wirklich sehen*« erspare ich uns und euch in diesem Buch kein Stück Handarbeit. Es gibt zwar viele schöne Tools da draußen, die uns das Leben leichter machen könnten, seien das nun die STL, WinAPI-Funktionen, FMod oder gar die D3DX-Hilfsbibliothek. Es gibt allerdings handfeste Gründe, warum ich mich dem in meinen Büchern größtenteils verschließe. Wir sind hier, um etwas zu lernen. Und wir wollen nicht lernen, wie ich die Funktionen aufrufe, die ein anderer geschrieben hat. Dazu gibt es Referenzen, und man braucht keine Bücher dafür.

D3DX-Bibliothek

Betrachten wir einmal die folgende Situation. Seit der Version 7 von DirectX gibt es die so genannte D3DX-Bibliothek, die allerlei schicke Funktionen bietet, die einem die Arbeit an einer Engine sehr erleichtern. Insbesondere die 3D-Mathematik ist dort sehr umfangreich implementiert. Durch Verwendung eben jener Funktionen ist es für einen Einsteiger sehr leicht, schnell etwas 3D-Mäßiges auf dem Bildschirm anzuzeigen. Fragt man ihn jedoch, warum er nun dieses oder jenes an einer bestimmten Stelle gemacht hat, dann wird in der Regel Folgendes hören: »*Na, das hab ich aus diesem Sample-Code rauskopiert ...*«

Fakt ist, dass man so den Ansatz einer 3D-Engine entwickeln kann, ohne selbst Ahnung von der Materie zu haben. Das ist gut – für diejenigen Entwickler, die eine 3D-Anzeige nur als zusätzliches Feature in einer anderen Software benötigen und aus firmeninternen Gründen keine Zeit und Mittel haben, sich langwierig in das Thema einzuarbeiten. Schlecht ist das aber für diejenigen, die das Letzte aus ihrer Grafikkarte herausholen wollen oder die andere Funktionalitäten brauchen als die gegebenen. Zugegebenermaßen sind die D3DX-Hilfsfunktionen für jegliche Prozessoren in Maschinenspra-

che optimiert. Sie nutzen SIMD-Technologien wie beispielsweise 3DNow! oder SSE (siehe Kapitel 4) und sind sehr, sehr schnell. Durch Unwissenheit kann man aber diesen Geschwindigkeitsvorteil auch verspielen. Möchte man beispielsweise ein Objekt im 3D-Raum verschieben, so kann man eine D3DX-Hilfsfunktion aufrufen, die diese Verschiebung in der Matrix des Objekts speichert. Alternativ könnte man, wenn man weiß, wie es funktioniert, diese Verschiebung auch direkt in der Matrix des Objekts setzen. So spart man sich den Overhead, eine Funktion aufrufen zu müssen, was in diesem Fall fast länger dauert als das eigentliche Ausführen der Funktion.

Dieses einfache Beispiel zeigt, dass es an vielen Stellen Sinn macht, wirklich zu wissen und auch zu begreifen, was eine bestimmte Funktion macht und warum man sie braucht. Sobald man Performance aus seinem Programm herausholen möchte oder auch nur ein wenig über den Tellerrand hinausschaut und Dinge implementieren will, für die es eben keine vorgefertigte Hilfsfunktion gibt, wird man es immer sehr schwer haben, wenn man zuvor noch nie die Pixel einer Bitmap angefasst, die Rotationswerte in einer Drehmatrix berechnet oder ein Billboard selbst an einer Kamera ausgerichtet hat.

*Wissen statt
nachmachen*

Letzten Endes muss hier aber jeder seinen eigenen Weg kennen. Es gibt bei diesem Thema grundsätzlich kein Richtig und Falsch. Wer sich einmal die Mühe gemacht hat, die interne Funktionsweise und die prinzipiellen Abläufe in einer 3D-Engine zu verstehen, der soll sich nicht scheuen, beispielsweise auf die D3DX-Hilfsfunktionen zurückzugreifen. Deren Implementierung wird von schlaun Köpfen bei Microsoft, Intel, AMD, NVidia, ATI und vielen anderen vorgenommen und ist entsprechend schnell. Den Quellcode dieser Funktionen veröffentlicht Microsoft beispielsweise auch deshalb nicht, weil hier zum Teil sehr spezifische Implementierungen für bestimmte Prozessoren oder Grafikkarten verwendet werden, die die jeweiligen Hersteller aus Konkurrenzgründen nicht veröffentlichen möchten.

Der eigene Weg

Aber jeder ist hier frei, sich seine eigene Philosophie über die Verwendung solcher Hilfen zu bilden. Meine Meinung ist aber, dass man vorher auf alle Fälle gelernt haben solltewas genau diese Funktionen bewirken und wie man das selbst zu Fuß machen könnte.

1.7 Auf los geht's los

Nun haben wir uns mit den wichtigsten Grundbegriffen befasst, die wir als Voraussetzung für die Arbeit mit diesem Buch benötigen. In diesem Kapitel wollte ich allgemein darauf eingehen, mit was für einem Thema wir uns auf den Hunderten von folgenden Seiten beschäftigen werden. Wir haben ein hartes Stück Arbeit vor uns, aber wir haben auch den Vorteil auf unserer Seite, dass uns diese Arbeit Spaß machen wird. Jedenfalls sollte sie das.

Natürlich hat jeder seine Favoriten unter den Themen, die mit der Spiele-Entwicklung verbunden sind. Ich höre schon jetzt die Flüche der eingefleischten Mathe-Hasser, wenn es an das Kapitel über 3D-Mathematik geht. Ebenso werden diejenigen im Kapitel unserer Rahmenanwendung aufstöhnen, die Initialisierungsarbeit für die achte Plage der Menschheit halten. Ich werde mich bemühen, alles so einfach und verständlich wie möglich darzustellen. Aber es gibt eben Themen, durch die man sich durchquälen muss. Am Ende dieses Leidenswegs steht dann aber immer das Wissen als Gegenpol, das man sich durch seine harte Arbeit erworben hat.

Was Menschen aber in der Regel fürchten und daher mit Skepsis betrachten, das ist Unsicherheit. Unsicherheit über das, was da kommen wird. Ich erinnere mich hier gern an meine Zeit als Soldat zurück. Am untersten Ende der Kommandokette sitzt der ordinäre Fußsoldat im abgedunkelten Kampfraum des Schützenpanzers, der durch das holprige Gelände rast, und sieht seine Hand vor Augen nicht. Plötzlich bremst der Panzer, und die Heckklappe geht auf. Automatisch sitzt man in Richtung der nächsten Deckung ab und geht in Stellung. Der Truppführer aus den Unteroffiziersrängen deutet mit dem Arm in die Richtung, in die man die Waffe richten soll. Immerhin kennt er wenigstens halbwegs den Auftrag, den sein Trupp erfüllen soll, aber ob man ihm überhaupt gesagt hat, in welchem Land er sich befindet, ist unwahrscheinlich. Wird der Truppführer getroffen, steht der Trupp mitten im Wald und weiß weder, wo er ist noch was er tun soll. Immerhin weiß der Kommandant des Schützenpanzers, wo sich die anderen Teile des Zugs befinden. Und der Zugführer weiß, wo sich die anderen Züge der Kompanie befinden. Der Kompaniechef letzten Endes weiß dann, wo sich die anderen Kompanien des Bataillons befinden und in welchem Land man gerade ist – hoffentlich.

In einer Armee macht eine solche Hierarchie des Wissens an einigen Stellen Sinn. Bei der Entwicklung eines Software-Produkts jedoch nicht. Daher dient das nächste Kapitel dazu, einen groben Entwurf dessen zu präsentieren, was wir im Verlauf der danach folgenden Kapitel programmieren wollen. Das sollte jedem Leser einen guten Überblick über die Gesamtsituation verschaffen. Somit fällt die Orientierung im Gelände unserer 3D-Engine nachher wesentlich leichter. Im nächsten Kapitel können wir uns also noch ein wenig entspannt zurücklehnen und müssen hauptsächlich unsere Augen und Ohren spitzen, um den Gesamtplan zu erfassen.

2 Design der ZFXEngine

»Den lieb ich, der Unmögliches begehrt.«
(Johann Wolfgang von Goethe)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Anforderungen an die *ZFXEngine*
- ➔ Design und Aufbau der *ZFXEngine*
- ➔ Zusammenspiel der einzelnen Module
- ➔ Interfaces und API-Unabhängigkeit

2.1 Anforderungen an die Engine

Schon der alte Goethe wusste das Streben nach schier Unmöglichem zu schätzen. Und auch die nun vor uns liegende Aufgabe mutet gewaltig an. Hat unser Meister uns doch aufgetragen, wir mögen ihm eine Engine entwickeln, welche das Volk in Staunen zu versetzen mag. Wohl an, meine Herren, stellen wir uns sogleich dieser Aufgabe. :-)

Der alte Goethe

Zunächst verpassen wir unserer Engine einen schnuckligen Namen, und zwar nennen wir sie in Anknüpfung an eine alte Tradition einfach *ZFXEngine 2.0*. Die erste Engine namens *ZFXEngine* wurde ja bereits vor einigen Jahren für die Umsetzung des Lokalmatadors *Wing Captain 1.0* verwendet.¹ Aus Kapitel 1 wissen wir schon das eine oder andere über Engines. Insbesondere können wir eine Anforderung an unsere Engine direkt ableiten. *Unsere Engine muss ...*

*Anforderungen an
die ZFXEngine 2.0*

- ➔ ... von einem konkreten Projekt vollkommen abgekoppelt sein.

1 Damit ist natürlich das Tutorial über die Programmierung von Direct3D 5 im Retained Mode von der Internetseite <http://www.stefanzerbst.de> gemeint. Im Verlauf dieses Tutorials, das quasi den Grundstein von ZFX bildete, wurde die einfache Space Combat Sim *Wing Captain 1.0* in Anlehnung an Origins erfolgreiche Serie *Wing Commander* entwickelt.

Diese Minimalanforderung an eine Engine besagt, dass innerhalb des Codes der Engine keine Funktion, kein Parameter, keine Variable und keine Klasse oder Struktur vorkommen darf, die zu einem konkreten (Spiele-)Projekt gehört. Wäre das der Fall, dann könnten wir unsere Engine nicht für vollkommen unabhängige andere Projekte wiederverwenden. Nun wird zwar jemand voreilig sagen wollen: »*Das ist doch aber eh das Prinzip der objekt-orientierten Programmierung!*« Aber das stimmt so natürlich nicht.

*Engine Game-
Code-abhängig
oder nicht?*

Man kann ja beispielsweise in seinem Game-Code eine Klasse `CIhtarlikFighter` haben, die ein Raumschiff einer fiesen Alien-Rasse implementiert. Aber nun gibt es zwei Wege, um eine Instanz dieses Raumschiffs auf den Bildschirm zu bringen. Man kann (A) dem Renderer über die Klasse eine Anzahl an Vertices und Indices und Texturen und Stage-Settings usw. übergeben und ihn dann rendern lassen. Alternativ ist es aber auch möglich, dass man (B) dem Renderer die Klasse `CIhtarlikFighter` bekannt macht. Dann muss man dem Renderer einfach diese Instanz übergeben und die Render-Methode im Renderer pult sich die notwendigen Informationen selbst aus dem Objekt heraus. Die beiden konkurrierenden Ansätze sehen wie folgt aus:



```
CIhtarlikFighter *pFighter = new CIhtarlikFighter();
```

```
// strategie A:
g_pRenderer->Render( pFighter->GetVertices(),
                    pFighter->GetNumVertices(),
                    pFighter->GetIndices(),
                    pFighter->GetNumIndices(),
                    pFighter->GetTexture() );
```

```
// strategie B:
g_pRenderer->Render( pFighter );
```

*Strategie B ist
schnell und
bequem*

Wie man schon sehen kann, ist Strategie B im Game-Code sehr bequem. Man hat sehr wenig Schreibarbeit. Ihr seht aber auch, dass die Klasse `CRenderer`, von der `g_pRenderer` aus dem Beispiel instanziiert wurde, die Klasse `CIhtarlikFighter` kennen muss, um eine Instanz dieser Klasse als Parameter aufzunehmen (Vererbungen und Basisklassen mal außen vor gelassen). Just in diesem Moment ist der Renderer auch eine Code-Komponente geworden, die nicht ohne den Code-Teil existieren kann, in dem diese Klasse existiert. Man kann den Renderer also nicht ohne Änderungen in einem Projekt einsetzen, in dem es keine Klasse `CIhtarlikFighter` gibt.

Diese Vorgehensweise hat aber auch ihre Existenzberechtigung. Wenn ein Renderer Kenntnis von allen Objekten hat, die er rendern soll, dann kann er entsprechende Optimierungen anbringen. Man geht damit im Game-Code von der Lowlevel-Ebene der Vertices und Indices weg und konzentriert sich auf eine Highlevel-Ebene von konkreten Objekten. Intern ist der Renderer

dann entsprechend auf jeden möglichen Objekttyp hin jeweils optimiert. Das kostet natürlich den Preis, dass man nur eine vorher festgelegte Menge an verschiedenen Objekten in einer bestimmten Version des Renderers verarbeiten kann. Man kann keine beliebigen anderen Objekte verarbeiten.

Die erstgenannte Strategie erscheint im Quellcode schon lästiger, weil man aus einem Objekt alle Informationen herauspulen muss, die man zum Rendern von Dreiecken benötigt. Der klare Vorteil dabei ist jedoch, dass eine Klasse `CRenderer` nur Vertices und Indices kennen muss. Nun kann der Renderer alles rendern, was aus Vertices und Indices besteht. Im Game-Code kann es jetzt beliebig viele verschiedene Klassen von Objekten geben, die dem Renderer vollkommen unbekannt sind. So lange sie Vertices und Indices liefern können, kann der Renderer sie auch verarbeiten. Nun kann man den Renderer ganz einfach vom Game-Code trennen und in einem anderen Projekt verwenden, ohne dass in diesem Projekt die Klasse `CIhtarlikFighter` bekannt sein muss.

Strategie A ist abstrahierend

Wir werden für unsere Engine also die Strategie A verwenden, denn wir möchten die Engine vollkommen losgelöst vom Game-Code entwickeln.

Nachdem das geklärt ist, kommen wir zu den weiteren Anforderungen an unsere Engine. Ich möchte hier nicht zu sehr ins Detail gehen, darum stelle ich einfach noch die folgenden Anforderungen vor, die eine Anwendung unserer Engine möglichst flexibel und bequem machen sollen. *Unsere Engine muss ...*

Weitere Anforderungen an die ZFXEngine 2.0

- ➔ ... effizient rendern können.
- ➔ ... Input über Tastatur, Maus und Joystick abfragen können.
- ➔ ... Soundeffekte abspielen können.
- ➔ ... Netzwerk-Verbindungen ermöglichen.

2.2 API-Unabhängigkeit durch Interface-Definitionen

Kommen wir nun zu der heiß geliebten Frage, ob DirectX² oder OpenGL besser ist. Natürlich gibt es darauf keine Antwort, weil das von vielen verschiedenen Komponenten, unter anderem auch der Laufzeitumgebung, abhängt: seien es nun die Fähigkeit des Programmierers, der Treiber, die Grafikkarte und letzten Endes auch die Operationen, die ausgeführt werden sollen. Damit solche Fragen aber gar nicht erst aufkommen, verwenden wir doch einfach beides. Wir implementieren beispielsweise nicht einfach stur Direct3D oder OpenGL in unsere Engine. Stattdessen gehen wir über so

Direct3D oder OpenGL?

2 Eigentlich kann man sowieso nur Direct3D mit OpenGL vergleichen, weil die anderen Komponenten von DirectX Funktionalitäten bieten, die OpenGL nicht mit umfasst.

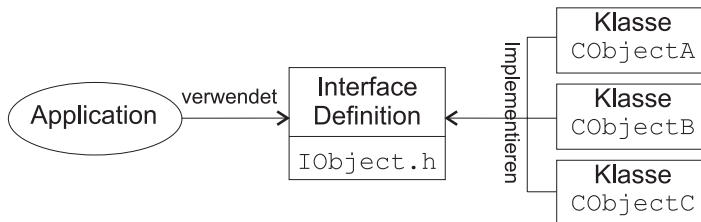
genannte Interfaces. Im nächsten Kapitel machen wir das auch wirklich mit Quellcode, und dort gehe ich etwas genauer auf diese Dinge ein.

Interface = virtuelle Klasse

An dieser Stelle sei nur so viel gesagt, dass ein Interface in C++ einfach eine rein virtuelle Klasse ist. Das bedeutet, dass man eine Klasse definiert, die nur `public`-Methoden enthält. Das ist quasi eine Liste von Methoden, die man von diesem Objekt erwartet. Aber diese Klasse wird man nicht implementieren, die Methoden werden also explizit nicht definiert (ausprogrammiert). Unser Programm kann ein solches Interface verwenden, um die benötigten Methoden aufzurufen. Hinter dem Interface stehen abgeleitete Klassen, die dann tatsächlich die Methoden implementieren und beliebige `private` und `protected` Attribute hinzufügen können, die sie benötigen. Abbildung 2.1 zeigt ein solches Interface namens `IObject.h`. Dahinter stehen drei verschiedene Implementierungen in den Klassen `CObjectA`, `CObjectB` und `CObjectC`. Welche API die Implementierung in den Klassen verwendet, ist der Applikation total egal. Und wenn alle drei Objekte verschiedene APIs verwenden, ist das auch Wurscht. Die Applikation ruft ja nur die Methoden auf, die das Interface zur Verfügung stellt. Ob die Implementierung der Objekte intern beispielsweise auf `Direct3D` oder `OpenGL` zugreift, stört die Anwendung nicht.

Abbildung 2.1:

Ein Interface definiert ein Set an Methoden, die eine Applikation verwenden kann, die mit diesem Interface arbeitet. Das Interface kann dann durch beliebig viele verschiedene Implementierungen umgesetzt werden.



Trennung der implementierenden Klassen von der Applikation

In dem Diagramm sieht das ja auch schön und gut aus. Bleibt die Frage, wie man aber nun wirklich eine Klasse, die von einem Interface abgeleitet ist und dessen Methoden implementiert, so vom Code der Applikation trennen kann, dass die Applikation trotzdem eine Instanz dieser Klasse auf einen Zeiger vom Typ des Interfaces erhält? Man braucht also folgende Konstellation:



```
CAppliation *pMyApp = new CAppliation();
```

```
// m_pObject vom Typ IObject*
pMyApp->m_pObj = new CObjectA();
```


Scheinbar muss die Applikation also doch die Klasse `CObjectA` kennen, auch wenn sie nur mit den Methoden von `IObject` arbeitet. Wie man dieses Problem umschiffen kann, sehen wir im nächsten Kapitel. Wir begnügen uns hier erst einmal mit der Tatsache, dass das geht. :-)

2.3 Struktur der ZFXEngine

Nun kommen wir zum wirklichen Aufbau, den wir unserer Engine verpassen werden. Werft einmal einen Blick auf Abbildung 2.2, und versucht nachzuvollziehen, was dort dargestellt ist.

Vier Interfaces und zwei Bibliotheken

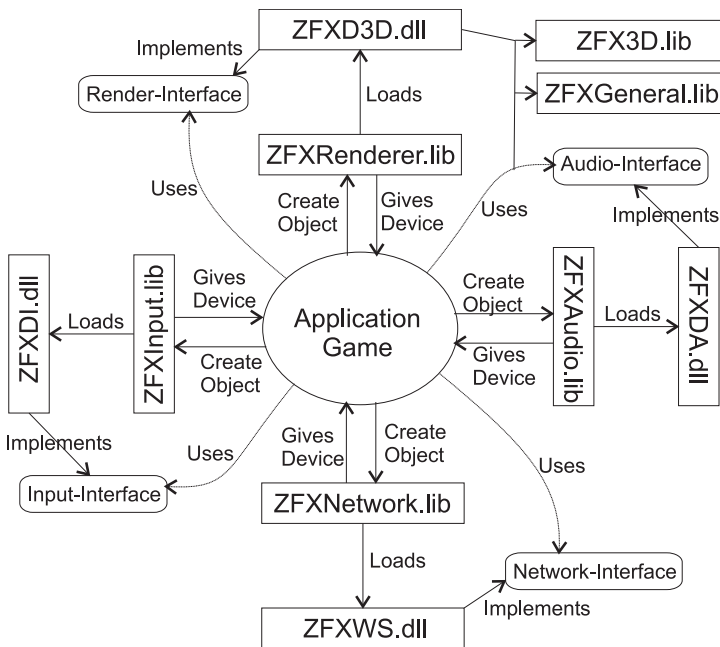


Abbildung 2.2: Aufbau und Design der ZFXEngine 2.0, die aus vier Interfaces und zwei Bibliotheken besteht. Eine Applikation verwendet beliebig viele dieser sechs Komponenten. Die vier Interfaces sind in mindestens je einer DLL implementiert, die über eine LIB geladen werden kann. (Weitere Erklärungen findet ihr im Text.)

Unsere *ZFXEngine* besteht nicht aus einer einzigen Datei, die in Form einer DLL oder LIB von einer Applikation verwendet werden kann. Stattdessen bietet die Engine vier Interfaces und zwei statische Bibliotheken, die eine Applikation alle oder auch einzeln verwenden kann. Dabei habe ich mich ein wenig am Aufbau von DirectX orientiert. Die jeweiligen Interfaces – ein Renderer, eine Input-Klasse, eine Audio-Klasse und eine Netzwerk-Klasse – sind in einer DLL implementiert. Diese DLLs werden jeweils durch eine zugehörige statische LIB geladen. Eine Applikation muss also die entsprechende LIB linken, um das entsprechende Interface verwenden zu können. Die Implementierung der Interfaces erfolgt im Rahmen dieses Buches nur in einer Version in je einer DLL. Man kann aber beliebig viele Implementierungen des Interfaces erzeugen. Dann hätte man mehrere DLLs zur Auswahl, die durch die entsprechende LIB geladen werden könnten.

Interfaces

Ihr erkennt, dass das Schema in der Abbildung für alle vier Interfaces gleich ist. Die Applikation verwendet einen Zeiger vom Typ der Interface-Klasse und arbeitet mit den Interface-Methoden. Eine DLL stellt eine mögliche Implementierung eines Interfaces zur Verfügung, die über eine statische LIB geladen werden kann. Die Applikation linkt dazu diese LIB und erstellt sich ein Objekt einer Klasse aus der LIB, die nur dazu dient, die DLL zu laden. Über dieses Objekt lässt sich die Applikation dann auf ihren oben genannten Zeiger die Adresse einer Instanz der Klasse geben, die das Interface implementiert. Diese Instanz wurde beim Laden der DLL in der LIB erstellt.

Keine Panik, wenn euch die Details, wie man das alles implementiert, noch sehr verwirrend erscheinen. Im nächsten Kapitel klären wir das alles Zeile für Zeile an echtem Quellcode. Nach der Lektüre des folgenden Kapitels werdet ihr eventuell noch einmal zu dieser Abbildung zurückkehren wollen. Dann werden all die kleinen Puzzle-Teile sich zu einem großen Gesamtbild zusammenfinden.

LIBs Neben diesen vier Interfaces und den entsprechenden Bibliotheken enthält die Engine auch noch zwei statische Bibliotheken, die 3D-Mathematik in umfangreichen Klassen zur Verfügung stellen (`ZFX3D.lib`) bzw. einen Timer und Kamerafunktionalitäten bieten (`ZFXGeneral.lib`).

2.4 Komponenten der ZFXEngine

Im folgenden Abschnitt sind noch einmal kurz die Komponenten der Engine beschrieben, die wir im Verlauf dieses Buches entwickeln wollen.

ZFXRenderDevice Interface

Direct3D Die umfassendste und komplexeste Komponente in unserer Engine wird natürlich der Renderer, der das Interface `ZFXRenderDevice` implementieren muss. Im folgenden dritten Kapitel werden wir die Arbeit an dieser Komponente beginnen, wobei wir uns zunächst auf die Erstellung und Implementierung eines Interfaces allgemein konzentrieren. Viel mehr, als die Grafikkarte zu initialisieren und den Bildschirm zu löschen, wird das Interface dann noch nicht können. Im sechsten Kapitel machen wir uns dann richtig an die Arbeit, greifen den Renderer wieder auf und formen ihn zu einer wirklich anwendbaren Komponente unserer Engine. Wie bereits des Öfteren erwähnt, verwenden wir natürlich Direct3D für unseren Renderer. Aber alternativ kann man die Implementierung des Interfaces auch durch OpenGL umsetzen, wenn man dies möchte.

ZFXInputDevice Interface

Die Input-Komponente unserer Engine wird einen sehr einfachen Zugriff auf die Tastatur, die Maus und den Joystick bieten. Wir abstrahieren hier so weit von der Lowlevel-Ebene weg, dass der Anwender unserer Engine mit einem Funktionsaufruf prüfen kann, ob eine bestimmte Taste auf irgendeinem Eingabegerät gedrückt wurde, und die Bewegung der Maus und des Joysticks ebenso leicht abfragen kann. Für die Implementierung dieser Komponente verwenden wir DirectInput. Aber auch hier kann man die Implementierung des Interfaces über eine beliebige andere API vornehmen.

DirectInput

ZFXNetworkDevice Interface

Ein Netzwerkmodus ist heutzutage für jedes Computerspiel unverzichtbar. Es muss ja nicht immer gleich ein MMORPG sein. Ein einfacher Deathmatch-Modus ist aber nahezu in jedem 3D-Spiel zu finden. Unsere Engine wäre keine vernünftige Game-Engine, wenn sie nicht auch die Möglichkeit bieten würde, über ein Netzwerk zu kommunizieren. Vor der Programmierung einer Netzwerk-Komponente haben viele Einsteiger immer noch großen Respekt, da die Arbeit mit Netzwerk-Nachrichten, IP-Adressen, Internet-Protokollen und LAN-Verbindungen so unheimlich komplex erscheint. Bei der Implementierung dieser Komponente werden wir aber feststellen, dass die Programmierung für ein Netzwerk eigentlich eine ziemlich triviale Sache ist. Als API dient uns zur Implementierung des Interfaces WinSock. Aber auch hier gilt, dass man eine eigene Implementierung des Interfaces mit einer beliebigen anderen API, beispielsweise Berkeley Sockets oder DirectPlay, erstellen kann.

WinSock

ZFXAudioDevice Interface

Das vierte und letzte Interface unserer Engine ist die Audio-Komponente. Ein Computerspiel ist ganz klar ein interaktives Multimedia-Programm. Das Wort Multimedia beinhaltet die Verbindung von (animierter) Grafik und Audio. Wer jemals ein Spiel ohne Boxen spielen musste, beispielsweise weil der Kumpel an seinem Zweitrechner keine Lautsprecher hatte und man trotzdem etwas im LAN zocken wollte, der wird wissen, um wie viele Potenzen ein Spiel schlechter ist, wenn es einfach keinen Sound bietet. Wir brauchen also auch ein Interface für den Sound, und daher implementieren wir ein solches Interface mit Hilfe von DirectMusic. Natürlich kann man auch hier wieder eine beliebige andere API wählen.

DirectAudio

ZFX3D Bibliothek

SSE und C++

Nun muss ich zum ersten Mal im Verlauf dieses Buches zugeben, wieder einen Namen ziemlich blöd gewählt zu haben. Auch wenn diese Bibliothek den Namen `ZFX3D` trägt, so hat sie nichts mit 3D-Grafik zu tun. Jedenfalls nicht direkt. In diese Bibliothek werden wir etliche Klassen stecken, die Berechnungen im 3D-Raum vereinfachen. Es handelt sich dabei nämlich um eine umfassende Mathe-Bibliothek, die Vektoren, Matrizen usw. beinhaltet. Es handelt sich hierbei jedoch um eine statische Bibliothek und kein Interface mehr, denn wir benötigen keine besondere API, um diese Funktionalität umzusetzen. Wir werden mehr oder weniger reines C++ verwenden und an einigen Stellen auf eine besondere Form des Assemblers zugreifen. Dazu kommen wir aber detaillierter, als es einigen lieb sein wird, im vierten Kapitel.

ZFXGeneral Bibliothek

Allgemeines

Die letzte Komponente unserer Engine ist eine kleine Hilfsbibliothek, die insbesondere einige Klassen für verschiedene Arten von Kameras zur Verfügung stellt. Diese Bibliothek ist nicht direkt ein unverzichtbarer Teil der Engine an sich, aber er erleichtert es einer Applikation ganz ungemein, eine Navigation durch die virtuelle Welt mit wenigen Funktionsaufrufen zu bewerkstelligen. Dabei muss sich die Applikation nicht mehr um Rotationen von lokalen Achsen, das Verhindern eines Gimbal Lock und all solche Sachen kümmern. Als zusätzliche Komponente benötigt diese Bibliothek aber die Mathe-Bibliothek unserer Engine, denn wir müssen dort natürlich mit Vektoren und Quaternions um uns werfen.

2.5 Ein Blick zurück, zwei Schritt nach vorn

Design im Schnelldurchlauf

Nun hat ihr die wichtigsten Dinge gehört bzw. gesehen, die ihr wissen müsst, um dem Buch gut folgen zu können. Dieses Kapitel war sehr knapp gehalten, aber ich denke, es reicht aus, um euch einen Überblick darüber zu geben, was wir auf den folgenden neunhundert Seiten so alles zu tun haben. Ich fand es recht wichtig, dass wir vorher eine Vorstellung davon haben, welche Komponenten zu unserer Engine gehören sollen und wie diese umgesetzt werden. Wenn euch das jetzt immer noch viel vorkommt, und euer Gehirn einen Information-Overload meldet, dann blättert einfach schnell um, lest das folgende Kapitel und kehrt dann noch einmal hierher zurück. Ich bin sicher, dass ihr diese paar Seiten hier dann im Schnelldurchlauf überfliegen und »Ach ja, so ist das. Klar!« sagen werdet. :-)

3 Rahmenanwendung der ZFXEngine

» Was man bekommt, wenn man nicht bekommt, was man will,
ist Erfahrung.« (Désirée Nick)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Was ist ein Interface?
- ➔ Erstellung statischer und dynamischer Bibliotheken
- ➔ Anzeigen von Dialogen zur Interaktion mit Programmen
- ➔ Entwicklung einer Bibliothek als Renderer für die *ZFXEngine*
- ➔ Initialisierung und Starten von Direct3D 9 über eine DLL
- ➔ Schreiben eines Rahmenprogramms für die *ZFXEngine*

3.1 Begriffsbestimmung Interface

Die Bezeichnung »Interface« wird im Bereich der Informatik für eine Vielzahl von Dingen verwendet. So ist ein GUI beispielsweise ein *Graphical User Interface*, also nichts anderes als eine Ansammlung von Buttons, Menüs und anderen Steuerelementen, mit deren Hilfe der Benutzer am Bildschirm das Programm bedienen kann. Die Abkürzung API wiederum steht für *Application Programming Interface*. Damit meint man diejenigen Funktionen, die eine Bibliothek von Funktionen einer anderen Applikation für die Erledigung diverser Aufgaben zur Verfügung stellt. Die DirectGraphics-Komponente von DirectX ist beispielsweise eine solche Grafik-API, über deren Funktionen wir in begrenztem Umfang unsere Grafikkarte ansprechen können, ohne dass wir selbst mit dem Treiber der Grafikkarte kommunizieren müssten.

Daher nennt man ein Interface in der deutschen Sprache »Schnittstelle«. Ein Interface ist immer eine Schnittstelle zwischen zwei Objekten, die auf irgendeine Weise miteinander kommunizieren müssen. Wenn der Benutzer am Bildschirm sein Programm dazu bringen möchte, etwas für ihn zu tun, dann muss er das dem Programm über die GUI tun, beispielsweise durch Anklicken eines bestimmten Buttons. Andersherum muss das Programm,

Interfaces

wenn es dem Benutzer etwas mitteilen möchte, ein Dialog-Fenster öffnen, um so über die GUI mit dem Benutzer zu kommunizieren.

Abstrakte Klassen

Im strengen programmiertechnischen Sinne bezeichnet eine API aber mehr oder weniger lediglich eine abstrakte Klasse einer objektorientierten Programmiersprache. Wir verwenden in diesem Buch die Sprache C++, also orientieren wir die folgenden Erklärungen an diesem konkreten Beispiel. Eine abstrakte Klasse ist eine Klasse, die nicht instanziiert werden kann. Man kann also kein Objekt dieser Klasse erzeugen. Das erreicht man, indem man in einer Klasse eine rein virtuelle Member-Funktion deklariert, selbst wenn die Klasse noch andere, nicht rein virtuelle Methoden beinhaltet.

Hm, so kommen wir irgendwie nicht weiter. Ein Fremdwort folgt hier auf das nächste. Vielleicht sollten wir Schritt für Schritt vorgehen.

**Schlüsselwort
virtual**

Durch das Schlüsselwort `virtual` in C++ können wir Member-Funktionen einer Klasse virtuell machen. Das hat nur dann eine Auswirkung, wenn abgeleitete Klassen existieren, die dieselbe Methode implementieren. Normalerweise entscheidet der Typ des Pointers, über den eine Methode aufgerufen wird, aus welcher Klasse die Methode verwendet wird. So kann man Objekte abgeleiteter Klasse durchaus über einen Pointer vom Typ der Basisklasse ansprechen. In diesem Fall würde jedoch die Methode aus der Basisklasse verwendet, was nicht immer erwünscht sein muss. Ist die Methode in der Basisklasse mit dem Schlüsselwort `virtual` deklariert, so wird die Methode aus der Klasse genommen, von der das Objekt instanziiert wurde, auf das der Pointer zeigt und nicht mehr aus der Klasse des Pointers. So kann man namensgleiche Methoden in der Basisklasse und in davon abgeleiteten Klassen haben, alle Objekte dieser Klasse über einen Pointer vom Typ der Basisklasse ansprechen und dennoch die Methoden aus der jeweils richtigen Klasse aufrufen.

**Rein virtuelle
Member-
Funktionen**

Als rein virtuell bezeichnet man eine Member-Funktion genau dann, wenn sie einerseits mit dem Schlüsselwort `virtual` definiert wurde, aber andererseits in der Basisklasse gar keine eigene Implementierung hat. Die Methode ist also in der Basisklasse nicht definiert, aber deklariert. Um dies dem Compiler anzuzeigen, erhält der Prototyp der Member-Funktion den Zusatz `= 0` in der Deklaration. Eine rein virtuelle Funktionsdeklaration sieht also wie folgt aus:

```
virtual vector3d GetCenterpoint(void) = 0;
```

Sobald eine Klasse eine rein virtuelle Member-Funktion enthält, wird sie als abstrakte Klasse bezeichnet. Von einer solchen abstrakten Klasse können keine Objekte erzeugt werden, da eine abstrakte Klasse eben auch Methoden anbietet, die gar nicht implementiert sind. Man kann aber natürlich einen Pointer vom Typ dieser Klasse erzeugen.

Wozu Interfaces?

Nun stellt sich die Frage nach dem Sinn einer abstrakten Klasse, wenn man ja quasi gar nicht mit ihr arbeiten kann. Interessant wird es dann, wenn

man eine solche abstrakte Klasse als eine Art Interface-Definition versteht. Man kann beispielsweise eine abstrakte Klasse mit diversen Attributen und rein virtuellen Funktionen definieren. Diese abstrakte Klasse ist dann eine zwingende Vorlage, was für Member-Funktionen eine Ableitung dieser Basisklasse mindestens definieren muss. Bleibt die Frage, warum man das überhaupt machen sollte, und nicht gleich die korrekte Klasse implementiert. Nun sind wir beispielsweise bei der verteilten Arbeit in einem Team angelangt. Unser Ziel ist es, ein schönes 3D-Spiel zu programmieren. Dazu arbeitet ein Teil des Teams an dem Game-Code, also an der Ablauflogik des Spiels, und ein anderer Teil des Teams programmiert die 3D-Engine. Das erstgenannte Teilteam muss aber bereits frühzeitig in der Entwicklung wissen, auf welche Funktionen der 3D-Engine es zurückgreifen kann und welche Parameterlisten diese Funktionen verwenden. Also wird zu Beginn der Arbeit ein Interface in Form einer abstrakten Klasse definiert, das exakt vorschreibt, welche Funktionen die 3D-Engine später zwingend bieten muss. Basierend auf dieser Vorgabe, entwickelt das Teilteam der 3D-Engine eine Ableitung der abstrakten Klasse, in der die entsprechenden Member-Funktionen implementiert werden. Das zweite Teilteam kann aber bereits mit Pointern der abstrakten Klasse im Game-Code arbeiten. So ist sichergestellt, dass die beiden separat entwickelten Teilkomponenten des Spiels später problemlos zusammenpassen werden.

3.2 Unser Interface

Jetzt dürftet ihr schon eine ganz gute Vorstellung davon haben, wozu man Interfaces verwendet. Im Verlauf dieses Kapitels werden wir lernen, wie man ein solches Interface erzeugen und eine abgeleitete Klasse definieren kann, die dieses Interface implementiert. Unser Ziel ist es, unsere 3D-Engine so unabhängig wie möglich von einer bestimmten Grafik-API oder API-Version zu machen. Wir werden also ein Interface definieren, das uns sämtliche Funktionalität für die Ausgabe von Grafik zur Verfügung stellt, die wir für unsere Engine brauchen. Von diesem Interface können wir dann beliebig viele Klasse ableiten, die die vorgeschriebenen Funktionen implementieren. Auf diese Weise können wir beispielsweise in einer Klasse einen DirectX-Renderer programmieren, in einer anderen Klasse einen OpenGL-Renderer und in einer dritten Klasse einen Software-Renderer.

*Interface für den
Renderer*

Der große Vorteil ist nun folgender: Wir können die jeweiligen Renderer in einer jeweils eigenen Bibliothek erstellen. Unsere *ZFXEngine* kann nun mittels des Interfaces komplett programmiert werden. Danach können wir beliebig eine der drei oben genannten Bibliotheken in das Projekt einbinden und erhalten somit Zugriff auf einen von drei Renderern. Der Engine selbst ist es vollkommen egal, ob dort DirectX, OpenGL oder eine eigene Software-Implementierung verwendet wird. Und wir werden noch einen Schritt weiter gehen: Die Implementierung unserer Renderer werden wir zu einer

*Dynamische
Auswahl eines
Renderers*

DLL (Dynamic Link Library) kompilieren. Haben wir schließlich die *ZFX-Engine* oder ein ganzes Spiel unter Verwendung der Engine fertig gestellt, können wir es ruhigen Gewissens kompilieren. Die Render-DLL wird ja erst zur Laufzeit des Programms geladen. Ohne die Engine oder das Spiel neu kompilieren zu müssen, können wir einfach die Render-DLL(s) auswechseln und durch neue Implementierungen ersetzen. Die Engine bzw. das Spiel bleibt weiterhin ausführbar, nutzt jedoch sofort die neue Implementierung aus – und das, ohne dass auch nur eine einzige Änderung am Engine- oder Game-Code notwendig ist.

Analog kann man diese Technik auch für jede andere Komponente eines Projekts durchführen. Man könnte also die Musik- und Soundausgabe in einer DLL kapseln, ebenso wie die Inputverarbeitung oder gar die künstliche Intelligenz. Das macht aber relativ wenig Sinn, denn diese Komponenten sind nicht derartigen Veränderungen unterworfen, wie der Renderer das sein wird. Beim Erscheinen einer neuen DirectX-Version oder einer neuen OpenGL-Spezifikation kann es durchaus sehr viel Sinn machen, den Renderer eines laufenden Projekts neu zu schreiben. Zum Beispiel kann so eine neue Version zusätzliche Vertex- und Pixel-Shader bieten oder einfach nur Optimierungen enthalten. Ebenso ist es denkbar, dass man zunächst eine Art Prototyp-Renderer entwickelt, wobei die kurze Entwicklungszeit das wichtigste Kriterium ist. Mit diesem Prototyp kann man die Engine oder ein konkretes Spiel dann bereits in der frühen Entwicklungsphase am Bildschirm testen. Zeitgleich wird dann ein optimierter Renderer entwickelt, der später lediglich die DLL des Prototyps ersetzt.

*Selbst definierte
Renderer für die
ZFXEngine*

Für dieses Buch werde ich lediglich eine DLL verwenden, die die Render-Funktionalität mit Hilfe von DirectX implementiert. Die Schritte zur Erstellung einer solchen DLL laufen allerdings immer ganz analog ab. Wer sich einen eigenen Renderer unter Verwendung von OpenGL oder eigener Software-Routinen erstellen will, der muss lediglich die entsprechenden Funktionen in einer eigenen DLL implementieren und kann dann die komplette *ZFXEngine* unverändert mit seinem Renderer betreiben. So kann man natürlich auch die Performance verschiedener Implementierungen direkt vergleichen, indem man ein konkretes Projekt mit verschiedenen Render-DLLs unter identischen Bedingungen (z.B. auf derselben Hardware) testet.

Wichtig ist dabei jedoch eines: Ein solcher selbst definierter Renderer, also die konkrete Implementierung nach einem vorgegebenen Interface einer abstrakten Klasse, darf als öffentliche Methoden lediglich die rein virtuellen Member-Funktionen der abstrakten Basisklasse definieren. Lediglich als `private` oder `protected` deklarierte Member-Funktionen kann die abgeleitete Klasse neu einführen und verwenden, ebenso wie zusätzliche Member-Variablen (Attribute). Zusätzliche öffentliche Funktionen in den abgeleiteten Klassen wären ja nur zugänglich, wenn man einen Pointer des Typs der abgeleiteten Klasse verwenden würde. Sinn und Zweck der ganzen Angele-

genheit ist es aber, in der Engine einen Pointer vom Typ der abstrakten Basisklasse zu verwenden, unabhängig davon, welche Ableitung (DirectX oder OpenGL usw.) man als DLL in sein Projekt lädt und verwendet.

3.3 Der Arbeitsbereich für unsere Implementierung

Nach der langen Vorrede ist es nun so weit. Jetzt können wir uns an die Tasten machen und einen Renderer in einer DLL programmieren. Dafür gibt es im Grunde zwei Möglichkeiten: Wir können die DLL direkt in unserem Projekt verlinken, oder wir laden sie per Funktionsaufruf. Die erste Variante ist sicherlich die bequemere. Sie erfordert jedoch, dass unser Projekt zu der statischen Bibliothek gelinkt wird, die beim Erstellen der DLL erzeugt wird.

Statische und dynamische Bibliotheken

Eine Bibliothek ist ein Programm, das lediglich Klassen und Funktionen beinhaltet, die ein anderes Programm verwenden kann. Im Gegensatz zu einem ausführbaren Programm enthält eine Bibliothek keine Startfunktion und kann daher nicht autark ausgeführt werden.

Kompiliert man sein Projekt zu einer Bibliothek, so wird daraus wahlweise eine Datei mit der Endung `*.lib` (Static Library, statische Bibliothek) oder `*.dll` (Dynamic Link Library, dynamische Bibliothek) erzeugt. Diese Datei kann man dann in anderen Projekten verlinken, um die Klassen und Funktionen aus der Bibliothek verwenden zu können.

Eine statische Bibliothek wird dabei beim Kompilieren des ausführbaren Projekts, das die Bibliothek verwendet, in dessen kompilierte, ausführbare Datei mit eingebunden. Sie ist damit in der Release-Version direkt mit enthalten. Eine DLL hingegen wird nicht in die ausführbare Datei eingebunden. Sie wird vom Programm erst zur Laufzeit geladen und muss daher zusammen mit dem Programm ausgeliefert werden.

Da die fertig kompilierte DLL nicht direkt in ein ausführbares Projekt eingebunden wird, *weiß* dieses Programm auch nicht, welche Klassen und Funktionen die DLL enthält. Beim Kompilieren einer DLL wird daher auch immer eine namensgleiche Datei `*.lib` mit erzeugt. Diese Datei enthält quasi eine Aufstellung all der Klassen und Funktionen, die die DLL enthält. Möchte ein Projekt eine DLL verwenden, so muss das Projekt (nach der oben erstgenannten Methode) nicht zu der DLL gelinkt werden, sondern zu der zugehörigen statischen Bibliothek. Damit entfällt aber der Vorteil einer DLL, weil ein ausführbares Projekt eben doch neu kompiliert werden muss, wenn sich an der DLL etwas geändert hat. Die zugehörige statische Bibliothek muss ja neu in die ausführbare Datei eingebunden werden. Zudem werden auf diese Weise beim Start des Programms sofort alle DLLs geladen, mit deren `*.lib`-Dateien das Programm verlinkt ist. Hierbei verschwendet man unnötig Speicher, wenn man nicht wirklich alle diese DLLs benötigt.



*Aus diesem Grunde muss man bei der Programmierung eines DirectX-Projekts beispielsweise immer zu diversen *.lib-Dateien linken, obwohl die eigentliche Implementierung der DirectX-Komponenten als DLLs vorliegt.*

Dieses Problem der DLLs kann man jedoch umgehen. Es ist möglich, eine DLL erst zur Laufzeit eines Programms an einer beliebigen Stelle im Ablauf des Programms gezielt zu laden, ohne dessen *.lib-Datei verlinkt zu haben. Wir schauen uns gleich an, wie das funktioniert. Wichtig dabei ist, dass wir unserem Projekt dann noch irgendwie anders mitteilen müssen, welche Klassen und Funktionen unsere DLL enthält. Dazu kann man den Funktionen den Zusatz `__declspec(dllexport)` mit auf den Weg geben. Eine andere Alternative ist, dass man eine Datei mit der Endung *.def erstellt, die die exportierten Funktionen der DLL auflistet. Streng genommen ist es (mit beiden Varianten) unmöglich, eine Klasse aus einer DLL zu exportieren. Man ist hier auf Funktionen im C-Stil beschränkt. Diese Beschränkung kann man aber umgehen, indem man ein Interface verwendet. Das werden wir gleich noch im Quelltext sehen, aber ich werde es hier schon einmal allgemein beschreiben.

Klassen aus einer DLL verwenden

Wir definieren ein Interface, also eine abstrakte Klasse. Dieses Interface binden wir in das ausführbare Projekt, beispielsweise in Form einer Header-Datei, ein. In dem DLL-Projekt erzeugen wir eine Klasse, die von diesem Interface abgeleitet ist und dessen Funktionen implementiert. Nun schreiben wir eine reine C-Funktion für die DLL, die ein Objekt dieser Klasse instanziert, eine Typumwandlung dieses Objekts in den Typ der abstrakten Klasse des Interfaces durchführt und das gecastete Objekt an den Aufrufer zurückgibt. Diese C-Funktion ist nun das Einzige, was wir von der DLL exportieren müssen. Über sie erhalten wir in unserem ausführbaren Projekt einen Pointer vom Typ des Interfaces, der aber auf ein Objekt der Klasse aus der DLL zeigt. Wir wissen aber, dass dieses Objekt alle Funktionen implementiert, die das Interface vorschreibt. Und genau diese Funktionen können wir aufrufen.

Das klingt vielleicht für den Anfang ein wenig kompliziert, aber wenn wir uns später das gesamte Projekt ansehen, dann wird die Sache schnell klar. Um es aber nicht ganz so einfach zu machen, werden wir hier noch eine statische Bibliothek zusätzlich verwenden. Die ganze Aufruferei von Funktionen für das Laden und Verwenden von DLLs zur Laufzeit ist nicht besonders schön. Vor allem ist es unkomfortabel für die Benutzer unserer DLL. Also schreiben wir den gesamten Code, der das Laden der DLL betrifft, in eine statische Bibliothek. Diese wird zwar fest in das ausführbare Projekt eingebunden, das unsere DLL verwenden möchte, wird sich allerdings nie mehr ändern müssen, selbst wenn sich unsere DLL verändert. Diese statische Bibliothek beinhaltet also nur eine Klasse, die das Handling der DLLs managt. Sie wird auch nicht sonderlich umfangreich sein, aber dennoch eine Menge unansehnlichen Codes vor dem Anwender unserer DLL verstecken. Zusätzlich schreiben wir dann, wie geplant, eine eigene DLL für jeden Renderer, den wir verwenden möchten.

Im Folgenden werde ich die notwendigen Schritte noch einmal explizit beschreiben, wenn sie notwendig werden. Aber es kann ja nicht schaden, hier schon einmal einen groben Überblick über das zu bekommen, was wir gleich tun werden. Als Erstes benötigen wir einen *Visual C++*-Arbeitsbereich. Zu Beginn erzeugen wir ein Projekt für eine statische Bibliothek. Diese wird unser Manager für das DLL-Handling. Dann fügen wir in diesen Arbeitsbereich ein neues Projekt ein – diesmal jedoch ein Projekt für eine dynamische Bibliothek. Diese wird unseren DirectX-Renderer enthalten, also die Implementierung unseres Interfaces. Für jeden weiteren Renderer müsste man in dem Arbeitsbereich ein weiteres neues DLL-Projekt einfügen. Sehen wir uns das nun etwas genauer an.¹

ZFXRenderer, eine statische Bibliothek als Manager

Bisher haben wir ja immer von dem Renderer an sich gesprochen. Diese Namensgebung werden wir jetzt ein wenig korrigieren. Dabei orientieren wir uns ein wenig an der Namensgebung von DirectX. Die statische Bibliothek, die das Laden der DLL erledigt, nennen wir **ZFXRenderer**. Dieser Renderer hat nur zwei Aufgaben. Zum einen soll er die DLL laden, die der Benutzer unserer *ZFXEngine* verwenden möchte, also die DirectX-Version oder die OpenGL-Version oder eben jede andere Variante eines Renderers, die wir in einer DLL implementiert haben. Zum anderen benötigen wir den ZFXRenderer dann auch noch, um beim Beenden des Programms die verwendeten Ressourcen wieder freizugeben.

Der Benutzer unserer Engine muss sich mit dem ZFXRenderer entsprechend nur zweimal auseinander setzen. Bei der Initialisierung des Programms muss er ein Objekt dieser Klasse instanziiieren und sich von diesem ein Device-Objekt übergeben lassen. Beim Beenden des Programms wird der Renderer einfach wieder per `delete`-Aufruf über den Destruktor gelöscht. Bei dem Device-Objekt handelt es sich um einen Zeiger auf eine Instanz der Klasse aus der DLL. Diese Klasse nennen wir, analog zur DirectX-Namensgebung, **ZFXRenderDevice**. Dies ist also die Bezeichnung, die unser Interface, also die abstrakte Klasse, erhält. Die tatsächliche Klasse in einer DLL erhält einen anderen Namen, wird jedoch, wie oben besprochen, von der abstrakten Klasse ZFXRenderDevice abgeleitet und implementiert deren rein virtuelle Funktionen.

Jetzt machen wir uns an die Arbeit. Als Erstes starten wir *Visual C++* und wählen aus dem Menü DATEI den Befehl NEU ... aus. In dem nun erscheinenden Dialog wählen wir die Registerkarte PROJEKTE aus und klicken in der Auswahlliste der verschiedenen Projekttypen einmal auf *Win-32 Bibliothek (statische)*. Auf der rechten Seite wählen wir noch einen entsprechen-

¹ Der folgende Quellcode basiert auf dem hervorragenden Tutorial *Striving for Graphics API Independence* auf GameDev.net von Erik Yuzwa, www.gamedev.net/reference/articles/article1672.asp.

den Pfad aus, in dem wir unseren Arbeitsbereich anlegen wollen. Im Feld PROJEKTNAME geben wir die Bezeichnung *ZFXRenderer* ein. Ein Klick auf den Button OK führt zu einem weiteren Dialog-Fenster von *Visual C++*. Hier können wir über diverse Checkboxes weitere Optionen für das Projekt festlegen. Wir achten darauf, dass keine dieser Checkboxes angewählt ist, erstellen den Arbeitsbereich nun endgültig mit einem Klick auf den Button FERTIGSTELLEN und bestätigen noch einmal per Mausklick auf die Schaltfläche OK im neu auftauchenden Dialog.

Einfügen der Dateien

Jetzt haben wir den Arbeitsbereich vor uns. Dieser ist jedoch gähnend leer. Also wählen wir aus dem Menü PROJEKT den Menüpunkt DEM PROJEKT HINZUFÜGEN und dort die Option NEU aus dem Untermenü. Jetzt erscheint der Dialog für das Einfügen von Objekten in den Arbeitsbereich. Dort wählen wir die Registerkarte DATEIEN aus. Aus der Liste suchen wir den Eintrag *C/C++-Header-Datei* beziehungsweise *C++-Quellcodedatei* und geben auf der rechten Seite im Dialog den Namen für die neue Datei an. Auf diese Weise fügen wir dem Projekt die folgenden neuen drei Dateien hinzu:

- ➔ ZFXRenderer.cpp
- ➔ ZFXRenderer.h
- ➔ ZFXRenderDevice.h

Die ersten beiden Dateien sind für die Implementierung der Klasse ZFXRenderer gedacht. Die letztgenannte Datei ist die Definition des Interfaces, wird also eine abstrakte Klasse enthalten, von der sich dann jeweils die Klassen der DLLs ableiten. Die eigentliche Implementierung der Funktionen und Klassen in den Dateien schauen wir uns gleich an. Zunächst erweitern wir unseren Arbeitsbereich noch ein wenig, um auch eine DLL verwenden zu können. Im Anschluss definieren wir noch das Interface, damit wir überhaupt wissen, was wir eigentlich implementieren müssen.

ZFXD3D, eine dynamische Bibliothek als Render-Device

Mehrere Projekte in einem Arbeitsbereich

In unserem eben erzeugten Arbeitsbereich haben wir nun ein Projekt namens ZFXRenderer. Viele werden bisher ihre Arbeitsbereiche auch immer nur so verwendet haben, dass sie ein einziges Projekt darin verwendeten. Ein Arbeitsbereich in *Visual C++* kann aber beliebig viele Projekte enthalten. Insbesondere bei logisch zusammenhängenden Projekten von Bibliotheken (statischen und dynamischen) macht es Sinn, diese in einem Arbeitsbereich zu halten. Das Projekt für die statische Bibliothek ist ja die Basis, über die wir auf unsere DLLs zugreifen werden. Also fügen wir in denselben Arbeitsbereich für jede DLL, die wir zum Rendern erstellen, ein eigenes Projekt ein. Im Verlauf dieses Buches arbeiten wir nur mit einer DLL, und zwar auf Basis der Direct3D-Komponente von DirectX. Für jede eigene Kapselung einer anderen API (OpenGL, Software-Renderer usw.) erzeugt man einfach analog ein eigenes DLL-Projekt in diesem Arbeitsbereich.

Wir gehen also wieder genauso vor wie eben bei dem Einfügen von Dateien in das Projekt ZFXRenderer. Diesmal wählen wir jedoch nicht die Registerkarte DATEIEN, sondern die Registerkarte PROJEKTE. Hier wählen wir den Eintrag *Win32 Dynamic-Link Library*. Bei der Pfadangabe auf der rechten Seite hängen wir noch ein Unterverzeichnis namens *ZFXD3D* an den Pfad des bisherigen Projekts an. Auch als Namen des Projekts geben wir *ZFXD3D* an. Ein Klick auf den OK-Button zeigt einen neuen Dialog an, in dem wir einfach den Eintrag *Ein leeres DLL-Projekt* wählen und wieder durch OK bestätigen. Den daraufhin auftauchenden Informationsdialog klicken wir mit OK schnell weg. Nun haben wir ein zweites Projekt, diesmal eine DLL, in unserem Arbeitsbereich.

Aktives Projekt im Arbeitsbereich festlegen

Wenn man in einem Arbeitsbereich von Visual C++ mehrere verschiedene Projekte angelegt hat, dann muss man immer darauf achten, welches davon gerade aktiv ist. In der Dateiübersicht von Visual C++ erkennt man das daran, dass der Name des aktiven Projekts in einer fett formatierten Schrift dargestellt wird. Alle projektbezogenen Aktionen von Visual C++ beziehen sich auf dieses eine aktive Projekt, insbesondere das Einfügen von Dateien und auch das Kompilieren.

Welches Projekt gerade aktiv ist, kann man über das Menü PROJEKT festlegen. Dort gibt es den Menüpunkt AKTIVES PROJEKT FESTLEGEN, wo man alle Projekte des Arbeitsbereichs in einer Auswahlliste zur Verfügung hat. Sobald man Änderungen an einem Projekt vorgenommen hat, sollte man darauf achten, dass man auch das entsprechende Projekt als aktives Projekt festgelegt hat, bevor man die Dateien kompiliert.

Da wir es ja nicht anders wollten, enthält das DLL-Projekt noch keine Dateien. Aber das ändern wir nun sofort. Analog zu den Arbeitsschritten beim Einfügen von Dateien in das Projekt der statischen Bibliothek fügen wir nun die folgenden Dateien in das Projekt der DLL ein:

- ➔ ZFXD3D_init.cpp
- ➔ ZFXD3D_enum.cpp
- ➔ ZFXD3D_main.cpp
- ➔ ZFXD3D.h
- ➔ ZFXD3D.def

Die ersten vier Dateien dienen zur Implementierung der Klasse ZFXD3D. Dabei wird es sich um eine von dem Interface abgeleitete Klasse handeln, in der wir Direct3D als Grafik-API verwenden. Zur besseren Übersicht werden wir jedoch die Funktionen der Klasse nicht in eine einzige Datei quetschen, sondern ein wenig trennen. Die Funktionen für die Initialisierung und das

Beenden von Direct3D schreiben wir in die Datei mit dem Suffix `_init`. Die Enumeration vorhandener Grafikhardware ist aber, wie man es von DirectX gewohnt ist, auch recht komplex, daher kapseln wir diese Funktionen in einer eigenen Klasse in der Datei mit dem Suffix `_enum`. Die eigentliche Funktionalität steckt dann in der Datei mit dem Suffix `_main`. Eventuell kommen bei Bedarf später noch andere Dateien hinzu, aber jetzt kommen wir erst mal damit aus. Die letztgenannte Datei, `ZFXD3D.def`, gibt an, welche Funktionen die DLL exportiert und für externe Benutzer zugänglich macht.

Nun haben wir unseren Arbeitsbereich so eingerichtet, dass wir mit der Arbeit beginnen können. Um uns noch einmal zu vergewissern, was wir hier tun, sollten wir erneut einen Blick auf den Arbeitsbereich werfen. Wir haben dort eine statische Bibliothek, die die Auswahl und Verwendung einer DLL für ein anderes Projekt ermöglichen wird. Dann haben wir im Arbeitsbereich für jede Grafik-API, die wir unterstützen wollen (bisher nur Direct3D), ein eigenes DLL-Projekt.



*Per Voreinstellung erzeugt Visual C++ die kompilierten Dateien, in diesem Fall die *.lib- und *.dll-Dateien, in den Ordnern Debug bzw. Release – je nach gewählter Kompilierungsart. Im Menü PROJEKT unter dem Menüpunkt EINSTELLUNGEN ... kann man auf der Registerkarte ALLGEMEIN diese Verzeichnisse bzw. den ganzen Pfad ändern. So kann man beispielsweise einen Ordner auf der Festplatte erstellen, in den man alle Bibliotheken einer Engine hineinkompiliert.*

Fügt man diesen Verzeichnispfad in Visual C++ für Bibliotheken-Verzeichnisse hinzu, so kann man andere Projekte zu seinen Bibliotheken linken, ohne nach dem Ändern der Bibliotheken an den anderen Projekten etwas ändern zu müssen, was nötig wäre, wenn wir die kompilierten Bibliotheken einfach nur in das Verzeichnis dieser Projekte kopieren würden.

Bevor wir uns daran machen, unsere Bibliotheken zu implementieren, müssen wir uns noch ein paar Gedanken über das Interface machen.

ZFXRenderDevice, ein Interface als abstrakte Klasse

Sämtliche Funktionen, die wiederum Methoden einer spezifischen API verwenden, müssen wir für die *ZFXEngine* in einer DLL kapseln. Nur so erreichen wir wirkliche Unabhängigkeit von einer bestimmten API oder API-Version. Dazu ist es aber nötig, dass wir uns zunächst überlegen, welche Funktionen wir überhaupt brauchen. Aus diesen Überlegungen heraus konstruieren wir dann ein Interface, also eine abstrakte C++-Klasse, an die sich unsere DLLs sozusagen *halten* müssen.

Für dieses Kapitel werden wir das Interface sehr übersichtlich halten. Hier geht es schließlich um die Technik, wie wir unsere *ZFXEngine* unabhängig von einer Grafik-API halten können, indem wir die entsprechenden Funk-

tionen in einer beliebig austauschbaren DLL kapseln. Am Ende dieses Kapitels wird unsere DLL nicht viel mehr können, als den Bildschirm in einer beliebigen Farbe löschen zu können – das dafür aber sehr komfortabel. Intern wird in der DLL aber wesentlich mehr stecken. Insbesondere soll die DLL dazu in der Lage sein, die vorhandene Grafikkhardware zu erkennen und ihre Eigenschaften abzufragen. Per Auswahldialog kann der Benutzer dann einen verfügbaren Modus (Bildschirmauflösung, Farbtiefe usw.) wählen und die Engine starten.

Schauen wir uns erst einmal die Definition des Interfaces an:

```
// in der Datei: ZFXRenderDevice.h
#define MAX_3DHWND 8

class ZFXRenderDevice {
protected:
    HWND      m_hWndMain;           // Hauptfenster
    HWND      m_hWnd[MAX_3DHWND];  // 3D-Fenster
    UINT      m_nNumhWnd;           // Anzahl Fenster
    UINT      m_nActivehWnd;        // aktives Fenster
    HINSTANCE m_hDLL;               // DLL-Modul
    DWORD     m_dwWidth;            // Screen-Breite
    DWORD     m_dwHeight;           // ScreenHöhe
    bool      m_bWindowed;          // Windowed Mode?
    char      m_chAdapter[256];     // Name der GaKa
    FILE      *m_pLog;              // Logfile
    bool      m_bRunning;

public:
    ZFXRenderDevice(void) {};
    virtual ~ZFXRenderDevice(void) {};

    // INIT/RELEASE STUFF:
    // =====
    virtual HRESULT Init(HWND, const HWND*, int,
                        int, int, bool)=0;
    virtual void Release(void) =0;
    virtual bool IsRunning(void) =0;

    // RENDERING STUFF:
    // =====
    virtual HRESULT UseWindow(UINT nHwnd)=0;
    virtual HRESULT BeginRendering(bool bClearPixel,
                                   bool bClearDepth,
                                   bool bClearStencil)
        =0;
    virtual void EndRendering(void)=0;
    virtual HRESULT Clear(bool bClearPixel,
                          bool bClearDepth,
```



```

bool bClearStencil)=0;

virtual void SetClearColor(float fRed,
                           float fGreen,
                           float fBlue)=0;

}; // class

typedef struct ZFXRenderDevice *LPZFXRENDERDEVICE;

```

Virtuelle Destruktoren

An Attributen definiert das Interface schon einen ganzen Satz wichtiger Variablen, insbesondere die Höhe, Breite und Farbtiefe der Auflösung. Diese Attribute stehen den abgeleiteten Klassen natürlich auch zur Verfügung. Interessanter sind hier aber die rein virtuellen Funktionen. Als Erstes beachte man, dass auch der Destruktor der Klasse als virtuell definiert ist. Auf diese Weise verhindern wir, dass wir das Aufrufen des Destruktors verpassen. Normalerweise entscheidet ja der Typ des Pointers auf ein Objekt, von welcher Klasse der Destruktor verwendet wird. Später arbeiten wir aber mit Pointern vom Typ der Klasse ZFXRenderDevice. Diese zeigen allerdings auf Objekte vom Typ der abgeleiteten Klasse ZFXD3D. Definieren wir nun auch den Destruktor virtuell, so entscheidet der Typ des Objekts, von welcher Klasse wir den Destruktor verwenden.

Dann folgen ein paar Funktionen für die Initialisierung, die Freigabe und die Prüfung auf erfolgreiche Initialisierung der Klasse. Auch für das Rendern durch unsere DLL haben wir bisher nur vier Funktionen. Die Funktion `ZFXRenderDevice::BeginnRendering` dient dazu, den Render-Vorgang in einem Frame zu starten. Über die Parameterliste der Funktion wird gesteuert, ob und welche Buffer (Pixel-Buffer, Depth-Buffer, Stencil-Buffer) dabei gelöscht werden sollen. Dazu haben wir noch eine weitere Funktion, mit der lediglich der oder die angegebenen Buffer gelöscht werden, ohne jedoch die Szene zu starten. Dies kann hilfreich sein, wenn man mitten in einer Szene beispielsweise den Depth-Buffer löschen muss. An grafischen Funktionen ist im Interface bisher nur eine Methode vorgesehen, mit der man die Hintergrundfarbe ändern kann, mit der der Pixel-Buffer gelöscht wird. In den späteren Kapiteln werden wir die Funktionalität des Interfaces natürlich noch um einiges erweitern. Für dieses Kapitel reichen diese Funktionen aber aus.

Zu guter Letzt definieren wir noch die Struktur `LPZFXRENDERDEVICE`, die nichts weiter als ein Pointer auf eine Instanz der Klasse ist.

3.4 Implementierung der statischen Bibliothek

Der Ankerpunkt für die Arbeit mit unseren DLLs zum Rendern ist die statische Bibliothek ZFXRenderer. Ihre Aufgabe ist es zu entscheiden, welche DLL geladen wird. Eine DLL repräsentiert dabei ein Render-Device, über das dann tatsächlich Grafik ausgegeben werden kann. Die Implementierung

der statischen Bibliothek ist recht kurz und bündig und wird im Verlauf des Buches nicht mehr wirklich geändert. Neu kompilieren muss man sie nur in zwei Fällen: Zum einen natürlich dann, wenn man an ihren Funktionen doch noch etwas ändert. Das ist immer dann der Fall, wenn man beispielsweise eine komplett neue DLL unterstützen möchte. Dann muss man ja den Namen der DLL angeben und die Bedingung, unter der diese spezifische DLL geladen werden soll. Zum anderen müssen wir die statische Bibliothek neu kompilieren, wenn wir an unserem Interface `ZFXRenderDevice` etwas ändern, da die Bibliothek den Header `ZFXRenderDevice.h` auch mit einbindet.

Wo wir gerade bei Headern sind, schauen wir uns den Header gleich einmal an, in dem die Definition der Klasse `ZFXRenderer` steht.

```
// in der Datei: ZFXRenderer.h
#include "ZFXRenderDevice.h"

class ZFXRenderer {
public:
    ZFXRenderer(HINSTANCE hInst);
    ~ZFXRenderer(void);

    HRESULT          CreateDevice(char *chAPI);
    void             Release(void);
    LPZFXRENDERDEVICE GetDevice(void)
    { return m_pDevice; }

    HINSTANCE        GetModule(void)
    { return m_hDLL;   }

private:
    ZFXRenderDevice *m_pDevice;
    HINSTANCE        m_hInst;
    HMODULE          m_hDLL;
}; // class

typedef struct ZFXRenderer *LPZFXRENDERER;
```



Wie versprochen, ist die Klasse nicht sehr umfangreich. Dem Konstruktor der Klasse geben wir den Instanzzähler des Windows-Programms an, das die Klasse später verwendet. Diesen Zähler speichern wir in einem Attribut der Klasse, falls wir ihn später einmal brauchen sollten. Viel wichtiger ist das Attribut `m_hDLL`, das später auf den Instanzzähler der geladenen DLL hinweisen wird. Das Attribut `m_pDevice` ist dann endlich der ersehnte Pointer auf die von dem Interface abgeleitete Klasse `ZFXD3D` aus der DLL. Das zugehörige Objekt erstellen wir mit der Methode `ZFXRenderer::CreateDevice`.

Aber schauen wir uns erst mal den Konstruktor und den Destruktor an. In diesen werden lediglich die Attribute initialisiert bzw. die `Release()`-Methode beim Löschen des Objekts aufgerufen.



```
ZFXRenderer::ZFXRenderer(HINSTANCE hInst) {
    m_hInst = hInst;
    m_hDLL = NULL;
    m_pDevice = NULL;
}

ZFXRenderer::~ZFXRenderer(void) {
    Release();
}
```

Gehen wir also gleich zu den interessanten Dingen dieser Manager-Klasse. Und das ist definitiv die Erstellung eines Device-Objekts in der entsprechenden Funktion. Dafür benötigen wir aber noch schnell die folgende Definition in der Header-Datei des Interfaces:



```
// in der Datei: ZFXRenderDevice.h
extern "C" {
    HRESULT CreateRenderDevice(HINSTANCE hDLL,
                              ZFXRenderDevice **pInterface);
    typedef HRESULT (*CREATERENDERDEVICE)(HINSTANCE hDLL,
                                           ZFXRenderDevice **pInterface);

    HRESULT ReleaseRenderDevice(ZFXRenderDevice
                                **pInterface);
    typedef HRESULT (*RELEASESERENDERDEVICE)(ZFXRenderDevice
                                              **pInterface);
}
```

**Export von
Klassen aus einer
DLL**

Auf diese Weise definieren wir die Symbole `CREATERENDERDEVICE` und `RELEASESERENDERDEVICE` für die entsprechenden Funktionen `CreateRenderDevice()` und `ReleaseRenderDevice()` im C-Stil, die die DLL später bereitstellen muss. Und das sind nämlich genau die Funktionen, die als einzige von unseren DLLs exportiert werden. Weiter oben hatten wir ja schon besprochen, dass wir aus einer DLL keine Klassen exportieren können. Daher verwenden wir diese Funktionen, um die DLL einen Pointer vom Typ des Interfaces `ZFXRenderDevice` auf ein Objekt ihrer eigenen Klasse, beispielsweise `ZFXD3D`, setzen zu lassen. Diese Klasse ist ja von der Klasse `ZFXRenderDevice` abgeleitet. So erhalten wir dann doch Zugriff auf die Klasse aus der DLL. Entsprechend muss jede DLL, die hier als Render-Device verwendet werden soll, diese beiden Funktionen definieren und exportieren. Aber so weit sind wir ja noch nicht.

Nun schauen wir uns an, wie die Klasse `ZFXRenderer` ein Render-Device erzeugen kann. Die entsprechende Funktion nimmt als Parameter einen String auf, der den Namen einer API angeben soll. Im Verlauf dieses Buches entwickeln wir ja nur eine DLL für Direct3D, also akzeptiert diese Funktion bisher auch nur den String »Direct3D«. Enthält der Parameter etwas anderes, bricht die Funktion mit einer Fehlermeldung ab, weil sie ja weiß, dass sie nur die DLL für Direct3D zur Auswahl hat.



```

HRESULT ZFXRenderer::CreateDevice(char *chAPI) {
    char buffer[300];

    if (strcmp(chAPI, "Direct3D") == 0) {
        m_hDLL = LoadLibraryEx("ZFXD3D.dll", NULL, 0);
        if (!m_hDLL) {
            MessageBox(NULL,
                "Loading ZFXD3D.dll from lib failed.",
                "ZFXEngine - error", MB_OK | MB_ICONERROR);
            return E_FAIL;
        }
    }
    else {
        sprintf(buffer, "API '%s' not supported.", chAPI);
        MessageBox(NULL, buffer, "ZFXEngine - error",
            MB_OK | MB_ICONERROR);
        return E_FAIL;
    }

    CREATERENDERDEVICE _CreateRenderDevice = 0;
    HRESULT hr;

    // Zeiger auf die dll Funktion 'CreateRenderDevice'
    _CreateRenderDevice = (CREATERENDERDEVICE)
        GetProcAddress(m_hDLL,
            "CreateRenderDevice");

    // aufruf der dll Create-Funktion
    hr = _CreateRenderDevice(m_hDLL, &m_pDevice);
    if (FAILED(hr)) {
        MessageBox(NULL,
            "CreateRenderDevice() from lib failed.",
            "ZFXEngine - error", MB_OK | MB_ICONERROR);
        m_pDevice = NULL;
        return E_FAIL;
    }

    return S_OK;
} // CreateDevice

```

Wenn die Funktion als Parameter einen String übergeben bekommt, mit dem sie etwas anfangen kann, dann macht sie sich an die Arbeit. Im ersten Schritt versucht die Funktion, die entsprechende DLL zu laden. Im dem Fall, in dem Direct3D verwendet werden soll, ist das die DLL ZFXD3D.dll, für die wir ja bereits ein Projekt angelegt haben. Dazu stellt die WinAPI die folgende Funktion zur Verfügung:

```

HINSTANCE LoadLibraryEx(LPCTSTR lpLibFileName,
    HANDLE hFile,
    DWORD dwFlags);

```

Für den ersten Parameter dieser Funktion müssen wir den Namen der zu ladenden DLL als String angeben. Der zweite Parameter dieser Funktion ist für interne Zwecke reserviert und muss `NULL` sein. Im dritten Parameter können diverse Flags angegeben werden, das ist aber in unserem Fall unnötig. Daher setzen wir diesen Wert auf 0. Der Rückgabewert dieser Funktion hingegen ist sehr wichtig. Er ist das Handle von Windows auf die geladene DLL. Dieses brauchen wir beispielsweise dann, wenn wir in der DLL mit weiteren Funktionen der WinAPI arbeiten wollen, die als Parameter den `HINSTANCE`-Wert verlangen.

Funktionen in der DLL finden

War das Laden der DLL erfolgreich, dann möchten wir nun gern einen Pointer auf ein Objekt von der Klasse in der DLL. Dazu soll die DLL ja die Funktion `CreateRenderDevice()` exportieren. Anders als bei einer statischen Bibliothek können wir diese Funktion nicht direkt aufrufen, weil die dynamische Bibliothek eben nicht in das Projekt mit hineinkompiliert wird. Der Compiler weiß also zur Kompilierungszeit gar nicht, an welcher Stelle sich diese Funktion später, nach dem Laden der DLL zur Laufzeit, befinden wird. Glücklicherweise gibt es aber eine Funktion der WinAPI, mit der wir zur Laufzeit die Adresse einer Funktion auffinden können:

```
FARPROC GetProcAddress(HMODULE hModule,
                       LPCTSTR lpProcName);
```

Als ersten Parameter müssen wir dieser Funktion das Handle auf die geladene DLL angeben, also genau den Wert, den die Funktion `LoadLibraryEx()` zurückgeliefert hat. Im zweiten Parameter geben wir dann den Namen der Funktion an, die wir in der DLL suchen. Der Rückgabewert der Funktion ist dann die Adresse der Funktion, sofern sie gefunden wurde. Über diese Adresse, die wir im Pointer mit der Bezeichnung `_CreateRenderDevice` speichern, können wir nun die Funktion in der DLL aufrufen. Damit haben wir, sofern die Funktion keinen Fehler liefert, einen gültigen Pointer auf ein Objekt der Klasse aus der DLL. Keine Panik, diese DLL sehen wir uns auch gleich noch im Detail an, um zu sehen, was die exportierten Funktionen, die wir hier verwenden, überhaupt machen.

Ganz genauso arbeitet auch die Funktion für die Freigabe des Render-Device-Objekts. Dafür verwenden wir nur die zweite Funktion, die die DLL exportiert.



```
void ZFXRenderer::Release(void) {
    RELEASERENDERDEVICE _ReleaseRenderDevice = 0;
    HRESULT hr;

    if (m_hDLL) {
        // Zeiger auf dll-Funktion 'ReleaseRenderDevice'
        _ReleaseRenderDevice = (RELEASERENDERDEVICE)
            GetProcAddress(m_hDLL,
```

```

        "ReleaseRenderDevice");
    }
    // call dll's release function
    if (m_pDevice) {
        hr = _ReleaseRenderDevice(&m_pDevice);
        if(FAILED(hr)){
            m_pDevice = NULL;
        }
    }
} // release

```

Soll das Objekt aus der DLL, also das Render-Device, freigegeben werden, dann suchen wir zuerst die Adresse der entsprechenden Funktion, die die DLL für eben diesen Zweck für uns exportiert, also die Funktion `ReleaseRenderDevice()`. Haben wir diese Adresse gefunden, so speichern wir sie in einem passenden Pointer mit der Bezeichnung `_ReleaseRenderDevice` und rufen sie mit dem freizugebenden Objekt auf. Dieses ist ja als Attribut der Klasse `ZFXRenderer` gespeichert.

Am Anfang erscheint das ein wenig verwirrend, insbesondere die beiden ominösen Typdefinitionen im Header `CREATERENDERDEVICE` und `RELEASE-renderDevice`. Diese sind dazu da, dass der Compiler weiß, welche Parameterliste sich an der gefundenen Adresse zu den gesuchten Funktion in der DLL befindet. Nur so können wir einen korrekten Pointer auf diese Adresse setzen, um die Funktion richtig aufrufen zu können.

Die gute Nachricht ist, dass das schon alles war, was wir in der statischen Bibliothek implementieren müssen. Die einzige Änderung, die man hier später vornehmen muss, ist die Auswahl der zu ladenden DLL, wenn man mehrere zur Auswahl hat.

3.5 Implementierung der dynamischen Bibliothek

Als dynamische Bibliothek haben wir ja bereits das Projekt `ZFXD3D` erzeugt. In dieser dynamischen Bibliothek wollen wir eine Kapselung für `Direct3D` schreiben. In diesem Kapitel fällt unsere Implementierung noch ein wenig rudimentär aus, aber im Verlauf dieses Buches werden wir die Implementierung ständig um weitere Funktionen bereichern. Betrachten wir nun zuerst einmal die Definition der Klasse. Ein besonderes Augenmerk richten wir hier auf das Attribut `m_pChain[MAX_3DHWND]`, wobei es sich um ein Array von Swap-Chain-Elementen handelt. Selbst wer schon öfter mit `Direct3D` gearbeitet hat, dem sind die so genannten Swap Chains von `DirectX` eventuell immer noch neu. Diese brauchen wir, um mit `Direct3D` komfortabel in beliebig viele Child-Windows rendern zu können. Aber das werden wir später noch sehen. Der Wert von `MAX_3DHWND` ist übrigens als Definition in der Interface-Datei zu finden.



```
// in der Datei: ZFXD3D.h
class ZFXD3D : public ZFXRenderDevice {
public:
    ZFXD3D(HINSTANCE hDLL);
    ~ZFXD3D(void);

    // Initialisierungsfunktionen
    HRESULT Init(HWND, const HWND*, int, int, int,
                bool);
    BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

    // Interface-Funktionen
    void    Release(void);
    bool    IsRunning(void) { return m_bRunning; }
    HRESULT BeginRendering(bool,bool,bool);
    HRESULT Clear(bool,bool,bool);
    void    EndRendering(void);
    void    SetClearColor(float, float, float);
    HRESULT UseWindow(UINT hWndd);

private:
    ZFXD3DEnum        *m_pEnum;
    LPDIRECT3D9        m_pD3D;
    LPDIRECT3DDEVICE9  m_pDevice;
    LPDIRECT3DSWAPCHAIN9 m_pChain[MAX_3DHWND];
    D3DPRESENT_PARAMETERS m_d3dpp;
    D3DCOLOR            m_ClearColor;
    bool                m_bIsSceneRunning;
    bool                m_bStencil;

    // startet die API
    HRESULT Go(void);

    // Protokollieren des Ablaufs
    void Log(char *, ...);
}; // class
```

In dieser Definition finden wir neben dem Konstruktor und Destruktor zunächst acht öffentliche Member-Funktionen. Sieben davon sind Funktionen, die durch das Interface zwingend vorgeschrieben sind. Hinzugekommen ist hier lediglich die Funktion `ZFXD3D::DlgProc`. Da wir nur mit Pointern vom Typ des Interfaces arbeiten, ist diese Funktion von außen nicht sichtbar. Wir benötigen sie aber innerhalb der Klassen-Implementierung von `ZFXD3D` als öffentlich zugängliche Funktion. Das sehen wir aber gleich, sobald wir zu dem Dialog dieser Klasse kommen.

Member-Funktionen in abgeleiteten Klassen, die die Implementierung von rein virtuellen Funktionen einer abstrakten Klasse darstellen, werden von C++ automatisch als virtuelle Funktionen behandelt. Die explizite Angabe des Schlüsselwortes `virtual` ist hier nicht nötig.



Enumeration aus dem DirectX SDK

An geschützten Mitgliedern verfügt die Klasse über eine Reihe von Attributen, die unter anderem die wichtigsten Direct3D-Objekte und Informationen über die Grafikkarte speichern. Dazu benötigen wir einen Satz an Funktionen, die die Informationen der Grafikkarte abfragen (Enumeration) und auflisten. Im Folgenden werde ich voraussetzen, dass die Enumeration der vorhandenen Grafikkarten unter DirectX bekannt ist.² Als Attribut verwende ich in dieser Klasse unter anderem `m_pEnum` als Objekt der Klasse `ZFXD3DEnum`. Diese Klasse führt die Enumeration der verfügbaren Grafikkarten und deren Modi durch. Das Abdrucken dieser Klasse spare ich mir hier, da diese Enumeration eine vereinfachte Variante der Common Files aus dem DirectX SDK ist, die ich als bekannt voraussetze. Die Implementierung dieser Klasse findet ihr natürlich auf der CD-ROM zu diesem Buch. Wer auf diesem Gebiet noch ein wenig unsicher ist, der sollte sich diese noch einmal zu Gemüte führen.

Ablauf des Hochfahrens von Direct3D

Besprechen wir erst einmal den groben Plan von dem, was wir vorhaben. Unser Interface besteht zunächst nur darauf, dass wir eine `Init()`-Funktion implementieren, durch die die DLL die verwendete Grafik-API hochfährt. Es ist aber durchaus empfehlenswert, dieses Hochfahren flexibel zu gestalten. In der Implementierung der ZFXD3D-DLL werde ich es so machen, dass die DLL bei Aufruf der Initialisierungsfunktion eine Dialogbox anzeigt. Diese Dialogbox listet alle gefundenen Grafikkarten auf dem Zielcomputer auf, ebenso wie sämtliche Bildschirmauflösungen, die die entsprechende Grafikkarte fahren kann. Ebenso kann der Benutzer dann auswählen, ob er im Fullscreen-Modus oder im Windowed-Modus starten möchte. So erhält der Benutzer unserer Engine in einem einzigen Funktionsaufruf von außen die Möglichkeit, flexibel eine Grafikkarte auszuwählen (falls mehrere in einem Computer vorhanden sind) und den Bildschirmmodus anzugeben oder die Anwendung im Fenster zu starten. Bequemer geht es gar nicht mehr.

Aber bevor wir dazu kommen, müssen wir uns erst noch ansehen, wie wir über die oben besprochenen exportierten Funktionen im C-Stil auf die Klasse in der DLL zugreifen können.

Exportierte Funktionen

Bei der Erstellung des Projekts ZFXD3D hatten wir dem Projekt ja auch gleich ein paar leere Dateien hinzugefügt – unter anderem die Datei `ZFXD3D.def`, in der die Exportinformationen für die DLL stehen werden. Das klingt viel aufwändiger, als es ist, denn die Datei erhält als Inhalt nur die folgenden Zeilen:

² Siehe auch das Tutorial unter www.zfx.info.

```
;ZFXD3D.def
;Die folgenden Funktionen werden von der DLL exportiert
LIBRARY "ZFXD3D.dll"
EXPORTS
    CreateRenderDevice
    ReleaseRenderDevice
```

Zum einen wird der Name der DLL angegeben, damit wir sie beim Laden auch eindeutig identifizieren können. Zum anderen sind dort die beiden Funktionen im C-Stil aufgelistet, über die wir ja bereits in der statischen Bibliothek ZFXRenderer auf die DLL zugegriffen haben. Wichtig sind hierbei nur die Namen der Funktionen, nicht aber die Parameterlisten. Die Implementierung dieser Funktionen schreiben wir dann in eine *.cpp-Datei, die zu dem DLL-Projekt gehört.



*Wenn man das Visual Studio .NET verwendet, dann funktioniert die Verwendung eines *.def Files nicht immer korrekt. Manchmal verweigert das Programm dann einfach den Dienst, weil die exportierten Funktionen in der DLL nicht gefunden werden – die Funktion GetProcAddress() liefert lediglich NULL zurück. Das kann man aber leicht beheben, indem man auf das *.def File verzichtet und die Funktionen anders exportiert.*

Aus der Interface-Header-Datei entfernt man die Deklarationen der exportierten Funktionen, hier wären das also CreateRenderDevice() und ReleaseRenderDevice() in der Datei ZFXRenderDevice.h. Die beiden typedef Anweisungen bleiben dort aber stehen. Die Deklarationen schreibt man statt dessen in eine Header-Datei in der DLL, und fügt sowohl den Deklarationen als auch den Funktionsköpfen bei der Funktions-Definition das folgende Präfix hinzu:

```
extern "C" __declspec(dllexport)
```

Auf diese Weise sind die Funktionen in der DLL auch als Exporte gekennzeichnet, und werden nun auch in einem Build von Visual Studio .NET korrekt erstellt.

Wenn wir uns den Quellcode dieser Funktionen einmal ansehen, so sind sie beinahe lächerlich kurz. In diesen Funktionen muss ja auch nichts anderes geschehen als das Erzeugen bzw. das Freigeben eines Objekts der Klasse. Das sieht dann so aus:



```
// in der Datei: ZFXD3D_init.cpp
#include "ZFX.h"

HRESULT CreateRenderDevice(HINSTANCE hDLL,
                           ZFXRenderDevice **pDevice) {
    if(!*pDevice) {
        *pDevice = new ZFXD3D(hDLL);
    }
}
```



```

        return ZFX_OK;
    }
    return ZFX_FAIL;
}

```

```

HRESULT ReleaseRenderDevice(ZFXRenderDevice **pDevice) {
    if(!*pDevice) {
        return ZFX_FAIL;
    }
    delete *pDevice;
    *pDevice = NULL;
    return ZFX_OK;
}

```

Bei der Erstellung eines Render-Device-Objekts unserer Klasse müssen wir das Handle auf die geladene DLL angeben, da wir diese im Konstruktor noch brauchen. Dann erstellen wir einfach ein Objekt unserer Klasse und setzen den Pointer vom Typ des Interfaces auf dieses Objekt. Damit können wir über den Interface-Pointer auf das Objekt zugreifen. Bei der Freigabe löschen wir dann dieses Objekt einfach wieder, wodurch natürlich der Destruktor der Klasse ZFXD3D aufgerufen wird.

In der Datei `ZFX.h` habe ich diverse Werte definiert. Insbesondere handelt es sich dabei um Rückgabewerte vom Typ `HRESULT` für unsere Funktionen. Später, wenn wir nicht mehr nur mit `ZFX_FAIL` oder `ZFX_OK` arbeiten, werden wir differenziertere Fehler ausgeben, anhand derer der Benutzer sehen kann, warum eine Funktion beispielsweise fehlgeschlagen ist. Diese Fehlerwerte liste ich hier auch nicht alle auf. Aufgrund ihres Präfixes sind sie eindeutig zu erkennen, und die Definitionen stehen in der angesprochenen Header-Datei, die wir in allen unseren Projekten verwenden werden.

*Eigene Fehler-
werte definieren*

Jetzt haben wir es tatsächlich geschafft, unsere Klasse in der DLL von außen verfügbar zu machen. Nun können wir endlich daran gehen, die Funktionalität unseres Render-Device-Objekts zu programmieren.

Komfort durch einen Dialog

Bevor wir die eigentliche Enumeration der vorhandenen Grafik-Hardware implementieren, benötigen wir auch eine Art Container, der die später gesammelten Informationen darstellen und zur Auswahl stellen kann. Hierfür verwenden wir eine Dialogbox. Wir aktivieren also das ZFXD3D-Projekt als aktives Projekt und fügen ihm eine Dialogbox hinzu. Dazu gehen wir in das Menü EINFÜGEN und wählen den Menüpunkt RESSOURCE ... aus. In der nun erscheinenden Dialogbox wählen wir aus der Liste den Eintrag *Dialog* und klicken auf den Button NEU. Jetzt haben wir ein neues Ressourcen-Objekt mit *Visual C++* erzeugt, und das Programm wechselt automatisch zu dem integrierten Editor für Dialogboxen.

*Einen Dialog
erzeugen*

Hier können wir durch einen Doppelklick mit der Maus auf ein Steuerelement oder den Dialog selbst ein Eigenschaften-Menü aufrufen. Unter dem Reiter ALLGEMEIN findet sich hier ein Feld mit der Beschriftung *ID*. Dies ist besonders wichtig für uns, weil wir hier eine eindeutige Bezeichnung für ein Steuerelement des Dialoges oder den Dialog selbst vergeben können. Dem Dialog geben wir auf diese Weise die Bezeichnung »dlgChangeDevice«, wobei es hier besonders wichtig ist, dass wir die ID in dem Feld auch mit den Anführungszeichen eingeben. Über diese ID werden wir den Dialog später aufrufen können. Die folgende Übersicht zeigt, welche Steuerelemente unser Dialog neben ein paar statischen Textfeldern noch enthält (die IDs werden hier aber ganz normal ohne Anführungszeichen eingegeben):

- ➔ Combobox mit der Bezeichnung IDC_ADAPTER
- ➔ Combobox mit der Bezeichnung IDC_MODE
- ➔ Combobox mit der Bezeichnung IDC_ADAPTERFMT
- ➔ Combobox mit der Bezeichnung IDC_BACKFMT
- ➔ Combobox mit der Bezeichnung IDC_DEVICE
- ➔ Radiobutton mit der Bezeichnung IDC_FULL
- ➔ Radiobutton mit der Bezeichnung IDC_WND
- ➔ Buttons mit der Bezeichnung IDOK und IDCANCEL

Die Comboboxen dienen dazu, die verfügbaren Grafikkarten und deren Betriebsmodi zur Auswahl anzubieten. Im DirectX-Jargon bezeichnet man eine solche Grafikkarte als *Adapter*. Jeder Adapter verfügt über eine gewisse Anzahl Modi, auch Bildschirmauflösung genannt, die er darstellen kann, beispielsweise 800x600, 1024x768 usw. Dann gibt es zwei weitere Comboboxen für die Farbformate, denn seit DirectX 9 ist es möglich, den Back-Buffer mit einer anderen Auflösung zu betreiben als den Front-Buffer, wenn das Programm im Fenstermodus betrieben wird. Die letzte Combobox dient dazu, zwischen den verschiedenen Direct3D-Device-Typen auszuwählen. Hier kommen eigentlich nur zwei Typen in Betracht: einmal das HAL-Device, also die Grafikkarte selbst, und zum anderen das REF-Device, also der *Software Reference Rasterizer*. Wir erinnern uns daran, dass DirectX beinahe sämtliche Funktionalität in Software emulieren kann, wenn diese durch die Hardware nicht unterstützt wird. Das ist zwar quälend langsam, kann zu Testzwecken aber sehr sinnvoll sein.

Die beiden Radiobuttons dienen zur Auswahl, ob die Anwendung im Full-screen-Modus oder im Fenster gestartet werden soll. Abbildung 3.1 zeigt, wie unser fertiger Dialog jetzt aussehen sollte.

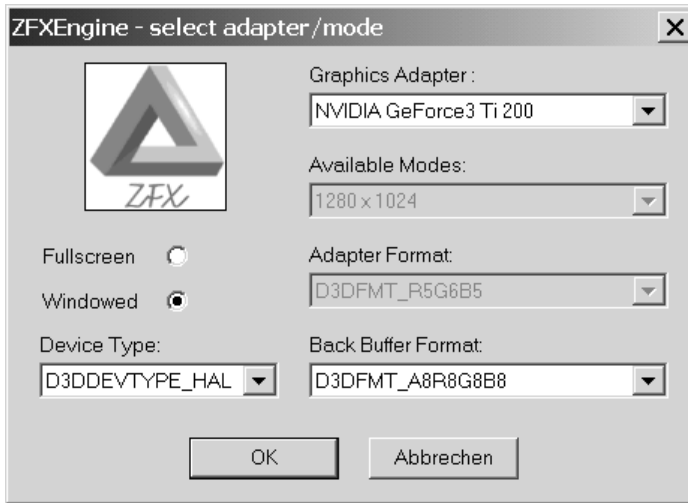


Abbildung 3.1:
Der Dialog, wie er
später im
Programm
erscheinen wird

Klicken wir nun auf den Button zum Speichern unseres gesamten Projekts in *Visual C++*, dann werden wir aufgefordert, unsere neue Ressource, also die Dialogbox, zu speichern. Dazu wählen wir das Verzeichnis unseres Projekts ZFXD3D aus und speichern die Ressource unter dem Namen `dlgChangeDevice.rc` ab, wobei automatisch die Datei `resource.h` miterzeugt wird. In der *Visual C++*-Projektübersicht haben wir auch schon einen Ordner namens *Ressourcendateien*, der für jedes Projekt automatisch angelegt wird. Auf diesen klicken wir nun mit der rechten Maustaste und wählen aus dem erscheinenden Menü den Eintrag *Dateien zu Ordner hinzufügen ...* aus. Jetzt fügen wir die beiden eben abgespeicherten Dateien in diesen Ordner ein. Damit können wir nun die Dialogbox in unserem Projekt verwenden.

*Dialog im Projekt
verfügbar machen*

Ich gehe davon aus, dass die Arbeit mit selbst erstellten Dialogen nichts neues für euch ist, daher behandle ich die entsprechenden Schritte eher knapp. Ein Dialog ist unter Windows nichts anderes als eine besondere Art von Fenster. Daher wird er auch ganz genauso wie ein Fenster behandelt. Wir haben in dem Dialog bereits einige Steuerelemente und deren IDs. Wenn wir nun diese Steuerelemente abfragen wollen, dann machen wir das genauso, wie wir es in jedem anderen Fenster machen würden, nämlich über eine Callback-Funktion, die die Nachrichten verarbeitet, die an den Dialog geschickt werden. Um einen Dialog aufzurufen und ihm eine Callback-Funktion zuzuweisen, verwenden wir die folgende Funktion der WinAPI:

*Anzeigen von
Dialogen*

```
int DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate,
             HWND hWndParent, DLGPROC lpDialogFunc);
```

Der erste Parameter der Funktion verlangt nach dem Instanzzähler der Anwendung. Dies ist der Wert, den uns die Funktion `LoadLibraryEx()` für die DLL zurückgeliefert hat. Für den zweiten Parameter geben wir die ID für den anzuzeigenden Dialog an. Wenn wir hier eine ID in Form eines Strings

angeben wollen, müssen wir die entsprechende ID im Ressourcen-Editor auf alle Fälle auch mit Anführungszeichen angegeben haben. Für den dritten Parameter geben wir das Handle auf das Fenster an, zu dem der Dialog gehört. Dies ist in der Regel das Hauptfenster der Anwendung, die unsere DLL verwendet. Als letzten Parameter geben wir den Namen der Callback-Funktion an, die die Nachrichten des Dialogs bearbeitet. Hier haben wir allerdings ein kleines Problem, weil wir so ohne weiteres keine Funktion einer Klasse als Callback-Funktion angeben können. Um das zu umgehen, erzeugen wir eine normale C-Funktion, die dann über ein Objekt der Klasse auf die entsprechende öffentliche Funktion zur Verarbeitung des Dialogs verweist. Und das ist eben jene Funktion `ZFXD3D::DlgProc`. Die Funktion zur Anzeige eines Dialogs hat aber noch einen Rückgabewert vom Typ `int`. Dieser Rückgabewert kann über die Funktion zum Beenden eines Dialogs angegeben werden. Dann kommen wir darauf zurück.

Den Aufruf des Dialogs und die Zuweisung der Callback-Funktion erledigen wir in der Initialisierungsfunktion der Klasse. Hier schauen wir uns erst einmal nur die Callback-Funktion an und besprechen die Funktionen, die sie zu erfüllen hat.



```
ZFXDEVICEINFO  g_xDevice;
D3DDISPLAYMODE g_Dspsmd;
D3DFORMAT      g_fmtA;
D3DFORMAT      g_fmtB;

BOOL CALLBACK ZFXD3D::DlgProc(HWND hDlg, UINT message,
                               WPARAM wParam,
                               LPARAM lParam) {

    DIBSECTION dibSection;
    BOOL        bWnd=FALSE;

    // hole die Handles
    HWND hFULL      = GetDlgItem(hDlg, IDC_FULL);
    HWND hWND       = GetDlgItem(hDlg, IDC_WND);
    HWND hADAPTER   = GetDlgItem(hDlg, IDC_ADAPTER);
    HWND hMODE      = GetDlgItem(hDlg, IDC_MODE);
    HWND hADAPTERFMT = GetDlgItem(hDlg, IDC_ADAPTERFMT);
    HWND hBACKFMT   = GetDlgItem(hDlg, IDC_BACKFMT);
    HWND hDEVICE    = GetDlgItem(hDlg, IDC_DEVICE);

    switch (message) {

        // Fenster-Modus vorselektieren
        case WM_INITDIALOG: {
            SendMessage(hWND, BM_SETCHECK, BST_CHECKED, 0);
            m_pEnum->Enum(hADAPTER, hMODE, hDEVICE,
                        hADAPTERFMT, hBACKFMT,
                        hWND, hFULL, m_pLog);
        }
    }
}
```

```

    return TRUE;
}

// Logo rendern (g_hBMP in Init() initialisiert)
case WM_PAINT: {
    if (g_hBMP) {
        GetObject(g_hBMP, sizeof(DIBSECTION),
                  &dibSection);
        HDC      hdc = GetDC(hDlg);
        HDRAWIB hdd = DrawDibOpen();
        DrawDibDraw(hdd, hdc, 50, 10, 95, 99,
                    &dibSection.dsBmih,
                    dibSection.dsBm.bmBits, 0, 0,
                    dibSection.dsBmih.biWidth,
                    dibSection.dsBmih.biHeight, 0);
        DrawDibClose(hdd);
        ReleaseDC(hDlg, hdc);
    }
    } break;

// ein Control hat eine Meldung
case WM_COMMAND: {
    switch (LOWORD(wParam)) {
        // Okay-Button
        case IDOK: {
            m_bWindowed = !SendMessage(hFULL,
                                         BM_GETCHECK, 0, 0);
            m_pEnum->GetSelections(&g_xDevice,
                                   &g_Dspmd,
                                   &g_fmtA,
                                   &g_fmtB);
            GetWindowText(hADAPTER, m_chAdapter, 256);
            EndDialog(hDlg, 1);
            return TRUE;
        } break;

        // Cancel-Button
        case IDCANCEL: {
            EndDialog(hDlg, 0);
            return TRUE;
        } break;

        case IDC_ADAPTER: {
            if (HIWORD(wParam) == CBN_SELCHANGE)
                m_pEnum->ChangedAdapter();
        } break;

        case IDC_DEVICE: {
            if (HIWORD(wParam) == CBN_SELCHANGE)
                m_pEnum->ChangedDevice();
        } break;
    }
}

```

```

        case IDC_ADAPTERFMT: {
            if(HIWORD(wParam)==CBN_SELCHANGE)
                m_pEnum->ChangedAdapterFmt();
            } break;
        case IDC_FULL: case IDC_WND: {
            m_pEnum->ChangedWindowMode();
            } break;

    } // switch [CMD]
    } break; // case [CMD]
} // switch [MSG]
return FALSE;
}

```

Bei der Initialisierung des Dialogs

Wenn der Dialog initialisiert wird, müssen wir bereits ein paar Aktionen durchführen. Insbesondere rufen wir zuerst die Funktion `ZFXD3DEnum::Enum` auf. Als Parameter erwartet diese Funktion die Handles auf sämtliche Steuerelemente des Dialogs, damit sie die Einträge der Comboboxen füllen kann. Die Klasse `ZFXD3DEnum` speichert diese Handles in Member-Variablen, um immer Zugriff auf die Comboboxen zu haben. Die Funktion fährt dann fort und führt die Enumeration durch. Die Comboboxen werden mit entsprechenden Einträgen betankt.

Im zweiten Abschnitt der Callback-Funktion des Dialogs malen wir das ZFX-Logo in die Dialogbox. Dazu muss die Bibliothek `vwf32.lib` (Video for Windows) gelinkt und der Header `vwf.h` eingebunden werden. Beachtet, dass die Bitmap-Datei bereits in der Funktion `ZFXD3D::Init` geladen wird.

Nachrichten der Steuerelemente des Dialogs

Die Nachricht `WM_COMMAND` wird von Windows an den Dialog geschickt, wenn eines seiner Steuerelemente an Windows gemeldet hat, dass ein Event stattgefunden hat. Wir suchen also in der Callback-Funktion im unteren Wort (`LOWORD`) des `wParam`-Parameters der Nachricht nach der ID des Steuerelementes, das die Nachricht ausgelöst hat. Handelt es sich dabei um eine der Comboboxen, deren gewählter Eintrag sich geändert hat, oder um einen der beiden Radiobuttons, dann rufen wir die entsprechende Funktion der Klasse `ZFXD3DEnum` auf:

- ➡ `ZFXD3DEnum::ChangedAdapter`
- ➡ `ZFXD3DEnum::ChangedDevice`
- ➡ `ZFXD3DEnum::ChangedAdapterFmt`
- ➡ `ZFXD3DEnum::ChangedWindowMode`

Diese Funktionen dienen dazu, die Listen der Enumeration zu durchlaufen, und die Einträge der Comboboxen zu aktualisieren. Wählt der Benutzer beispielsweise einen anderen Adapter, so müssen die Einträge der anderen Comboboxen mit den verschiedenen Betriebsarten dieses Adapters gefüllt

werden usw. Die entsprechenden Funktionen sind wiederum größtenteils analog zu denen des DirectX SDK. Sehen wir uns also noch die beiden verbleibenden Steuerelemente an, nämlich die Buttons für »OK« und »Abbrechen«.

```
BOOL EndDialog(HWND hWnd, int nResult);
```

*Beenden eines
Dialogs*

Der erste Parameter der Funktion ist natürlich das Handle auf den Dialog, der beendet werden soll. Der zweite Parameter ist der Wert, den die Funktion, durch die der Dialog erzeugt wurde, als Rückgabewert liefern soll. In diesem Fall war das die Funktion `DialogBox()`. Wurde also der ABBRECHEN-Button im Dialog angeklickt, dann geben wir den Wert 0 zurück, um ein Abbrechen des Dialogs ohne Fehler anzuzeigen. Mehr müssen wir beim Anklicken des ABBRECHEN-Buttons nicht tun.

Wurde nun aber der OK-Button im Dialog angeklickt, dann haben wir etwas mehr zu tun. Jetzt ist davon auszugehen, dass der Benutzer in dem Dialog die Grafikkarte, die Bildschirmauflösung und alle weiteren Einstellungen gewählt hat und das Render-Device starten möchte. In diesem Fall holen wir uns die Einträge und Zustände aller Steuerelemente des Dialogs über die Funktion `ZFXD3DEnum::GetSelections` und speichern ihre aktuellen Einstellungen in den entsprechenden globalen Variablen bzw. Attributen der Klasse `ZFXD3D`. Die Struktur `ZFXDEVICEINFO` beinhaltet dabei ein paar Werte aus der Enumeration, beispielsweise die Eigenschaften des Devices, zu welchem Adapter es gehört und welche Modi es verwenden kann. Diese Struktur ist aber ebenfalls fast analog zu dem Äquivalent aus der Enumeration des DirectX SDK.

Dann beenden wir den Dialog und geben den Wert 1 dabei zurück, der andeuten soll, dass der Dialog erfolgreich beendet wurde. Wie das Programm nun auf das Beenden des Dialogs reagiert, sehen wir gleich in der Initialisierungsfunktion.

Initialisierung, Enumeration und Shutdown

Zunächst einmal rufen wir in der aus der DLL exportierten Funktion `CreateRenderDevice()` den Konstruktor der `ZFXD3D`-Klasse auf. Dieser sieht wie folgt aus:

```
ZFXD3D *g_ZFXD3D=NULL;
```

```
ZFXD3D::ZFXD3D(HINSTANCE hDLL) {
    m_hDLL          = hDLL;
    m_pEnum          = NULL;
    m_pD3D           = NULL;
    m_pDevice        = NULL;
```



```

m_pLog          = NULL;
m_ClearColor    = D3DCOLOR_COLORVALUE(
    0.0f, 0.0f, 0.0f, 1.0f);
m_bRunning      = false;
m_bIsSceneRunning = false;

m_nActivehWnd   = 0;

g_ZFXD3D = this;
}

```

Es handelt sich hierbei um einen vollkommen durchschnittlichen Konstruktor, der nur dazu genutzt wird, ein paar Startwerte für die Attribute festzulegen. Beachtet, dass wir hierbei das Handle auf die geladene DLL in einem Attribut speichern. Bemerkenswert ist hier lediglich, dass wir uns in einem globalen Pointer namens `g_ZFXD3D` noch die Adresse des durch den Konstruktor erstellten Objekts merken. Diese benötigen wir später noch einmal in unserem Programm, und zwar für die Callback-Funktion des Dialogs. Der Destruktor der Klasse sieht wie folgt aus:



```

ZFXD3D::~ZFXD3D() {
    Release();
}

void ZFXD3D::Release() {
    if (m_pEnum) {
        m_pEnum->~ZFXD3DEnum();
        m_pEnum = NULL;
    }

    if(m_pDevice) {
        m_pDevice->Release();
        m_pDevice = NULL;
    }

    if(m_pD3D) {
        m_pD3D->Release();
        m_pD3D = NULL;
    }

    fclose(m_pLog);
}

```

Wir müssen nur darauf achten, dass wir auch brav wieder alle Objekte freigeben, die initialisiert worden sein könnten.

Sehen wir uns die Funktion für die Initialisierung an. Dort werden also die spannenden Sachen passieren, wenn das schon im Konstruktor nicht der Fall war. Diese Funktion können wir später aus anderen Anwendungen heraus separat aufrufen, nachdem wir über die exportierten Funktionen ein Objekt der Klasse `ZFXD3D` erhalten haben.



```

HBITMAP g_hBMP;

HRESULT ZFXD3D::Init(HWND hWnd, const HWND *hWnd3D,
                    int nNumhWnd, int nMinDepth,
                    int nMinStencil, bool bSaveLog) {
    int nResult;

    m_pLog = fopen("log_renderdevice.txt", "w");
    if (!m_pLog) return ZFX_FAIL;

    // sollen wir Child-Windows verwenden?
    if (nNumhWnd > 0) {
        if (nNumhWnd > MAX_3DHWND) nNumhWnd = MAX_3DHWND;
        memcpy(&m_hWnd[0], hWnd3D, sizeof(HWND)*nNumhWnd);
        m_nNumhWnd = nNumhWnd;
    }
    // sonst speichern wir das Handle des Hauptfensters
    else {
        m_hWnd[0] = hWnd;
        m_nNumhWnd = 0;
    }
    m_hWndMain = hWnd;;

    if (nMinStencil > 0) m_bStencil = true;

    // Erzeuge das Enum-Objekt
    m_pEnum = new ZFXD3DEnum(nMinDepth, nMinStencil);

    // Lade das ZFX-Logo
    g_hBMP = (HBITMAP)LoadImage(NULL, "zfx.bmp",
                                IMAGE_BITMAP, 0, 0,
                                LR_LOADFROMFILE |
                                LR_CREATEDIBSECTION);

    // Öffne den Auswahl-Dialog
    nResult = DialogBox(m_hDLL, "dlgChangeDevice", hWnd,
                       DlgProcWrap);

    // Ressource wieder freigeben
    if (g_hBMP) DeleteObject(g_hBMP);

    // Fehler im Dialog
    if (nResult == -1)
        return ZFX_FAIL;
    // Dialog vom Benutzer abgebrochen
    else if (nResult == 0)
        return ZFX_CANCELED;
    // Dialog mit OK-Button beendet
    else
        return Go();
}

```

*Wohin mit der
Grafikausgabe?*

Dieser Funktion können wir sechs Parameter übergeben. Zuerst einmal ist dies ein Handle auf das Hauptfenster der Anwendung. Dies benötigen wir einerseits, um den Dialog als Kind dieses Fensters anzuzeigen, und andererseits wird dieses Fenster für die Grafik-Darstellung von unserem Render-Device verwendet, wenn wir eine Applikation im Fullscreen-Modus ausführen. Möchten wir jedoch die Anwendung im Fenster starten, so können wir im zweiten Parameter optional ein ganzes Array von Handles auf Child-Windows angeben, die für die Grafikausgabe verwendet werden sollen. Das ist beispielsweise bei Editoren sinnvoll, bei denen die Grafikausgabe nur einen Teil des Programm-Fensters einnimmt. Über dieses Array können wir also beliebig viele (na gut, nur `MAX_3DHWND` Stück) Child-Windows angeben. Diese können dann alle zum Rendern von Grafik über unser Device benutzt werden. Die Umschaltung zwischen diesen verschiedenen Fenstern (beim Rendern kann ja immer nur eins aktiv sein) erledigen wir über die Funktion `ZFXD3D::UseWindow`.

Der dritte Parameter gibt schließlich an, wie viele Handles auf Child-Windows in dem Array zu finden sind. Die folgenden beiden Parameter geben an, wie viele Bits der Depth-Buffer und der Stencil-Buffer jeweils mindestens haben sollen. Der letzte Parameter dient dazu, ein abgesichertes Log zu schreiben, aber das brauchen wir erst in späteren Kapiteln.

*Anzeige des
Dialogs und
Callback-Funktion*

Wie wir aber unschwer erkennen können, macht die Initialisierungsfunktion eigentlich auch nicht viel. Nach dem Umspeichern von Handles in Member-Variablen und dem Laden unseres Eye-Candy, des ZFX-Logos aus einer Bitmap-Datei, ruft sie lediglich die WinAPI-Funktion `DialogBox()` auf, um unseren Dialog namens *dlgChangeDevice* anzuzeigen. Als Rückgabewerte sind hier, durch die Callback-Funktion, drei Werte möglich. Falls der Benutzer den Dialog korrekt über den OK-Button beendet hat, rufen wir die Funktion `ZFXD3D::Go` auf. Diese wird dann Direct3D mit den im Dialog gewählten Einstellungen starten. Aber dazu kommen wir gleich. Zuerst schauen wir uns die Callback-Funktion näher an. Wir verwenden hier die Funktion `DlgProcWrap()`, weil wir als Callback-Funktion keine normale Member-Funktion einer Klasse angeben können. Also brauchen wir ein kleines Stützkorsett, das uns dabei hilft. Und das ist eben diese Funktion:



```

BOOL CALLBACK DlgProcWrap(HWND hDlg,
                           UINT message,
                           WPARAM wParam,
                           LPARAM lParam) {
    return g_ZFXD3D->DlgProc(hDlg,
                             message,
                             wParam,
                             lParam);
}

```

Über das globale Objekt unserer Klasse, das wir im Konstruktor gespeichert haben, leiten wir den Aufruf der Callback-Funktion einfach auf die entsprechende Member-Funktion unserer Klasse um. Das ist deshalb notwendig, weil wir in der Callback-Funktion auf diverse Member-Attribute der Klasse zugreifen wollen. Die Implementierung der Funktion `ZFXD3D::DlgProc` haben wir ja vorhin schon gesehen. In dieser Funktion geht es hauptsächlich um die Enumeration und die Auflistung der vorhandenen Grafikkarten und Bildschirmmodi. Die entsprechenden Funktionen, die die Callback-Funktion dafür aufruft, werden dann über die Klasse `ZFXD3DEnum` implementiert.

*Aufgabe der
Callback-Funktion*

Nun können wir endlich dazu kommen, wie wir Direct3D starten. Wird die Dialogbox durch Anklicken des OK-Buttons beendet, liest die Callback-Funktion die gewählten Einträge aus den Steuerelementen und kehrt zu der Initialisierungsfunktion zurück. Dort wird dann die Funktion `ZFXD3D::Go` aufgerufen.

Direct3D starten

```
HRESULT ZFXD3D::Go(void) {
    ZFXCOMBOINFO    xCombo;
    HRESULT          hr;
    HWND             hwnd;

    // Erzeuge das Direct3D-Hauptobjekt
    if (m_pD3D) {
        m_pD3D->Release();
        m_pD3D = NULL;
    }
    m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );
    if(!m_pD3D)
        return ZFX_CREATEAPI;

    // Finde die passende Combo
    for (UINT i=0; i<g_xDevice.nNumCombo; i++) {
        if ( (g_xDevice.d3dCombo[i].bWindowed ==
            m_bWindowed)
            && (g_xDevice.d3dCombo[i].d3dDevType ==
            g_xDevice.d3dDevType)
            && (g_xDevice.d3dCombo[i].fmtAdapter ==
            g_fmtA)
            && (g_xDevice.d3dCombo[i].fmtBackBuffer ==
            g_fmtB) )
        {
            xCombo = g_xDevice.d3dCombo[i];
            break;
        }
    }

    // Betanke die PresentParameters-Struktur
    ZeroMemory(&m_d3dpp, sizeof(m_d3dpp));
    m_d3dpp.Windowed      = m_bWindowed;
```



```

m_d3dpp.BackBufferCount      = 1;
m_d3dpp.BackBufferFormat     = g_Dspmd.Format;
m_d3dpp.EnableAutoDepthStencil = TRUE;
m_d3dpp.MultiSampleType      = xCombo.msType;
m_d3dpp.AutoDepthStencilFormat =
                                xCombo.fmtDepthStencil;
m_d3dpp.SwapEffect            =
                                D3DSWAPEFFECT_DISCARD;

// ist ein Stencil-Buffer aktiv??
if ( (xCombo.fmtDepthStencil == D3DFMT_D24S8)
    || (xCombo.fmtDepthStencil == D3DFMT_D24X4S4)
    || (xCombo.fmtDepthStencil == D3DFMT_D15S1) )
    m_bStencil = true;
else
    m_bStencil = false;

// Fullscreen-Modus
if (!m_bWindowed) {
    m_d3dpp.hDeviceWindow = hwnd = m_hWndMain;
    m_d3dpp.BackBufferWidth = g_Dspmd.Width;
    m_d3dpp.BackBufferHeight = g_Dspmd.Height;
    ShowCursor(FALSE);
}

// Windowed-Modus
else {
    m_d3dpp.hDeviceWindow = hwnd = m_hWnd[0];
    m_d3dpp.BackBufferWidth =
        GetSystemMetrics(SM_CXSCREEN);
    m_d3dpp.BackBufferHeight =
        GetSystemMetrics(SM_CYSCREEN);
}

// Erstelle das Direct3D-Device
hr = m_pD3D->CreateDevice(g_xDevice.nAdapter,
                        g_xDevice.d3dDevType,
                        m_hWnd, xCombo.dwBehavior,
                        &m_d3dpp, &m_pDevice);

// Swap Chains erstellen, falls nötig
if ( (m_nNumhWnd > 0) && m_bWindowed) {
    for (UINT i=0; i<m_nNumhWnd; i++) {
        m_d3dpp.hDeviceWindow = m_hWnd[i];
        m_pDevice->CreateAdditionalSwapChain(
            &m_d3dpp, &m_pChain[i]);
    }
}

m_pEnum->ZFXD3DEnum();
m_pEnum = NULL;

```

```

if(FAILED(hr))
    return ZFX_CREATEDevice;

m_bRunning          = true;
m_bIsSceneRunning = false;
return ZFX_OK;
} // go

```

In dieser Funktion tun wir nichts anderes, als alle Combos vom Typ `ZFXCOMBOINFO` für das ausgewählte Device zu durchlaufen. Mit den entsprechenden Werten der Combo befüllen wir dann die `D3DPRESENT_PARAMETERS`-Struktur und initialisieren das Direct3D-Device durch den Aufruf von `IDirect3D8::CreateDevice`. Wir müssen dabei lediglich ein paar Unterscheidungen der beiden Fälle Fullscreen- oder Fenster-Modus beachten. Ab diesem Moment ist unser Render-Device dann einsatzbereit, immer vorausgesetzt, es sind keine Fehler aufgetreten. Eine Combo ist hier allerdings nicht mit einem Steuerelement *Combobox* zu verwechseln. Microsoft verwendet diese Bezeichnung seit DirectX 9 dafür, um eine Kombination von Front-Buffer- und Back-Buffer-Format, Adapter, Device-Typ, Vertex-Processing-Art und Depth-Stencil-Buffer-Format zusammenzufassen. Bei der Enumeration wurden entsprechend alle diese Combos für jedes vorhandene Device aller Adapter erzeugt.

**Struktur erstellen
und Device
initialisieren**

Am Ende der Funktion lüften wir dann das Geheimnis, wie wir durch Direct3D in beliebig viele Child-Windows rendern können. Falls der Benutzer das Programm im Fenster-Modus startet und die Applikation ein oder mehrere Child-Window-Handles angegeben hat, dann erstellen wir für jedes Child-Window ein eigenes Direct3D-Swap-Chain-Objekt. So können wir das Hauptfenster der Applikation weiterhin als normales Fenster nutzen und die Grafikausgabe in ein oder mehrere seiner Child-Windows realisieren.

Swap Chains

Im bisherigen Design würden wir den Anwender unserer DLL immer dazu zwingen, den Dialog aufzurufen, wenn er das Render-Device nutzen möchte. Dies ist sicherlich für Programme sinnvoll, die für den Fullscreen-Modus entwickelt wurden. Hier kann der Benutzer frei die Bildschirmauflösung bestimmen oder das Programm eben doch im Fenster-Modus laufen lassen.



Wenn man allerdings ein Programm entwickelt, das nur im Fenster-Modus laufen soll, beispielsweise einen Editor, dann macht das wenig Sinn. Hierzu habe ich in das Interface zusätzlich noch die Funktion `ZFXRenderDevice::InitWindowed` als Alternative zu dem Aufruf von `ZFXRenderDevice::Init` eingefügt. Diese ist eine Mischung aus der `Init()`- und der `Go()`-Methode und initialisiert das Device ohne Aufruf des Dialogs gleich im Fenstermodus. Für die Parameterliste ist die Angabe der Bytes für den Depth- und Stencil-Buffer daher unnötig.

Zwischen Child-Windows wechseln

Haben wir der `ZFXD3D::Init`-Methode tatsächlich ein Array von mehreren Handles auf Child-Windows angegeben, so müssen wir natürlich auch jederzeit in der Lage sein, eines der Child-Windows als aktives Render-Ziel für Direct3D einzustellen. Per Voreinstellung nutzen wir im Fenstermodus das erste Array-Element mit dem Index 0 (oder eben das Hauptfenster, falls gar keine Child-Windows angegeben sind).

Swap Chains wechseln

Mit Hilfe der folgenden Funktion können wir also zwischen den Child-Windows umschalten, sofern mehrere vorhanden sind. Dazu muss man wissen, dass Direct3D für jedes Child-Window, für das wir ein Swap-Chain-Objekt erzeugt haben, einen eigenen Back-Buffer erstellt hat. Möchten wir jetzt das aktive Swap-Chain-Objekt (also das Child-Window) ändern, dann müssen wir von dem zu aktivierenden Swap-Chain-Objekt einen Pointer auf dessen Back-Buffer abfragen und genau diesen Back-Buffer dann als neues Render-Target für das Direct3D-Device setzen. An den Aufrufen zum Rendern, Löschen des Back-Buffers usw. brauchen wir dabei überhaupt nichts zu verändern.



```
HRESULT ZFXD3D::UseWindow(UINT nHwnd) {
    LPDIRECT3DSURFACE9 pBack=NULL;

    if (!m_d3dpp.Windowed)
        return ZFX_OK;
    else if (nHwnd >= m_nNumHwnd)
        return ZFX_FAIL;

    // versuche, den richtigen Back-Buffer zu holen
    if (FAILED(m_pChain[nHwnd]->GetBackBuffer(0,
        D3DBACKBUFFER_TYPE_MONO, &pBack)))
        return ZFX_FAIL;

    // aktiviere ihn für das Device
    m_pDevice->SetRenderTarget(0, pBack);
    pBack->Release();
    m_nActiveHwnd = nHwnd;
    return ZFX_OK;
}
```

Einfacher geht's nicht

Damit sind wir nun in der Lage, beliebig viele Ansichten mit 3D-Grafik für unsere Applikationen zu erzeugen, und zwar ganz einfach dadurch, dass wir die Handles der Child-Windows der `ZFXD3D::Init`-Funktion übergeben und per Aufruf der Methode `ZFXD3D::UseWindow` ein beliebiges dieser Fenster aktivieren und dann Grafik rendern.

Demo-Applikation

Abbildung 3.2 zeigt einen Screenshot des Beispielprogramms dieses Kapitels. Dort sind vier Child-Windows im Hauptfenster der Applikation angelegt, die der Initialisierungsfunktion unseres Devices als zu verwendende



Abbildung 3.2:
Vier Child-Windows
im Hauptfenster

Fenster bekannt gemacht wurden. Startet man die Engine durch Auswahl im Dialog im Fenster-Modus, dann kann man diese vier Child-Windows verwenden. Startet man über den Dialog das Programm jedoch im Fullscreen-Modus, dann werden alle vier Child-Windows ignoriert, und das Hauptfenster der Anwendung wird zum Rendern verwendet. Den kompletten Quelltext der Demo-Applikation findet ihr am Ende dieses Kapitels.

Render-Funktionen

Jetzt haben wir es geschafft, unser Render-Device aus der DLL komfortabel zu initialisieren und können es auch wieder freigeben. Allerdings – fehlt da nicht noch etwas? Na klar, wir müssen ja auch irgendwie mit dem Ding arbeiten können! Wirklich etwas rendern werden wir jetzt noch nicht, lediglich den Bildschirm mit der Hintergrundfarbe löschen. Unter Direct3D benötigt man auch noch Funktionen, die zu Beginn und am Ende eines Render-Vorgangs aufgerufen werden müssen, um interne Strukturen für das Rendern einzustellen. Wir werden hier das Löschen der Buffer gleich mit diesem *Starten der Szene* vor dem Rendern verbinden. Ebenso müssen wir eine im Pixel-Buffer existierende gerenderte Szene auf den Bildschirm bringen (also vom Back-Buffer in den Front-Buffer flippen). Diesen Aufruf koppeln wir in unserer Implementierung an das Beenden der Szene nach dem Rendern.

Hier sind die Funktionen unseres Render-Devices, mit denen wir das alles umsetzen:

```
HRESULT ZFXD3D::BeginRendering(bool bClearPixel,  
                                bool bClearDepth,  
                                bool bClearStencil) {  
  
    DWORD dw=0;
```



```

// soll irgendetwas gelöscht werden?
if (bClearPixel || bClearDepth || bClearStencil) {
    if (bClearPixel)    dw |= D3DCLEAR_TARGET;
    if (bClearDepth)    dw |= D3DCLEAR_ZBUFFER;

    if (bClearStencil && m_bStencil)
        dw |= D3DCLEAR_STENCIL;

    if (FAILED(m_pDevice->Clear(0, NULL, dw,
                               m_ClearColor,
                               1.0f, 0)))

        return ZFX_FAIL;
}

if (FAILED(m_pDevice->BeginScene()))
    return ZFX_FAIL;

m_bIsSceneRunning = true;
return ZFX_OK;
}
/*-----*/

HRESULT ZFXD3D::Clear(bool bClearPixel,
                     bool bClearDepth,
                     bool bClearStencil) {
    DWORD dw=0;

    if (bClearPixel)    dw |= D3DCLEAR_TARGET;
    if (bClearDepth)    dw |= D3DCLEAR_ZBUFFER;

    if (bClearStencil && m_bStencil)
        dw |= D3DCLEAR_STENCIL;

    if (m_bIsSceneRunning)
        m_pDevice->EndScene();

    if (FAILED(m_pDevice->Clear(0, NULL, dw,
                               m_ClearColor,
                               1.0f, 0)))

        return ZFX_FAIL;

    if (m_bIsSceneRunning)
        m_pDevice->BeginScene();
}
/*-----*/

void ZFXD3D::EndRendering(void) {
    m_pDevice->EndScene();
    m_pDevice->Present(NULL, NULL, NULL, NULL);
}

```



```

    m_bIsSceneRunning = false;
}
/*-----*/

void ZFXD3D::SetClearColor(float fRed,
                           float fGreen,
                           float fBlue) {
    m_ClearColor = D3DCOLOR_COLORVALUE(fRed,
                                         fGreen,
                                         fBlue,
                                         1.0f);
}

```

Die Funktion für das Starten der Szene ist auch so flexibel gehalten, dass wir angeben können, welche der vorhandenen Buffer gelöscht werden sollen. Bei vielen Anwendungen, die den gesamten Pixel-Buffer in einem Frame neu rendern, ist es beispielsweise unnötig, den Pixel-Buffer zu löschen. Ebenso können wir auf das Löschen des Stencil-Buffers verzichten, wenn wir diesen gar nicht verwenden oder er eine konstante Maske enthält, die sich im Verlauf des Programms nicht ändert. Hierbei ist aber zu beachten, dass das Flag für den Stencil-Buffer auf keinen Fall gesetzt sein darf, wenn unser Programm diesen nicht verwendet. Das würde zu einem Fehlschlag der DirectX-Funktion `Clear()` führen. Ebenfalls gilt es zu beachten, dass wir die Szene sozusagen kurz unterbrechen müssen, wenn der Benutzer einen oder mehrere der Buffer löschen möchte, während die Szene noch nicht beendet wurde. Um diese Möglichkeit zu überwachen dient das Attribut `m_IsSceneRunning`.

*Löschen der
Buffer*

Die letzte der vier gezeigten Funktionen dient dazu, die Hintergrundfarbe für den Löschvorgang zu ändern, sollte dies nötig werden. Normalerweise ist das aber nur einmal zu Beginn einer Anwendung nötig, um die gewünschte Farbe der Anwendung einzustellen.

Nun haben wir alles für dieses Kapitel komplett, was die Implementierung unseres Interfaces über eine DLL betrifft. Im verbleibenden Teil dieses Kapitels entwerfen wir eine Rahmenanwendung, die die Verwendung unserer bisherigen Arbeit an einem konkreten, lauffähigen Beispiel demonstriert.

3.6 Testlauf der Implementierung

Da wir alle schon einmal mit einer 3D-API gearbeitet haben, wissen wir, wie viel Arbeit es sein kann, diese zu starten, alle möglichen Fehler abzufangen, Bildschirmmodi auszuwählen usw. Halt der ganze Kram, den wir gerade in die DLL gestopft haben. Umso mehr wird uns der Quelltext der nun folgenden Anwendung erschrecken, weil er durch Verwendung der DLL so kurz und einfach gehalten werden kann.

Ein neuer Arbeitsbereich

Wir öffnen also Visual C++ und legen einen neuen Arbeitsbereich für eine *Win32-Anwendung* an. Diese nennen wir schlicht und einfach *Demo* und fügen ihr die neuen Dateien `main.cpp` und `main.h` hinzu. Um Verwirrung zu vermeiden, kopieren wir die nötigen Dateien aus dem Verzeichnis *ZFXRenderer* in das neue Verzeichnis *Demo*. Das ist nur deshalb nötig, weil ich hier keinen festen Ordner auf unserer Festplatte voraussetze, in den wir unsere statische und dynamische Bibliothek sowie die notwendigen Header hinein-erstellt bzw. kompiliert haben. Das wäre aber die praktischere Methode, die jeder zu Hause bei sich anwenden sollte. Kopieren müssen wir also die folgenden Dateien:

- ➔ ZFXRenderer.h
- ➔ ZFXRenderDevice.h
- ➔ ZFXRenderer.lib
- ➔ ZFXD3D.dll

Nun können wir loslegen. In unserem Programm benötigen wir nur vier kurze Funktionen. Eine davon ist natürlich die `WinMain()`-Funktion, also der Eintrittspunkt eines jeden Windows-Programms. Diese geht natürlich Hand in Hand mit einer Callback-Funktion für die der Anwendung automatisch von Windows zugeteilten Nachrichten. Die anderen beiden Funktionen schreiben wir für die Initialisierung und das Beenden des Programms, um unser Render-Device zu erstellen und freizugeben. Das komplette Programm sieht wie folgt aus:



Listing 3.1:

Demo-Applikation
für die Verwendung
der ZFXD3D-DLL

```

////////////////////////////////////
// FILE: main.h
LRESULT WINAPI MsgProc(HWND, UINT, WPARAM, LPARAM);
HRESULT ProgramStartup(char *chAPI);
HRESULT ProgramCleanup(void);
////////////////////////////////////

////////////////////////////////////
// FILE: main.cpp

#define WIN32_MEAN_AND_LEAN

#include "ZFXRenderer.h" // Unser Interface
#include "ZFX.h"         // Rückgabewerte
#include "main.h"        // Prototypen

// Unsere statische Bibliothek einbinden
#pragma comment(lib, "ZFXRenderer.lib")

// Windows-Kram
HWND      g_hWnd = NULL;

```

```

HINSTANCE g_hInst = NULL;
TCHAR     g_szAppClass[] = TEXT("FrameWorktest");

// Anwendungskram
BOOL  g_bIsActive = FALSE;
bool  g_bDone     = false;
FILE  *pLog       = NULL;

// ZFX-Render- und -RenderDevice-Objekte
LPZFXRENDERER  g_pRenderer = NULL;
LPZFXRENDERDEVICE g_pDevice = NULL;

/**
 * WinMain-Funktion als Startpunkt.
 */
int WINAPI WinMain(HINSTANCE hInst,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow) {
    WNDCLASSEX  wndclass;
    HRESULT     hr;
    HWND        hWnd;
    MSG         msg;

    // Fenster-Attribute initialisieren
    wndclass.hIconSm    = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hIcon      = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.cbSize      = sizeof(wndclass);
    wndclass.lpfnWndProc = MsgProc;
    wndclass.cbClsExtra  = 0;
    wndclass.cbWndExtra  = 0;
    wndclass.hInstance  = hInst;
    wndclass.hCursor     = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = g_szAppClass;
    wndclass.style        = CS_HREDRAW | CS_VREDRAW |
                          CS_OWNDC | CS_DBLCLKS;

    if(RegisterClassEx(&wndclass) == 0)
        return 0;

    if (!(hWnd = CreateWindowEx(NULL, g_szAppClass,
                              "Cranking up ZFXEngine...",
                              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                              GetSystemMetrics(SM_CXSCREEN)/2 -190,
                              GetSystemMetrics(SM_CYSCREEN)/2 -140,
                              380, 280, NULL, NULL, hInst, NULL)))
        return 0;

```

```

g_hWnd = hWnd;
g_hInst = hInst;

pLog = fopen("log_main.txt", "w");

// Starte die Engine
if (FAILED( hr = ProgramStartup("Direct3D"))) {
    fprintf(pLog, "error: ProgramStartup() failed\n");
    g_bDone = true;
}
else if (hr == ZFX_CANCELED) {
    fprintf(pLog, "ProgramStartup() canceled\n");
    g_bDone = true;
}
else
    g_pDevice->SetClearColor(0.1f, 0.3f, 0.1f);

while (!g_bDone)
{
    while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    if (g_bIsActive)
    {
        if (g_pDevice->IsRunning()) {
            g_pDevice->BeginRendering(true,true,true);
            g_pDevice->EndRendering();
        }
    }
}

// Cleanup-Stuff
ProgramCleanup();

UnregisterClass(g_szAppClass, hInst);
return (int)msg.wParam;
} // WinMain

/*-----*/

/**
 * MsgProc zur Nachrichtenverarbeitung.
 */
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg,
                        WPARAM wParam,
                        LPARAM lParam) {

    switch(msg) {

```

```

// Anwendungsfokus
case WM_ACTIVATE: {
    g_bIsActive = (BOOL)wParam;
    } break;

// Tastaturereignis
case WM_KEYDOWN: {
    switch (wParam) {
        case VK_ESCAPE: {
            g_bDone = true;
            PostMessage(hWnd, WM_CLOSE, 0, 0);
            return 0;
        } break;
    }
    } break;

// Zerstöre das Fensterobjekt
case WM_DESTROY: {
    g_bDone = true;
    PostQuitMessage(0);
    return 1;
    } break;

default: break;
}
return DefWindowProc(hWnd, msg, wParam, lParam);
}
/*-----*/

/**
 * Erstelle ein RenderDevice-Objekt.
 */
HRESULT ProgramStartup(char *chAPI) {
    HWND hWnd3D[4];
    RECT rcWnd;
    int x=0,y=0;

    // Wir haben noch keine OpenGL-DLL ...
    if (strcmp(chAPI, "OpenGL")==0) return S_OK;

    // Erstelle einen Renderer
    g_pRenderer = new ZFXRenderer(g_hInst);

    // Erstelle ein RenderDevice
    if (FAILED( g_pRenderer->CreateDevice(chAPI) ))
        return E_FAIL;

    // Speichere einen Pointer auf das Device
    g_pDevice = g_pRenderer->GetDevice();
    if(g_pDevice == NULL) return E_FAIL;

```

```

// Größe des Fensterbereichs abfragen
GetClientRect(g_hWnd, &rcWnd);

for (int i=0; i<4; i++) {
    if ( (i==0) || (i==2) ) x = 10;
    else x = rcWnd.right/2 + 10;

    if ( (i==0) || (i==1) ) y = 10;
    else y = rcWnd.bottom/2 + 10;

    hWnd3D[i] = CreateWindowEx(WS_EX_CLIENTEDGE,
        TEXT("static"), NULL, WS_CHILD |
        SS_BLACKRECT | WS_VISIBLE, x, y,
        rcWnd.right/2-20, rcWnd.bottom/2-20,
        g_hWnd, NULL, g_hInst, NULL);
}

// Device initialisieren (Dialogbox anzeigen)
return g_pDevice->Init(g_hWnd, // Hauptfenster
    hWnd3D, // Child-Windows
    4, // 4 Children
    16, // 16-Bit-Z-Buffer
    0, // 0-Bit-Stencil
    false);
} // ProgramStartup
/*-----*/

/**
 * Freigabe der verwendeten Ressourcen.
 */
HRESULT ProgramCleanup(void) {
    if (g_pRenderer) {
        delete g_pRenderer;
        g_pRenderer = NULL;
    }
    if (pLog)
        fclose(pLog);
    return S_OK;
} // ProgramCleanup
/*-----*/

```

**Mit dem Render-
Device arbeiten**

Hierbei handelt es sich ja um wenig mehr als eine typische minimale Windows-Anwendung. Das kleine Bisschen mehr, was wir hier haben, findet fast alles in der Funktion `ProgramStartup()` statt. Als Erstes erzeugen wir eine Instanz der Klasse `ZFXRenderer` aus unserer statischen Bibliothek. Dieser Konstruktor macht noch nicht viel. Dann rufen wir aber die Initialisierungsfunktion für unser Render-Device, also `ZFXRenderer::CreateDevice`, des Renderers auf. Der Renderer lädt daraufhin die passende DLL (hier `ZFXD3D.dll`) und erzeugt eine Instanz der Klasse `ZFXD3D`, also das eigentliche

Render-Device. Auf diese Instanz lassen wir uns dann einen Pointer geben, den wir global speichern, um ihn überall im Programm verfügbar zu haben.

Dann basteln wir uns automatisiert vier Child-Windows, die gleichmäßig im Hauptfenster angeordnet sind (siehe Abbildung 3.2). Im letzten Schritt rufen wir die Initialisierungsfunktion für das Render-Device auf. Diese Funktion bringt den Dialog für die Auswahl der Grafikkarte und deren Modi zur Anzeige.

Child-Windows

Sobald der Benutzer den Dialog abgehandelt hat, läuft das Programm weiter. In der Hauptschleife löschen wir einfach so lange den Bildschirm in jedem Frame, bis der Benutzer das Programm durch Drücken von `ESC` beendet oder im Fenster-Modus auf das kleine Kreuzchen oben rechts im Fenster klickt.

Wenn man das Programm nun startet, erscheint zunächst das Fenster, dann der Dialog. Dieser benötigt eine kleine Weile, um die vorhandene Hardware zu untersuchen, um sie in den Steuerelementen anzuzeigen. Jetzt kann der Benutzer einen Bildschirmmodus und eine Grafikkarte (falls mehrere vorhanden sind) auswählen und das Programm starten. Im Fenster-Modus werden die vier Child-Windows angezeigt, im Fullscreen-Modus natürlich lediglich ein nackter Bildschirm, der das Hauptfenster verwendet.

3.7 Ein Blick zurück, zwei Schritt nach vorn

Rückblickend kann man sagen, dass wir in diesem Kapitel unsere Grundkenntnisse in verschiedenen Bereichen der Programmierung wieder haben aufleben lassen. Zudem hoffe ich, dass wir unsere Kenntnisse in einigen Bereichen, speziell was den Umgang mit DLLs angeht, auch erweitern konnten. Hier haben wir es geschafft, uns die Basis für eine DLL zu schaffen, in der wir die Verwendung einer API kapseln können. Diese DLL, die nun unsere eigene ZFX-Grafik-API ist, beliebig zu erweitern stellt auch kein Problem mehr dar. An dem Code zum Laden und zur Verwendung der DLL ändert sich nichts mehr.

Solide Basis

Ein weiterer Punkt, der für viele neu gewesen sein wird, ist das flexible Handling von beliebig vielen Child-Windows, die wir gleich im Initialisierungsaufwurf unseres Devices angeben können. Durch diese eine Zeile Quelltext erhalten wir die Möglichkeit, beispielsweise die vier typischen Fenster (von rechts, von vorne, von oben und 3D-Ansicht) eines Modell/Level-Editors anzulegen. Zusätzlich können wir auch noch ein Child-Window anlegen, in dem wir dann ausgewählte Texturen anzeigen oder eine Preview eines einfügbaren 3D-Objekts usw. Später werden wir diesen flexiblen Ansatz noch weiter ausbauen, indem wir in jedem Child-Window beliebig viele Viewports zulassen werden. Während die Child-Windows lediglich

Flexibilität

nebeneinander angeordnet sein können, bieten die Viewports die Möglichkeit, Fenster in einem Fenster anzulegen, also beispielsweise einen kleinen Bereich in der 3D-Ansicht, der eine andere Ansicht der Szene zeigt – beispielsweise die Sicht eines vorgeschobenen Aufklärers oder einen Rückspiegel.

*Vorteile durch
Verwendung einer
eigenen Grafik-
API*

Welchen großen Vorteil wir durch die Verwendung einer eigenen Grafik-API, hier basierend auf Direct3D, erhalten, das zeigt unsere kleine Demo-Applikation. Wir benötigen lediglich vier kurze Funktionsaufrufe, um beispielsweise Direct3D zu starten. Dabei sind wir über unseren Dialog noch immer variabel genug, die API im Fenster-Modus oder im Fullscreen-Modus bei beliebiger Auflösung zu starten, ohne das irgendwo im Programm *hart verdrahten* zu müssen. Damit verschwindet der ganze hässliche Initialisierungs- und Auswahl-Kram komplett aus dem Quelltext einer Anwendung, was deutlich zu deren Übersichtlichkeit beiträgt. Und wenn eine neue DirectX-Version auf den Markt kommt? Auch das ist jetzt kein Problem mehr für uns. Die Zeiten, in denen wir Tausende Zeilen von Quellcode nach API-abhängigem Code durchsuchen mussten, sind endgültig vorbei. Wir laden einfach den Arbeitsbereich mit der DLL und ändern deren Code entsprechend, ohne die Funktionsprototypen anzufassen, die durch das Interface vorgeschrieben sind. Dann kompilieren wir die DLL neu und können sie sofort mit unseren fertig kompilierten anderen Projekten verwenden, ohne diese irgendwie verändern zu müssen.

Und jetzt?

Nun ist es an der Zeit, dass wir die Welt der Grafik-APIs verlassen. Dort draußen wartet noch etwas anderes Unheimliches auf uns, und zwar die 3D-Mathematik. Wenn wir unabhängig von einer Grafik-API bleiben wollen, müssen wir natürlich auch unsere eigenen Funktionen für 3D-Mathematik implementieren. Aus dieser Not werden wir jedoch eine Tugend machen, und diese Implementierung so umfangreich und komfortabel machen, dass wir uns die Arbeit an einer 3D-Engine ohne sie gar nicht mehr werden vorstellen können.

4 Schnelle 3D-Mathematik

»Gib mir nur einen Punkt, wo ich hintreten kann, und ich bewege die Erde.« (Archimedes)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Grundlagen der Assembler-Programmierung
- ➔ erweiterte Befehlssätze neuer CPUs (MMX, 3DNow!, SSE, SSE2)
- ➔ Identifikation von Prozessoren
- ➔ prozessoroptimierte Vektor- und Matrizenrechnung
- ➔ Kollisionsabfragen und Culling mit Strahlen, Ebenen, Polygonen und Bounding-Boxen
- ➔ Grundlagen von Quaternions

4.1 Schnell, schneller, am schnellsten

Dank moderner 3D-Grafik-APIs ist es heutzutage vergleichsweise einfach, eine 3D-Engine zu programmieren. Die schnelle Grafik-Hardware sorgt des Weiteren dafür, dass eine solche Engine zunächst unheimlich schnell wirkt. Das ist jedoch nur so lange der Fall, bis die Engine richtig etwas zu tun bekommt und mal mehr als nur ein paar Tausend Triangles in einem Frame rendern muss. Dieses *Mehr* sind nämlich hauptsächlich Berechnungen, die zu Lasten der CPU (Central Processing Unit), also des Hauptprozessors, gehen. War es noch vor ein bis zwei Jahren so, dass die Füllrate der Grafikkarte die Geschwindigkeit eines 3D-Programms limitierte, so stellt sich die Situation heute etwas diffiziler dar. Früher berechnete die schnelle CPU fix alle Daten und schickte dann Texturen und Dreiecke zum Rendern an die Grafikkarte. Dann musste die CPU warten, bis die Grafikkarte mit dem Rendern eines Frames fertig war. Erst dann konnten die Berechnungen an einem Frame weitergehen.

Heute, da die Grafikkarten auch schon die Transformation und Beleuchtung vornehmen und leistungsstarke eigene GPUs (Graphic Processing Unit) haben, ist die Situation eher andersherum. Selbst eine schnelle CPU hat

heute Mühe, die notwendigen Daten schnell genug zu berechnen, um die Grafikkarte konstant mit Daten zu füttern. Während die CPU also fleißig rechnet, sitzt die Grafikkarte da und wartet auf Daten. Der Bottleneck heutiger 3D-Programme liegt damit zu großen Teilen in zwei Bereichen: Zum einen kann es sein, dass die CPU so viele Daten zu berechnen hat, dass die GPU nicht gut ausgelastet ist. Zum anderen ist es so, dass die berechneten Daten, um aus dem System-RAM des Computers zu der Grafikkarte zu kommen, über den vergleichsweise langsamen Bus des Computers zum Video-RAM der Grafikkarte geschickt werden müssen.

Das zweite Problem kann man unter anderem dadurch beheben, dass man so viele der Daten wie möglich eben nicht über den Bus schickt. Hier bieten das Culling, statische Vertex-Buffer und Vertex-Shader die Möglichkeit, nicht sichtbare Daten gleich zu ignorieren und benötigte Daten bereits im VRAM der Grafikkarte zu halten bzw. nach dem Transfer der Vertexdaten über den Bus gewisse Attribute erst nachträglich zu erzeugen. Beispielsweise könnte man die Beleuchtung und Schattierung für vorbeleuchtete Vertices oder die Texturkoordinaten durch einen Shader erzeugen. So spart man für jeden über den Bus wandernden Vertex eine `DWORD`-Variable für die Lichtfarbe sowie zwei oder mehr `float`-Werte für die Texturkoordinaten. Damit hat man je Vertex eine Ersparnis von $3 \cdot 32 = 96$ Bit. Bei mehreren Tausend Vertices je Frame macht sich das durchaus bemerkbar, und der Bus wird es uns danken.

Die Auslastung der CPU hingegen kann nicht immer so einfach reduziert werden. In der CPU müssen oftmals viele aufwändige Berechnungen durchgeführt werden, die für das Culling, Kollisionsabfragen, die Bewegung der Objekte, die künstliche Intelligenz und vieles mehr zuständig sind. Der Programmierer ist hier aufgefordert, möglichst effiziente Algorithmen zu finden, die ohne viel Rechenaufwand gute Ergebnisse erzielen. Bevor man sich also an die Lowlevel-Optimierung von Programmcode macht, sollte man immer zuerst die verwendeten Algorithmen optimieren. Wenn auch das nicht den gewünschten Erfolg bringt, kann man mit der Lowlevel-Optimierung der wichtigsten Stellen im Programm beginnen. Bleiben also zwei Fragen: Was sind Lowlevel-Optimierungen, und wo sind die wichtigsten Stellen im Programm? Nun, Lowlevel-Optimierungen bestehen darin, dass man zeitkritische Stellen im Quelltext nicht mehr durch eine Hochsprache, wie beispielsweise C++, implementiert, sondern direkt in Assembler. Während man sich in einer Hochsprache auf einer sehr abstrakten Ebene befindet, in der lediglich der Ablauf des Programms von Bedeutung ist, stellt Assembler eine sehr maschinennahe Sprache dar. Ein Prozessor erwartet die Befehle für das, was er tun soll, als Bitmuster im dualen Zahlensystem – also quasi als Abfolgen von Strom-An- und Strom-Aus-Anweisungen. Die Sprache Assembler bietet nun für alle Befehle, die ein Prozessor unterstützt, also für die Bitmuster, Bezeichnungen. Diese nennt man Mnemo-Code.

Das Wort Mneme stammt aus dem Griechischen und bezeichnet die Erinnerung oder das Gedächtnis. Die Mnemonik, oder auch Mnemotechnik, ist die Disziplin, in der man versucht, das Gedächtnis durch Unterstützung von (technischen) Hilfsmitteln zu unterstützen.



Versucht man also in den Hochsprachen, eine mehr oder weniger dem alltäglichen Sprachgebrauch ähnliche Sprache zu entwickeln, dient Assembler nur dazu, direkt in der Sprache des Prozessors zu reden und ihm Anweisungen zu geben, die er direkt verstehen kann. Vereinfacht gesagt, kann man so den Strom direkt in den Registern eines Prozessors verschieben. Die Programmierung in Assembler hat in der Regel einen großen Geschwindigkeitsvorteil gegenüber der Programmierung in einer Hochsprache. Das ist jedoch wiederum differenzierter zu sehen.

Eine Hochsprache wird durch einen Compiler in Maschinensprache übersetzt. Je nach Compiler kann dies mehr oder weniger effektiv geschehen. Die Preisunterschiede der verschiedenen Entwickler-Tools oder deren Versionen, beispielweise *Visual C++ Standard* und *Professionell*, beruhen darauf, dass die teureren Programme einen wesentlich besser optimierten Maschinencode erzeugen. Ein vollkommen identisches C++-Programm kann daher um einiges schneller laufen, wenn man es mit der *Professionell*-Version kompiliert hat. Kommen wir nun zum traditionellen Assembler zurück. Wenn man mit einem guten Compiler arbeitet, sind die Geschwindigkeitsvorteile, die man durch die Programmierung in Assembler erzielt, nicht mehr so eindeutig. Aufgewogen gegen den Nachteil, dass Assembler schwerer zu überblicken und zu erlernen ist als eine Hochsprache, und wegen des damit zwangsweise verbundenen erhöhten Wartungsaufwands verzichtet man heute in der Regel darauf, Programme in Assembler zu schreiben. Lediglich an zeitkritischen Stellen von Programmen verwendet man in die Hochsprache eingebetteten Assembler – so genannten *Inline*-Assembler.

Eben erwähnte ich ja, dass Assembler quasi nur Makros für die Bitbefehle eines Prozessors zur Verfügung stellt. Da wirft sich doch wie von selbst die Frage auf, ob sich an dem Befehlssatz der Prozessoren nichts ändert? Und ob sie nur deshalb immer schneller werden, weil ihre Taktfrequenz immer größer wird? Und wie optimiert man eigentlich einen Prozessor für 3D Grafik? All diese Fragen wird der nächste Abschnitt beantworten.

Grundlagen der Assembler-Programmierung

Natürlich ist dies kein Buch, in dem die Programmierung mit Assembler im Vordergrund steht. Ebenso wenig kann ich hier die Grundlagen der Assembler-Programmierung vermitteln, für die es ganze Bücher mit Hunderten von Seiten gibt. Dennoch müssen wir uns wenigstens ein wenig damit auskennen, um unsere Engine in kritischen Bereichen extrem zu beschleunigen. Im Folgenden werde ich daher einen kurzen Überblick über die Assembler-

Programmierung geben, der als Einstieg in das Thema und zum Verständnis des später entwickelten Codes ausreichen sollte. Ich empfehle aber dringend, ein weiteres Buch zur Assembler-Programmierung als Lektüre für lauschtige Abendstunden.

Prozessor- Aufbau

Ein Prozessor stellt bautechnisch betrachtet wenig mehr dar als ein paar Speicherstellen für Daten. Diese Speicherstellen bezeichnet man als Register. Durch bestimmte Anweisungen ist es dem Programmierer möglich, Daten zwischen den verschiedenen Registern zu verschieben und ihre Werte zu verändern. Der Vorteil von Assembler ist, dass man mit genügend Kenntnissen des internen Aufbaus eines Prozessors und der Ablaufdauer der Instruktionen ein Programm besser optimieren kann als ein Compiler, der eine Hochsprache kompiliert. Die Ressourcen, die ein Prozessor dem Programmierer zur Verfügung stellt, sind auch heute noch sehr begrenzt. Für allgemeine Aufgaben stellt ein Prozessor lediglich acht Register zur Verfügung, die jeweils einen 32 Bit großen Wert (also Bits 0 bis 31, die jeweils den Wert 0 oder 1 haben, daraus resultieren bei 32 Bit 2^{32} verschiedene Werte) aufnehmen können. Die folgende Übersicht listet diese Register mit ihren Namen auf:

- ➔ EAX (*Extended Accumulator*)
- ➔ EBX (*Extended Base*)
- ➔ ECX (*Extended Counter*)
- ➔ EDX (*Extended Data*)
- ➔ ESI (*Extended Source Index*)
- ➔ EDI (*Extended Destination Index*)
- ➔ EBP (*Extended Base Pointer*)
- ➔ ESP (*Extended Stack Pointer*)

Die Namen dieser Register sind historisch gewachsen und hatten früher eine konkrete Bedeutung – beispielsweise der *Extended Accumulator* EAX (früher in der 16-Bit-Version nur *Accumulator* AX genannt). Heutzutage sind diese speziellen Funktionen aber so gut wie gänzlich verwaschen, und alle diese Register können gleichwertig als Speicheradressen für Daten verwendet werden. Lediglich die Register EBP und ESP sollte man besser in Ruhe lassen, weil diese direkt auf dem Stack des Computers arbeiten und dessen Werte verändern. Wer also nicht ganz genau weiß, was er tut, der hält sich von den beiden besser fern. Anderenfalls riskiert man im besten Fall eine Veränderung von Daten auf dem Stack, also lokaler Variablen oder Funktionsparameter usw. Mit Ausnahme dieser beiden Register können wir also alle anderen sechs heutzutage für jegliche Operationen verwenden, die wir über Assembler durchführen können.

Heap und Stack

Vereinfacht dargestellt teilt der Computer seinen Speicher in zwei Bereiche auf: den **Heap** und den **Stack**. Der Heap ist dabei der Speicherbereich, den der Programmierer verwalten kann. Jede globale Funktion oder Variable wird dort an einer fixen Adresse gespeichert, wozu ihr Datentyp und damit ihre Größe bekannt sein muss (daher kann man auch keine Arrays mit Variablen als Größenangabe definieren).

Der Stack (Stapel) hingegen wird vom Prozessor verwaltet. Hier werden Speicheradressen und Werte zur Programmlaufzeit temporär abgelegt. Immer dann, wenn das Programm aus einer Funktion in eine andere Unterfunktion springt, wird im Stack die Rücksprungadresse vermerkt, also die Stelle im Speicher, an die das Programm nach Bearbeitung der Unterfunktion zurückkehren soll. Ebenso werden alle lokalen Variablen auf dem Stack erzeugt. Dazu zählen auch die Parameter der Funktion, deren Werte aus der aufrufenden Funktion kopiert werden. Nach Bearbeitung der Funktion wird der Rückgabewert gegebenenfalls auch über den Stack an die aufrufende Stelle zurückgegeben. Danach ist sämtlicher Speicher, der in der Unterfunktion verwendet wurde, wieder freigegeben und damit ungültig.

Es ist auch heute noch möglich, lediglich auf die *unteren* 16 Bits (0 bis 15) dieser Register zuzugreifen, indem man die alten Bezeichnungen wie beispielsweise AX verwendet. Auch hier kann man wieder zwischen den unteren 8 Bits (0 bis 7) und den oberen 8 Bits (8-15) des AX-Registers unterscheiden. Diese werden dann entsprechend AL und AH genannt, wobei das L für *low* und das H für *high* steht. In der Regel werden wir uns aber hier darauf beschränken, mit kompletten 32-Bit-Registern zu arbeiten.

Die Register sind allerdings nur ein Teil des Prozessors. Sie gehören zu der CPU, die nur dazu da ist, Integer-Werte, also Ganzzahlen, zu verarbeiten. Das sind insbesondere arithmetische Operationen (Addition, Subtraktion usw.), logische Vergleiche, bedingte Sprünge und die Datenverschiebung zwischen den Registern. Zu den grundlegendsten und am häufigsten verwendeten Operationen gehört dabei die MOV-Anweisung, die den Inhalt eines Registers in ein anderes kopiert. Eine Zeile eines Assembler-Programms könnte also wie folgt aussehen:

CPU

```
MOV EDI, ESI ;kopiere von ESI nach EDI
```

Eine Zeile Assembler-Code ist immer nach demselben Schema aufgebaut. An erster Stelle steht die Anweisung, was gemacht werden soll. Danach folgt die Angabe des Registers, in dem das Ergebnis der Anweisung gespeichert wird, und abschließend folgt die Angabe des Registers, auf das die Anweisung angewendet wird. Ein Semikolon leitet einen Kommentar ein. Die folgende Zeile wäre also eine Addition des Inhaltes von Register ESI zu

dem Inhalt des Registers EDI mittels der Anweisung `ADD`. Das Register EDI enthält dann die Summe, der Inhalt von ESI bliebe unverändert:

```
ADD EDI, ESI ;addiere ESI zu EDI
```

FPU Ein anderer Teil eines Prozessors ist seit dem 486er die FPU, also die **Floating Point Unit** für Fließkommazahlen. Früher auf einem eigenständigen Co-Prozessor realisiert, wurde die Fließkommarechnung mit Einführung der FPU direkt in die CPU integriert. Mit der FPU wollen wir uns hier nicht weiter beschäftigen. Lediglich ihre acht Register, die als R0 bis R7 bezeichnet werden, schauen wir uns kurz an, da sie später noch von Interesse sein werden. Um auch mit Vorzeichen und vielen Kommastellen möglichst genau rechnen zu können, wurden die Register der FPU für einen einzigen Datentyp ausgelegt, und zwar für den 80 Bit großen Typ *Extended Real*. Im Vergleich zu den alten 16-Bit-Registern der CPU ermöglichten diese Register eine extrem große Präzision bei der Fließkommarechnung. Auf heutigen 32-Bit-Betriebssystemen rechnet man ja üblicherweise mit dem Fließkomma-Datentyp `float` für einfache Präzision, der *nur* 32 Bit groß ist.

Dies sollte als kleine Einführung in die Assembler-Programmierung an theoretischen Grundlagen genügen. Später in diesem Kapitel werden wir auch wirklich praktisch mit Assembler kleine Programme schreiben und sehen, dass das gar nicht so schwer ist. Bevor wir so weit sind, müssen wir uns aber noch mit ein paar Ergänzungen zu dem althergebrachten Assembler befassen.

Einführung in SIMD

Begriff Der Begriff SIMD ist ein Akronym für **Single Instruction Multiple Data**. Damit sagt der Name auch schon genau das aus, was die SIMD-Technologie so einzigartig macht. Anfangs hatte ich die Frage aufgeworfen, wie man bei der Assembler-Programmierung verschiedenen neueren Prozessor-Generationen Rechnung tragen kann und wie man einen Prozessor darauf optimieren kann, 3D-Grafik zu verarbeiten. Die Antwort auf diese Frage lautet heutzutage schlicht und einfach: SIMD.

SIMD-Prinzip In normalem Assembler arbeiten die Prozessoren in serieller Abfolge. Das bedeutet, dass die Befehle von dem Prozessor nacheinander bearbeitet werden. In jedem Takt kann eine Anweisung mit einem Wert durchgeführt werden. Eine Möglichkeit, die Leistung eines Prozessors zu steigern, wäre also das Erhöhen der Taktrate, damit die serielle Bearbeitung einfach nur schneller erfolgt. Eine ganz andere Möglichkeit ist die Parallelisierung gleichartiger Berechnungen. Der SIMD-Ansatz dient also dazu, mit einer Instruktion des Prozessors multiple Daten parallel zu verändern. Man kann sich das so vorstellen, dass eine Instruktion eines SIMD-Programms nicht nur ein Register eines Prozessors verändert, sondern mehrere gleichzeitig. Normaler

Assembler kann dagegen immer nur auf einem Register arbeiten. Er müsste eine Instruktion daher wiederholt auf verschiedene Register nacheinander anwenden. Ganz so ist es technisch gesehen natürlich nicht, aber so kann man sich das vorstellen. Die SIMD-Instruktionen arbeiten nämlich auch nur mit jeweils einem Register. Der Trick dabei ist jedoch der, dass diese Register quasi eine Überlänge haben und mehrere Variablen gleicher Länge aufnehmen können. Die SIMD-Instruktion behandelt ein entsprechendes Register dann wie ein Array von einzelnen Werten und wird parallel auf jeden der einzelnen Werte angewendet.

Beispielsweise könnte man sich ein SIMD-Register vorstellen, das drei `float`-Werte aufnehmen kann – einen Vektor im 3D-Raum. Nun lädt man zwei Vektoren in zwei solche Register und kann durch eine SIMD-Instruktion beispielsweise die beiden Vektoren addieren. Die drei einzelnen Additionen der jeweiligen Koordinaten werden dabei parallel, und nicht wie bei herkömmlichem Assembler seriell, berechnet. Da wir uns hier mit 3D-Grafik beschäftigen, erkennen wir sofort den großen Vorteil, den das für uns hat: Beinahe das gesamte mathematische Grundgerüst einer 3D-Engine arbeitet mit mehrkomponentigen Daten. Unsere Vektoren haben drei Dimensionen, sie benötigen aber noch eine vierte Dimension als Hilfsstütze für die Multiplikation mit Matrizen. Daher ist die kleinste Einheit, die wir in der 3D-Grafik kennen, grob gesagt eine Struktur aus vier `float`-Werten. SIMD macht es nun möglich, dass man grundlegende mathematische Operationen, wie beispielsweise arithmetische Operationen (Grundrechenarten), Wurzelziehen, Minimum/Maximum-Berechnungen oder eine Datenverschiebung in den Registern des Prozessors gleichzeitig auf vier `float`-Werte anwenden kann – und nicht mehr nur nacheinander. Dies beschleunigt die grundlegendsten Vektorrechnungen (wie beispielsweise Punktprodukt, Kreuzprodukt oder Längenberechnung von Vektoren) um ein Vielfaches, verglichen mit einer reinen C/C++-Implementierung und ca. um den Faktor zwei, verglichen mit einer Assembler-Implementierung.

*Anwendungen für
SIMD*

Aber bevor wir so weit sind, müssen wir noch eine geschichtliche Hürde nehmen. Die kryptischen Akronyme MMX, 3DNow!, SSE und SSE2, die der eine oder andere sicherlich schon einmal gehört oder gelesen hat, sind allesamt mehr oder weniger verschiedene Formen von SIMD. Dabei ist SIMD quasi nur die Beschreibung für die Technik an sich, bei den anderen Begriffen handelt es sich um konkrete Formen von SIMD, die auf verschiedenen Prozessoren zur Verfügung stehen. Eben hatte ich erwähnt, dass es mit SIMD möglich ist, vier `float`-Werte in einem Register parallel zu bearbeiten. Dies ist aber nur bei 3DNow! (den aktuellsten Versionen davon), SSE und SSE2 möglich. MMX hatte zunächst eine andere Zielsetzung.

Historie

Multimedia-Extensions, MMX

Die erste Erweiterung für Multimedia-Anwendungen unter Berücksichtigung von SIMD in der Assembler-Programmierung war MMX (*Multi-*

Media-Extension). Wie oben erwähnt, werden SIMD-Instruktionen zwar auch nur auf ein Register angewendet, jedoch enthält dieses Register nicht nur den Wert für eine Variable, sondern für mehrere. Zu diesem Zweck definiert MMX verschiedene neue Datentypen, die eine Größe von jeweils 64 Bit haben. Der Sinn und Zweck dieser Datentypen ist es, nicht mehr nur eine Variable aufzunehmen und mit deren Daten zu arbeiten, sondern Felder von Daten. In einem solchen 64-Bit-Datentyp kann man sozusagen ein Array von zwei 32-Bit-Werten ablegen. Das ist an sich nichts sensationell Neues. Neu ist daran nur, dass die MMX-Anweisungen wissen, dass sich in einem Register eben nicht ein 64-Bit-Wert befindet, sondern zwei 32-Bit-Werte, auf die die Anweisung dann parallel angewendet wird. So erreicht man in einem Anweisungsschritt, wozu man in reinem Assembler zwei Schritte benötigen würde.

Alte neue Register

Ja ... das Problem ist aber, dass ein Prozessor normalerweise nur 32 Bit große Register zur Verfügung stellt. Für die MMX-Anweisungen war es aber nötig, 64 Bit große Register zu verwenden. Um eine Neueinführung von Registern zu umgehen, wurden von *Intel* die bereits vorhandenen acht Register der FPU für MMX zweckentfremdet. Die MMX-Anweisungen arbeiten also mit 64-Bit-Datenstrukturen auf den unteren acht Byte der 10 Byte großen FPU-Register. Daraus ergibt sich auch die Beschränkung, dass man FPU und MMX-Register nicht gleichzeitig verwenden kann, weil es eben genau dieselben sind. Als Aliase für die *neuen* Register wurden die Bezeichnungen MM0 bis MM7 eingeführt.

MMX-Datentypen

Die 64 Bit großen Datentypen, die MMX neu eingeführt hat, erfüllen verschiedenste Funktionen. Darauf möchte ich nicht allzu detailliert eingehen, da MMX heutzutage nur noch eine untergeordnete Rolle spielt. Allen Datentypen ist aber gemeinsam, dass es sich dabei um Integer-Werte und nicht etwa um Fließkommazahlen handelt. Beispielsweise gibt es das *Short Packed Double Word* (Array von zwei 32-Bit-DWORD-Werten) und das *Short Packed Word* (Array von vier 16-Bit-WORD-Werten). Arbeitet man mit einem Short-Packed-Double-Word-Wert, so weiß MMX sozusagen, dass es sich dabei um eine Datenstruktur aus zwei Werten handelt, und kann die Anweisungen entsprechend für beide Werte in demselben Register durchführen. Ebenfalls zu bemerken ist, dass die Kontrolle der FPU-Register bei Verwendung von MMX-Anweisungen durch die CPU gesteuert wird, nicht etwa durch die FPU.

MMX-Anweisungen

Jetzt möchte ich noch kurz auf die Anweisungen eingehen, die MMX zur Verfügung stellt. Man sollte immer im Hinterkopf behalten, dass MMX eine veraltete Technologie ist, jedenfalls was den Bereich der 3D-Grafik angeht. Hier gibt es bereits weit fortschrittlichere Varianten von SIMD, zu denen wir gleich kommen. Wir werden also nur dann etwas mit MMX zu tun bekommen, wenn entweder alter Quellcode in unsere Hände fällt oder wenn wir kompatibel zu veralteten Prozessor-Modellen bleiben wollen.

Aktueller ist beispielsweise AMDs 3DNow!-Technologie (ab K6-2) und Intels SSE-Technologie ab (*Pentium III*). Als Beispiel für eine Anweisung von MMX nehmen wir erneut die Addition zweier Register. Gehen wir einmal davon aus, dass die beiden Register jeweils die Datenstruktur Packed Double Word enthalten. Um nun diese beiden Register komponentenweise zu addieren, gibt es die Anweisung `PADDQ`. Das Präfix `P` weist darauf hin, dass mit gepackten Daten (packed data), also den in Arrays gepackten Daten, gearbeitet wird. Das Suffix `Q` bei der `ADD`-Anweisung bezeichnet hier den genauen MMX-Datentyp, der verwendet wird: in diesem Fall das `DWORD`.

```
PADDQ MM1, MM0
```

Diese Anweisung addiert also komponentenweise den Inhalt des Registers `MM0` in das Register `MM1`. In reinem C++-Code würde das in etwa wie folgt aussehen:

```
// gegeben: DWORD dwMM0[2], dwMM1[2];
Add_PackedDoubleWords() {
    dwMM1[0] = dwMM0[0] + dwMM1[0];
    dwMM1[1] = dwMM0[1] + dwMM1[1];
}
```

Das Speichern der Daten und die eigentliche Addition für ein zwei-elementiges Array kann MMX also in einer einzigen Anweisung ausführen. Das ist bereits um ein Vielfaches schneller als der reine C++-Code und auch noch schneller als ein entsprechendes reines Assembler-Programm, das dafür zwei Anweisungen benötigen würde. Eine andere oft gebrauchte Anweisung ist `MOV`, also das Verschieben von Daten zwischen den Registern. Da MMX aber nur Datentypen der Länge 64 Bit kennt, benötigt man hier spezielle Befehle. Um Daten der Länge 64 Bit zwischen den MMX-Registern oder dem Speicher und einem MMX-Register zu kopieren, verwendet MMX die Anweisung `MOVQ`. Es kann allerdings auch vorkommen, dass man von einem MMX-Register nur 32 Bit ansprechen möchte. Die Anweisung `MOVD` bewegt folglich nur einen 32 Bit großen Bereich (`DWORD`) aus einem oder in ein MMX-Register. Sie wird beispielsweise verwendet, um eine Konstante oder einen Wert aus einem der Standard-Register der CPU in ein MMX-Register zu schreiben oder in letzterem Fall auch umgekehrt.

Abschließend sei erwähnt, dass die MMX-Technologie zunächst von *Intel* in die *Pentium*-Prozessoren integriert wurde. Als Nächstes sehen wir uns an, wie *AMD* die SIMD-Operationen in seine Prozessoren integriert hat.

AMD dreidimensional, 3DNow!

Nachdem *Intel* mit MMX die erste Implementierung der SIMD-Philosophie eingeführt hatte, legte auch *AMD* nach. Vom Grundprinzip her übernahm *AMD* einfach den MMX-Ansatz, also die Verwendung der jeweils ersten

*Fließkomma statt
Integer*

acht Byte der FPU-Register R0 bis R7 unter den neuen Bezeichnungen MM0 bis MM7. Allerdings machte AMD bereits den Sprung von den Integer-Werten zu den Fließkommazahlen. Im Bereich der grafikintensiven Multimedia-Anwendungen genügten Integer-Werte zwar für Berechnungen beispielweise auf Pixel-Ebene. Aber für die 3D-Grafik waren diese vollkommen ungenügend. Während Intel in der nächsten Generation SSE erschuf (dazu gleich mehr), verwendete AMD einfach die vorhandenen MMX-Register, allerdings für Fließkommazahlen einfacher Genauigkeit. Diese Technologie wurde als 3DNow! bezeichnet.

Bei 3DNow! hat der Programmierer die Möglichkeit, eine Struktur aus zwei float-Werten parallel zu bearbeiten, so wie es bei MMX beispielsweise mit zwei 32-Bit-Integer-Werten der Fall ist. Ebenfalls analog zu MMX haben die Anweisungen für arithmetische Operationen bei 3DNow! das Präfix P, was auf *packed data*, also Werte in Datenstrukturen, hinweist. Hinzu kommt nun noch ein zweites Präfix, nämlich F für *floating-point values*. Die Anweisung PFADD addiert beispielsweise den Inhalt zweier MMX-Register analog zu der Anweisung PADD – nur handelt es sich bei den einzelnen Komponenten eben nicht um Integer-, sondern um Fließkommawerte.

Die zweite Generation

Die 3DNow!-Technologie fand mit dem K6 Eingang in die AMD-Prozessoren. Allerdings wurden diverse Anweisungen im Laufe der Zeit ergänzt. Insbesondere stehen nun auch Operationen für Integer-Datentypen zur Verfügung, so dass die erweiterte 3DNow!-Technologie, die ab dem Athlon-Prozessor Verwendung fand und offiziell *Enhanced 3DNow!* heißt, die gleiche Funktionalität in diesem Bereich erfüllen kann wie MMX.

Ausführlicher möchte ich das Thema 3DNow! an dieser Stelle nicht behandeln, da wir für die ZFXEngine SSE verwenden werden. Weiterführende Informationen findet ihr auf der Internetseite von AMD.¹ Für die Pragmatiker unter Euch empfehle ich die Internetseite von David Scherfgen, der ein deutsches Tutorial über die Programmierung mit 3DNow! geschrieben hat.²

Intels Streaming-SIMD-Extensions, SSE

Nun wissen wir bereits eine ganze Menge über SIMD, also über die parallele Bearbeitung identischer Berechnungen auf verschiedenen Variablen gleichen Typs. Daher können wir bei der einleitenden Beschreibung von SSE entsprechend knapp vorgehen. Einer der Nachteile von MMX war, dass man dort nur mit Integer-Werten arbeiten konnte. In der 3D-Grafik rechnen wir aber in der Regel mit Fließkommazahlen einfacher Präzision, also mit float-Werten. Ein anderer Nachteil von MMX war, dass man dort nur maximal zwei 32-Bit-Komponenten mit einer Anweisung bearbeiten konnte. In der 3D-

1 www.amd.com

2 <http://www.scherfgen-software.net/>

Grafik arbeiten wir aber mit vierdimensionalen Vektoren. Zur Erinnerung: Für Rotations- und Skalierungsoperationen benötigen wir 3x3-Matrizen, wenn wir diese in einer einzigen Matrix zusammenfassen wollen. Kommt jedoch die Translation noch zu den Transformationen hinzu, so steht diese in der vierten Zeile einer 4x4-Matrix. Daher rechnet man in der 3D-Grafik mit 4x4-Matrizen und 4D-Vektoren, wobei die vierte Komponente des Vektors immer 1 ist. Anderenfalls könnten wir einen 3D-Vektor nicht mit einer 4x4-Matrix multiplizieren.

Uns 3D-Programmierern wäre es also am liebsten, wenn wir quasi als kleinste Einheit einen 4D-Vektor zur Verfügung hätten. Und nun die gute Nachricht: SSE bietet uns genau das. *Intel* hat mit der Einführung des *Pentium III*-Prozessors acht komplett neue Register in den Prozessor integriert. Und nicht nur das: Diese Register sind jeweils 128 Bit groß. Nach Adam Riese passen damit in ein einziges solches Register genau vier 32 Bit große Werte. Erfreulicherweise sind diese Register auch dafür gedacht, mit Fließkommazahlen einfacher Genauigkeit zu rechnen (oder Packed Single Reals, wie *Intel* sie nennt). Anders ausgedrückt, kann ein solches 128-Bit-Register ein Array von vier `float`-Werten aufnehmen, die komponentenweise bearbeitet werden können. Nennen wir das Kind doch beim Namen: Ein `float-Array[4]` ist ein 4D-Vektor. Diese neuen Register tragen übrigens die Bezeichnungen XMM0 bis XMM7. Mit ihnen ist es nun möglich, bei paralleler Bearbeitung gleichzeitig vier Werte zu verändern, nicht mehr nur zwei wie bei MMX oder gar einen wie bei reinem Assembler oder den Hochsprachen. Allerdings besteht auch die Möglichkeit, die Daten *skalar* zu bearbeiten. Das bedeutet in diesem Kontext, dass nur einer der vier möglichen `float`-Werte in einem XMM-Register, und zwar der in den Bits 0 bis 31, bearbeitet wird.

Neue Register

Schauen wir uns wieder als Beispiel die Anweisung für die Addition von Werten an. Hier gibt es insbesondere die beiden Anweisungen `ADDPS` und `ADDSS`. Die erste Anweisung addiert komponentenweise *packed single precision float*-Werte. Die zweite Anweisung addiert *scalar single precision float*-Werte, also nur den ersten 32-Bit-`float`-Wert in den betroffenen Registern. Das wird etwas deutlicher, wenn wir uns ansehen, wie das im SSE-Assembler-Code und im Pseudo-Code aussehen würde:

SSE-Anweisungen

```
ADDPS XMM1, XMM0    ; addiert zwei 4D-Vektoren
ADDSS XMM1, XMM0    ; addiert nur die x-Komponente
```

Und als Pseudo-Code:

```
// gegeben: float fXMM0[4], fXMM1[4];

Add_PackedFloats() {
    fXMM1[0] = fXMM0[0] + fXMM1[0];
    fXMM1[1] = fXMM0[1] + fXMM1[1];
```

```

    fXMM1[2] = fXMM0[2] + fXMM1[2];
    fXMM1[3] = fXMM0[3] + fXMM1[3];
}

Add_ScalarFloats() {
    fXMM1[0] = fXMM0[0] + fXMM1[0];
}

```

*aligned und not
aligned*

Kommen wir nun zum Kopieren der Daten in die XMM-Register bzw. zwischen den normalen 32-Bit-Registern und den 128-Bit-XMM-Registern. Auch hier gibt es eine Abart des Befehls `MOV ...` na ja, eigentlich sind es mehrere Abarten. Wenn wir schnell Daten von einem 32-Bit-Register in ein XMM-Register verschieben wollen oder dasselbe genau andersherum, dann verwenden wir den Befehl `MOVAPS`, der für **Move Aligned Packed Single-Precision Floating Point** steht. Das Wörtchen »Aligned« (ausgerichtet) deutet dabei an, dass die Daten im Speicher nach einem bestimmten Schema ausgerichtet sein müssen. Konkret gesagt müssen die Daten im Speicher des Computers an Adressen stehen, die glatt durch 16 teilbar sind, sie müssen 16 *Byte aligned* sein. Diese 16 Byte entsprechen nämlich genau den 128 Bit, die in so ein XMM-Register passen. Beim Kopiervorgang der Daten ist die Speicheradressierung damit vereinfacht. Eine solche Ausrichtung der Daten erreicht man, indem man die entsprechenden Variablen, aus denen oder in die man mit dem XMM-Register kopieren möchte, als `__declspec(aligned(16))` deklariert. Sollte der Compiler diese Anweisung nicht korrekt umsetzen, dann wird das Programm abstürzen, wenn man eine SSE-Anweisung mit dem Suffix `APS` verwendet.

Hat man seine Daten aus programmiertechnischen Gründen nicht an 16-Byte-Grenzen orientiert, dann lässt einen SSE auch nicht im Stich. Hier gibt es das Suffix `UPS`, also beispielsweise die Anweisung `MOVUPS` für **Move Unaligned Packed Single-Precision Floating Point**. Damit kann man nicht ausgerichteten Speicher für ein XMM-Register referenzieren. Diese Anweisung ist langsamer als `MOVAPS`, denn sie muss zusätzlich noch den Speicher ausrichten, um ihn sauber in das XMM-Register kopieren zu können bzw. um ihn aus dem XMM-Register in einen nicht-ausgerichteten Speicherbereich zu kopieren.

TIPP

Selbst wenn man mit nicht ausgerichteten Daten arbeitet, so sind die Daten auf alle Fälle innerhalb der XMM-Register immer als ausgerichtet zu betrachten. Wenn man Daten zwischen zwei XMM-Registern verschiebt, kann man also immer die schnelleren Anweisungen für ausgerichtete Daten verwenden, beispielsweise `MOVAPS` statt `MOVUPS`.

MOVAPS MOVUPS

Die folgenden Beispiele in C-Syntax verdeutlichen den Umgang mit den zwei verschiedenen `MOV`-Anweisungen von SSE:

```
// gegeben sind die folgenden beiden float-[4]-Arrays,
// die zwei Vektoren repräsentieren, deren Felder
// bereits mit Werten gefüllt sind:
// __declspec(align(16)) float fVektor_A[4];
//                               float fVektor_U[4];
// Aufruf; SSE_Add(&fVektor_A[0], &fVektor_U[0]);
```

```
void SSE_Add(float *fA, float *fU) {
    __asm {
        MOV     esi,    fA
        MOV     edi,    fU
        MOVAPS  XMM0,   [esi]
        MOVUPS  XMM1,   [edi]
        ADDPS   XMM0,   XMM1
        MOVAPS  XMM1,   XMM0
        MOVAPS  [esi],  XMM0
        MOVUPS  [edi],  XMM1
    }
}
```

Über den normalen MOV-Befehl werden zunächst die Speicheradressen der beiden als Call-by-Reference übergebenen float-Arrays in die Register ESI und EDI geschrieben. Danach wird der Inhalt im Speicherbereich an diesen Adressen in die Register XMM0 und XMM1 kopiert, jeweils über eine Länge von 128 Bit. Man beachte die Verwendung von MOVAPS und MOVUPS. MOVUPS kann grundsätzlich immer sicher verwendet werden, MOVAPS nur dann, wenn die Daten auch wirklich 16-Byte-ausgerichtet sind. Diese beiden XMM-Register werden dann komponentenweise addiert, und der Ergebnisvektor wird in XMM0 gespeichert. Das Ergebnis kopieren wir jetzt nach XMM1 und dann jeweils aus dem entsprechenden XMM-Register zurück in die Adressen, auf die ESI und EDI zeigen, also die beiden float-Parameter. Das Kopieren der Daten zwischen den XMM-Registern ist für den Zweck der Funktion eigentlich überflüssig, man hätte auch aus dem Register XMM0 das Ergebnis einmal nach ESI und einmal nach EDI kopieren können. Hier sieht man aber, dass die Daten in den XMM-Registern immer als aligned behandelt werden können. Wichtig ist nur, dass man in diesem Beispiel auf EDI nur mit Anweisungen arbeiten kann, die nicht-ausgerichtete Daten behandeln können. Als Ergebnis liefert diese Funktion eine Addition der beiden Vektoren, die auch jeweils in beiden Vektoren gespeichert wird.

Zur Datenmanipulation bietet SSE noch diverse weitere Abarten des MOV-Befehls. Diese werde ich im Folgenden kurz vorstellen, damit man sie einmal gesehen hat. Allzu häufig werden wir sie aber nicht brauchen.

Weitere MOVs

➔ MOVSS

Kopiert lediglich einen 32-Bit-Wert in oder aus den untersten Bits 0 bis 31 eines XMM-Registers.

➔ MOVLP

Kopiert lediglich 64 statt 128 Bit aus den unteren bzw. in die unteren (L=low) 64 Bit eines XMM-Registers. Dieser Befehl kann zum Kopieren zwischen XMM-Registers verwendet werden.

➔ MOVHP

Analog zu MOVLP, nur werden hier die oberen 64 Bits (H=high) verwendet.

➔ MOVLHP

Der Datenaustausch der oberen und unteren 64 Bits innerhalb eines oder zwischen zwei XMM-Registers wird über diesen Befehl geregelt. Er kopiert low to high, also die unteren Bits des Quellregisters in die oberen Bits des Zielregisters.

➔ MOVHLP

Analog zu MOVLHP werden hier 64 Bits ausschließlich zwischen XMM-Registers kopiert (nicht mit anderen Registers oder dem Speicher). Allerdings wird hier high to low, also die oberen zu den unteren Bits, kopiert.

Shufflen

Nun kommen wir zu dem letzten SSE-Befehl, auf den ich hier kurz eingehen möchte. Wenn wir diesen Befehl beherrschen, besitzen wir bereits genug Wissen, um die elementarsten Vektor- und Matrizen-Operationen in schnellem SSE-Code implementieren zu können. Es geht um die Anweisung zum Shufflen (Mischen).

Wir werden in unseren Programmen öfter an Stellen kommen, wo wir die Inhalte eines XMM-Registers, also die vier `float`-Werte, gern beliebig vertauschen würden – oder gar einen Austausch beliebiger Komponenten zwischen zwei XMM-Registers vornehmen würden. Zu diesem Zweck gibt es die Anweisung `SHUFPS` (*Shuffle Packed Single-Precision Floating Point*). Durch diese Anweisung kann man die vier Elemente des Ziel-Registers (ein XMM-Register) neu belegen. Die ersten beiden Elemente (Bits 0-31 und 32-63) werden dazu mit je einem beliebigen der vier Werte aus dem Zielregister (dasselbe XMM-Register) belegt. Die oberen beiden Elemente (Bits 64-95 und 96-127) werden mit je einem beliebigen der vier Werte aus einer Quelle (dasselbe oder ein anderes XMM-Register oder Speicherbereich) belegt. Das klingt sehr verwirrend, also sehen wir uns ein Beispiel dazu an:

```
SHUFPS XMM0, XMM1, 9C
```

Das hilft uns auch nicht viel weiter, bevor wir nicht noch ein paar andere Informationen haben. Welche Register wohin geschuffelt werden, wird durch eine Hexadezimalzahl bestimmt (hier 9C, wobei das kleine h für Hexadezimal steht). Damit wir diese Hexadezimalzahl richtig interpretieren können, benötigen wir zunächst eine Umrechnung von dem Hexadezimalsystem in das Dualsystem; das zeigt uns die Tabelle 4.1.

hexadezimal	dual	hexadezimal	dual
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Tabelle 4.1:
Umwandlung von
Hexadezimal- in
Dualzahlen

Wenn wir unsere Hexadezimalzahl 9Ch in eine Dualzahl umrechnen, erhalten wir folgende Darstellung:

10011100

Aus dieser Darstellung können wir ablesen, welches Element welchen Registers wohin geschuffelt wird. In dieser Darstellung stehen immer zwei Dualzahlen für ein Element eines `float`-Arrays [4]. Dabei steht die Dualzahl 00 für das Element [0], die Dualzahl 01 für das Element [1], die Dualzahl 10 für das Element [2] und die Dualzahl 11 für das Element [3]. In der obigen Darstellung sehen wir also, welches Element wohin kommt. Die Position eines Elementes in dieser Dualzahl sagt nämlich aus, in welches Element des Zielregisters es kopiert wird. Nun müssen wir aber bedenken, dass der Computer bei dem Least Significant Bit (LSB) mit dem Zählen beginnt, und das ist in dieser Darstellung von rechts und nicht von links. Das Shufflen mit 9Ch mit XMM0 und XMM1 ergibt also Folgendes:

- ➔ Element [0] von XMM0 erhält Element 00=[0] von XMM0
- ➔ Element [1] von XMM0 erhält Element 11=[3] von XMM0
- ➔ Element [2] von XMM0 erhält Element 01=[1] von XMM1
- ➔ Element [3] von XMM0 erhält Element 10=[2] von XMM1

Auf den ersten Blick ist das immer noch verwirrend, aber wenn man ein wenig damit herumspielt, dann sollte es schnell klarer werden. Die Dualzahl-Darstellung des hexadezimalen Shuffle-Wertes zeigt quasi den gewünschten Inhalt des Zielregisters nach dem Shufflen an (die Bits 0-127 von rechts nach links gesehen). Je zwei Stellen der Dualzahl stehen für ein 32 Bit großes Element. Zu beachten ist allerdings, dass die unteren beiden Elemente des Ziel-

registers auch mit Elementen aus dem Zielregister selbst belegt werden. Nur die oberen beiden Elemente kann man mit Elementen eines anderen Registers belegen.

Broadcast Die *SHUFPS*-Anweisung kann allerdings auch dazu verwendet werden, nur in einem Register die Werte zu mischen. Dann ist das Zielregister gleich dem Quellregister. Durch geschicktes Shufflen ist es möglich, die Werte innerhalb eines Registers beliebig zu vertauschen. Es ist auch möglich, den Inhalt eines Elements für alle anderen Elemente dieses Registers zu setzen. Diese Operation nennt sich *Broadcasting*. Schauen wir uns folgende Anweisung an:

```
SHUFPS XMM0, XMM0, 00h
```

Der Shuffle-Wert 00h entspricht der Dualdarstellung 00000000. So erhält jedes der vier Elemente von XMM0 denselben Wert, den das Element [0] von XMM0 vor dem Shufflen enthielt. Das *Broadcasten* mit dem Wert AAh würde dafür sorgen, dass alle vier Elemente des Registers den Wert des Elements [2] erhalten.

Intels Streaming-SIMD-Extensions, die Zweite: SSE2

Mit der Einführung des *Pentium IV*-Prozessors hat *Intel* die SIMD-Fähigkeiten seiner Prozessoren noch ein wenig erweitert. Da SSE2 hier keine weitere Rolle spielen wird, fasse ich mich kurz. Im Grunde genommen, wurde SSE dazu entwickelt, mit 128 Bit großen Strukturen für Fließkommazahlen zu arbeiten, während MMX für 64 Bit große Strukturen mit Integerzahlen gedacht war.

Die Erweiterungen von SSE auf SSE2 basieren nun hauptsächlich darauf, dass der Prozessor nun auch 128 Bit große Datenstrukturen für Integerzahlen in den XMM-Registern unterstützt. Die gesamte Palette der Anweisungen, die wir von MMX her kennen, ist nun auch auf den XMM-Registern für Integerzahlen verfügbar.

AMD dreidimensionaler, 3DNow!-Professional

Ab den neueren Versionen des *Athlon*-Prozessors, dem *Athlon XP*, hat *AMD* seine 3DNow!-Technologie wieder erweitert, und diesmal recht drastisch. War 3DNow! vorher noch eine Art MMX mit zusätzlicher Fähigkeit für Fließkommazahlen, so wurden nun ebenfalls wie bei *Intel* komplett neue 128-Bit-Register eingeführt. Der Befehlssatz für diese neuen Register ist identisch und kompatibel mit dem von *Intels* SSE-Anweisungen. Diese neue Form seiner SIMD-Technologie bezeichnet *AMD* als 3DNow!-Professional.

Bis zu diesem Zeitpunkt war es für Anwendungen möglich, eine von drei Strategien bei der Optimierung von Code-Teilen für SIMD anzuwenden:

1. Optimierung des kritischen Codes lediglich für 3DNow! oder Enhanced 3DNow!-Technologie, falls die Zielmaschine einen AMD-Prozessor (mindestens einen K6) hatte. Alternativ Verwendung eines nicht-SIMD-optimierten Codes, falls ein anderer Prozessor auf der Zielmaschine vorhanden war.
2. Einzelne Implementierung jeweils spezifisch für 3DNow!, Enhanced 3DNow! und andere SIMD-Technologien, wie beispielsweise SSE und SSE2, in jeweils getrennten Unterfunktionen. Auswahl einer geeigneten Unterfunktion je nach vorhandenem Prozessor auf der Zielmaschine.
3. Optimierung kritischer Code-Teile nur für andere Technologien, beispielsweise SSE und SSE2, aber nicht für 3DNow! oder Enhanced 3DNow!. Alternativ Verwendung nicht-SIMD-optimierten Codes, falls ein anderer Prozessor auf der Zielmaschine vorhanden war.

Mit der kompletten SSE-Unterstützung unter 3DNow!-Professional ist es möglich, dass auch ein AMD-Prozessor als SSE-fähiger Prozessor erkannt wird und im Szenario Nummer 3 der optimierte Code ausgeführt wird. Voraussetzung dafür ist aber, dass der Prozessor der Zielmaschine unabhängig von dem Hersteller auf SSE-Fähigkeit hin befragt wird. Zur Identifikation der verschiedenen Prozessoren kommen wir gleich. Zuerst müssen wir aber noch eine andere Frage klären.

Wie sag ich's meinem Compiler?

Eine relativ große Hürde auf dem Weg zur Verwendung von SIMD-Technologien (egal ob nun MMX, 3DNow! und seine Brüder, SSE oder SSE2) stellt der Compiler dar. Es ist gar nicht so einfach herauszubekommen, welche Ausrüstung man benötigt, um selbst SIMD-Code programmieren und kompilieren zu können. Mit der *Visual C++ 6.0 Standard Edition* ist dies beispielsweise gar nicht möglich. Der integrierte Assembler der Standard-Version kann nicht um die zusätzlichen Befehlssätze für SIMD erweitert werden, jedenfalls bietet *Microsoft* diese Option meines Wissens nach nicht an. Verfügt man entweder über die *Professional* oder die *Enterprise Edition* von *Visual C++ 6.0*, so funktioniert dies – unter der Voraussetzung, dass man mindestens das Service Pack 4 und das Processor Pack zusätzlich installiert hat. Beides gibt es auf der Internetseite³ von *Microsoft* kostenlos zum Download. Eine andere Möglichkeit ist die Verwendung von *Visual C++ .NET* (gibt es nur in der *Standard Edition*, dort funktioniert das aber) oder *Visual Studio .NET Professional* oder *Enterprise*. Bei diesen drei

*Kompatible
Compiler*

³ msdn.microsoft.com/vstudio/downloads/VisualStudio6.asp

Optionen ist es auch nicht erforderlich, Zusatzpacks zu installieren. Für MMX, SSE und SSE2 bietet *Intel* natürlich auch einen eigenen Compiler an.

Andere Optionen

Eine andere Möglichkeit, SIMD-Befehlssätze verwenden zu können, stellen Makros dar, die die entsprechenden Anweisungen direkt durch Binärcodierungen ersetzen. Im Internet finden sich an verschiedenen Stellen, unter anderem in einem älteren Artikel auf Gamasutra⁴, solche Hilfsimplementierungen. Diese benötigen dann lediglich einen normalen Assembler oder Inline-Assembler, um das Programmieren für eine SIMD-Technologie zu ermöglichen. Diese Variante ist aber voller Tücken und ist auch nicht problemlos zum Laufen zu bekommen. Wer ernsthaft eine schnelle 3D-Engine programmieren möchte, der sollte sich einen entsprechend kompatiblen Compiler kaufen. Wer weiterhin mit einem anderen Compiler arbeiten möchte, der kann die im Folgenden entwickelten Programme dieses Kapitels allerdings nicht kompilieren. Wir werden hier (nur in diesem Kapitel) mit SSE arbeiten, jedoch wird jede Funktion alternativ eine Implementierung in reinem C++ bieten. Entsprechend muss man dann bei Bedarf die SSE-relevanten Code-Teile auskommentieren oder löschen, weil der Compiler sonst Fehler melden würde.

Weitere Hürden

Es sei auch noch erwähnt, dass nicht nur der Prozessor die vom Programm verwendete SIMD-Technologie unterstützen muss, sondern auch das Betriebssystem. Unter *Windows 98* und neueren Versionen sollte es damit aber keine Probleme geben.

Identifikation einer CPU

Verschiedene SIMD-Ansätze auf verschiedenen Prozessoren

Jetzt kennen wir eine ganze Reihe von SIMD-Befehlssätzen, die leider in der Regel sehr verschieden sind. Wir haben hier einmal MMX, mit dem wir heutzutage in der 3D-Grafik kaum etwas anfangen können. Seit dem Erscheinen von SSE2 verliert MMX auch im Bereich der Integerzahlen seine Bedeutung, weil SSE2 hier mit größeren Zahlenbereichen oder mehr Komponenten in einer Datenstruktur arbeiten kann. Es stehen 128 Bit gegen 64 Bit. Dann hätten wir noch 3DNow!, das es in drei verschiedenen Ausprägungen gibt, und natürlich noch SSE und SSE2. Je nach Prozessor stehen uns verschiedene dieser Ansätze zur Verfügung. Daraus resultiert für uns das Problem, dass wir zur Laufzeit bei der Initialisierung des Programms feststellen müssen, welcher Prozessor auf der Zielmaschine läuft und welchen SIMD-Ansatz er unterstützt. Das erinnert an die Zeiten, in denen man noch für jede Grafikkarte eigene Routinen programmieren musste.

SIMD unserer Wahl

Wir werden in diesem Buch zwar lediglich Funktionen speziell für SSE entwickeln, aber ich zeige hier auf alle Fälle, wie man die verschiedenen Prozessoren identifizieren kann. Unser Ansatz zur Implementierung sieht dabei wie folgt aus: Wir entwerfen Funktionen, die wir für die 3D-Mathematik

4 www.gamasutra.com/features/wyatts_world/19990528/pentium3_01.htm

benötigen, einmal in ihrer reinen, langsamen C++-Form. Dann implementieren wir dieselbe Funktionalität in Assembler unter Verwendung des SSE-Befehlssatzes. Steht ein SSE-kompatibler Prozessor zur Verfügung (*Intel* ab *Pentium III* und *AMD* ab *Athlon XP*), dann werden die schnellen Assembler-Funktionen verwendet. Ist das nicht der Fall, dann tritt unser Kontingenzplan in Kraft, und wir verwenden die langsamen C++-Funktionen. Wer eine wirklich schnelle Engine haben möchte, der wird nicht umhin kommen, auch Unterstützung für die älteren 3DNow!-Ansätze zu implementieren und den reinen C++-Code durch reinen Assembler ohne SIMD zu ersetzen.

Nachdem das geklärt ist, müssen wir nur noch herausfinden, welcher Prozessor zur Laufzeit zur Verfügung steht. Glücklicherweise sind *Intel* und *AMD* so freundlich, entsprechende Informationen über die Identifikation ihrer Prozessoren auf ihren Internetseiten zur Verfügung zu stellen.⁵ Durch die Suchfunktionen auf diesen Internetseiten findet man einen Haufen von technischen Artikeln über die Features der verschiedenen Prozessoren und darüber wie man diese überprüfen kann. Wir beginnen unsere Arbeit mit dem Entwurf einer Struktur, die die notwendigen Informationen über einen Prozessor speichern kann.

*Informations-
beschaffung*

```
// in: ZFX3D.h
typedef struct CPUINFO_TYP {
    bool bSSE;           // Streaming SIMD Extensions
    bool bSSE2;          // SStreaming SIMD Extensions 2
    bool b3DNOW;         // 3DNow! (vendor independant)
    bool bMMX;           // MMX support
    char name[48];        // cpu name
    bool bEXT;           // extended features available
    bool bMMXEX;         // MMX (AMD specific extensions)
    bool b3DNOWEX;       // 3DNow! (AMD specific extensions)
    char vendor[13];     // vendor name
} CPUINFO;
```

Jetzt können wir eine Funktion schreiben, die eine CPU daraufhin untersucht, ob sie die in der Struktur aufgelisteten Features unterstützt. Nun ist es so, dass jeder Prozessor seit dem 386er eine spezielle Anweisung zur Verfügung stellt. Diese nennt sich *CPUID*. Man füllt zuerst einen bestimmten Wert in das Register EAX und ruft dann diese Anweisung auf. Abhängig von dem Wert, den man in EAX abgelegt hat, führt die *CPUID*-Anweisung verschiedene Anfragen an die CPU durch und liefert verschiedene Werte in verschiedene Register zurück. Diese Werte können wir dann auf bestimmte Merkmale hin untersuchen. Hier zwei kleine Beispiele: Wenn wir den Wert 0 in das Register EAX schreiben, dann liefert die *CPUID*-Anweisung den Namen des Prozessor-Herstellers (Vendor) zurück. Dieser Name darf maximal 12 Zeichen umfassen, und je vier davon werden durch die Anweisung

CPUID-Anweisung

⁵ AMD Processor Recognition; Publication #20734; 2002 Intel Processor Identification; Order No. 241618-021

in die Register EBX, EDX und ECX geschrieben. Schreiben wir jedoch den Wert 1 in EAX und rufen dann die `CPUID`-Anweisung auf, dann schreibt diese einen Bitstring in das Register EDX. Dieser Bitstring stellt eine Feature-Liste der CPU dar. Bei *Intel*-Prozessoren wird zusätzlich eine so genannte Brand ID in das Register EBX geschrieben, bei *AMD* wird diese in das Register EAX geschrieben.

Feature-Liste

Den Bitstring mit der Feature-Liste der CPU können wir nun auf verschiedene Werte hin testen. Jeder Prozessor-Hersteller muss hier identische Flags für bestimmte Features setzen. Indem wir also prüfen, ob bestimmte Bits in diesem Bitstring gesetzt sind, können wir den Prozessor auf verschiedene Features hin testen. In unserem Fall interessieren uns drei Features:

- ➔ SSE-Unterstützung
- ➔ SSE2-Unterstützung
- ➔ MMX-Unterstützung

Extended Features

Neben diesen Standard-Features gibt es auch erweiterte Features (Extended-Features). Der Unterschied zu der normalen Feature-Liste ist, dass die CPU unter Umständen nicht einmal eine Liste von Extended-Features zur Verfügung stellt. Also prüfen wir den Bitstring für die Extended-Features noch einmal extra. Dazu rufen wir wieder die Anweisung `CPUID` auf, allerdings nun mit einem Wert von `0x80000000` in dem EAX-Register. Daraufhin schreibt die Anweisung einen Wert in das EAX-Register, den wir daraufhin überprüfen müssen, ob er größer als `0x80000000` ist. Ist das der Fall, verfügt die CPU über eine Liste mit Extended-Features. Um allerdings an diese Liste heranzukommen, müssen wir den Wert `0x80000001` in das EAX-Register schieben und die `CPUID`-Anweisung nochmals aufrufen. Diese schreibt uns dann den gewünschten Bitstring mit der Extended-Features-Liste in das Register EDX. Doch was sind nun diese Extended-Features? Nun, bei grundsätzlich allen Prozessoren kann es sich dabei um die Unterstützung von 3DNow! handeln. Dies ist ein offener Standard, den jeder Hersteller verwenden könnte. Bei *Intel*-Prozessoren brauchen wir keine anderen Extended-Features mehr zu testen. Im Falle eines *AMD*-Prozessors gibt es jedoch noch zwei andere Dinge. Zum einen bietet *AMD* einen erweiterten MMX-Befehlssatz an, und zum anderen einen erweiterten 3DNow!-Befehlssatz.

Welche Bits?

Nun wissen wir schon alles, was wir über die Identifikation und die Untersuchung einer CPU wissen müssen. Die entsprechenden Bit-Werte, die im Bitstring der Features- und Extended-Features-Liste überprüft werden müssen, finden sich ebenfalls in den technischen Dokumenten der jeweiligen Hersteller. Die folgende Funktion befüllt ein Objekt unserer Datenstruktur mit den entsprechenden Daten:



```

CPUINFO GetCPUInfo() {
    CPUINFO info;
    char *pStr = info.vendor;
    int n=1;
    int *pn = &n;

    // alles auf 0 (=false) setzen
    memset(&info, 0, sizeof(CPUINFO));

    // 1: Herstellername, SSE2, SSE, MMX testen
    __try {
        __asm {
            mov  eax, 0          // Herstellername
            CPUID                // CPUID-Anweisung

            mov  esi,    pStr
            mov  [esi],  ebx      // erste 4 Chars
            mov  [esi+4], edx     // folgende 2 Chars
            mov  [esi+8], ecx     // letzte 4 Chars

            mov  eax, 1          // Feature-Liste
            CPUID                // CPUID-Anweisung

            test edx, 04000000h   // teste SSE2
            jz   _NOSSE2         // springe, falls negativ
            mov  [info.bsSE2], 1 // true

_NOSSE2: test edx, 02000000h   // teste SSE
            jz   _NOSSE         // springe, falls negativ
            mov  [info.bsSE], 1  // true

_NOSSE:  test edx, 00800000h   // teste MMX
            jz   _EXIT1         // springe, falls negativ
            mov  [info.bMMX], 1  // true
_EXIT1:  // fertig
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER) {
        if ( _exception_code() ==
            STATUS_ILLEGAL_INSTRUCTION )
            return info;        // CPU inaktiv
        return info;           // unerwarteter Fehler
    }

    // 2: Teste auf ExtendedFeatures
    __asm {
        mov  eax, 80000000h     // Extended Features?
        CPUID                  // CPUID-Anweisung
        cmp  eax, 80000000h     // > als 0x80?
        jbe  _EXIT2            // springe, falls negativ
    }
}

```

```

        mov [info.bEXT], 1      // true

        mov  eax, 80000001h     // Feat-Bits nach EDX
        CUID                     // CPUID-Anweisung
        test edx, 80000000h     // teste 3DNow!
        jz   _EXIT2             // springe, falls negativ
        mov  [info.b3DNOW], 1   // true
_EXIT2: // fertig
    }

    // 3: Hersteller-abhängiger Kram
    //   INTEL: CPU id
    //   AMD:   CPU id, 3dnow_ex, mmx_ex
    if ( (strcmp(info.vendor, "GenuineIntel", 12)==0)
        && info.bEXT) {        // INTEL
        _asm {
            mov  eax, 1         // Feature-Liste
            CUID                 // CPUID-Anweisung
            mov  esi,  pn       // Brand-ID
            mov  [esi], ebx
        }
        int m=0;
        memcpy(&m, pn, sizeof(char)); // nur untere 8 Bits
        n = m;
    }
    else if ( (strcmp(info.vendor, "AuthenticAMD",
        12)==0) && info.bEXT) { // AMD

        _asm {
            mov  eax, 1         // Feature-Liste
            CUID                 // CPUID-Anweisung
            mov  esi,  pn       // CPU-Typ
            mov  [esi], eax

            mov  eax, 0x80000001 // Ext.Feat. Bits
            CUID                 // CPUID-Anweisung

            test edx, 0x40000000 // AMD extended 3DNow!
            jz   _AMD1           // Sprung bei Fehler
            mov  [info.b3DNOWEX], 1 // true
_AMD1: test edx, 0x00400000     // AMD extended MMX
            jz   _AMD2           // springe, falls negativ
            mov  [info.bMMXEX], 1 // true
_AMD2:
        }

    }

    else {
        if (info.bEXT)
            ; /* UNBEKANNTER HERSTELLER */
    }

```

```
else
    ; /* KEINE Extended-Features-Liste */
}

info.vendor[13] = '\\0'
GetCPUName(info.name, n, info.vendor)
return info;
}
```

Diese Funktion unterteilt sich in drei Abschnitte. Zuerst speichern wir den Namen des Herstellers. Im Falle von *Intel* lautet dieser String `GenuineIntel`, und im Falle von *AMD* lautet er `AuthenticAMD`. Dann testen wir den Standard-Features-Bitstring auf die Unterstützung von SSE2, SSE und MMX. Fällt einer dieser Tests negativ aus, dann überspringen wir immer eine Zeile, in der wir die entsprechende Variable der Struktur auf `true` setzen würden. Im zweiten Abschnitt testen wir, ob die Extended-Features-Liste vorhanden ist. Falls ja, dann merken wir uns das und prüfen den Prozessor auf die Unterstützung von 3DNow!. Im dritten Abschnitt müssen wir dann erstmals zwischen verschiedenen Herstellern unterscheiden. Im Falle von *Intel* speichern wir ja nur die Nummer, die den Prozessor-Typ angibt. Haben wir es mit einem *AMD* zu tun, dann speichern wir auch dessen Prozessor-Typ-Nummer (die allerdings in einem anderen Register als bei *Intel* steht) und testen noch auf erweiterte Befehlssätze für MMX und 3DNow!.

Am Ende der Funktion müssen wir noch das Terminal-Zeichen '\\0' an den Namen des Herstellers anfügen, da dies in den 12 `char`-Werten in dem Prozessor ja nicht vorhanden ist. Würden wir dies nicht tun, so würde der Computer nicht wissen, wo der String zu Ende ist. Jetzt müssen wir nur noch eins tun, nämlich die Funktion `GetCPUName()` aufrufen. Bisher haben wir ja für den Prozessor-Typ nur eine hersteller-abhängige Nummer. Diese Nummer wollen wir noch in einen sprechenden Namen umwandeln. Diese Funktion werde ich hier jedoch nicht zeigen, sie ist aber auf der CD-ROM enthalten. Nach den beiden unterstützten Herstellern getrennt, handelt es sich dabei nur um eine große `switch`-Anweisung, die den jeweiligen Nummern einen String wie beispielsweise »Pentium IV« oder »Athlon« zuweist.

Die Identifikation eines bestimmten Prozessors ist bei AMD einfacher als bei Intel. Im Falle von Intel kann es also sein, dass das Programm eine bestimmte CPU nicht korrekt identifiziert, weil ich dort einen kürzeren, aber unsichereren Weg zu Demonstrationszwecken verwendet habe. Die Angaben über die Verfügbarkeit der einzelnen Features sind aber auf jeden Fall korrekt.

!!
STOP

Unterstützung für SSE zur Laufzeit überprüfen

*Betriebssystem
mit SSE-Support?*

Wenn nun eine CPU ein bestimmtes Feature unterstützt, dann ist das leider nur die halbe Miete. Bei SSE ist es nämlich so, dass auch das Betriebssystem diese Technologie unterstützen muss. Das Betriebssystem hat hier nämlich die Aufgabe, die States (Zustände) der neuen Fließkommazahlen-Register sichern und wiederherstellen zu können. Es gibt hier einen einfachen, brutalen Weg, um dies herauszufinden. Man führt einfach eine SSE-Anweisung aus (wenn man SSE-Unterstützung in der CPU erkannt hat) und schaut nach, ob dies zu einer Ausnahme im Programm führt. Ist dies der Fall, dann fängt man sie ab und weiß, dass das Betriebssystem SSE nicht unterstützt. Das sieht dann so aus:



```
bool OSSupportsSSE() {
    __try
    {
        __asm xorps xmm0, xmm0
    }
    __except(EXCEPTION_EXECUTE_HANDLER) {
        if ( _exception_code() ==
            STATUS_ILLEGAL_INSTRUCTION )
            return false; // sse nicht unterstützt
        return false;    // unerwarteter Fehler
    }
    return true;
}
```

Alle zusammen

Diese kleine Mini-Funktion liefert uns die erwünschte Auskunft. Jetzt können wir alles eben Erlernte zusammenfassen und eine kleine Funktion schreiben, die true oder false liefert, je nachdem, ob wir SSE tatsächlich auf einem Computer einsetzen können. Dazu muss SSE ja von der CPU und dem Betriebssystem eingesetzt werden können.



```
bool g_bSSE = false;

bool ZFX3DInitCPU(void) {
    CPUINFO info = GetCPUInfo();
    bool    bOS  = OSSupportsSSE();

    if (info.bSSE && bOS) g_bSSE= true;
    else g_bSSE = false;

    return g_bSSE;
}
```

Geschafft!

Jetzt haben wir es endlich geschafft. Nun können wir mit der Funktion ZFX3DInitCPU() zur Laufzeit entscheiden, ob wir SSE-Funktionalität nutzen können. Und nun die schlechte Nachricht: Das war lediglich die Grundlage,

die wir für dieses Kapitel brauchen. Jetzt können wir uns an den Hauptteil der Arbeit machen, nämlich an das Entwerfen einer Bibliothek für 3D-Mathematik und mehr. Diese soll in einer statischen Bibliothek alles enthalten, was wir bei der Arbeit mit einer 3D-Engine benötigen. Also auf in den Kampf!

Für dieses Kapitel erstellen wir ein Visual C++-Projekt für eine statische Bibliothek. Diese nennen wir `zfx3d.lib`. Sie wird unsere gesamte 3D-Mathematik (und noch ein wenig mehr) abdecken, ebenso wie die Identifikation des Prozessors und seiner Fähigkeiten, die wir eben besprochen haben.



4.2 Rechnen mit Vektoren

Die grundlegendste Einheit in unserer virtuellen 3D-Welt ist ein Vektor. Eigentlich handelt es sich dabei um nicht viel mehr als eine Struktur aus drei Fließkommazahlen. In jeder der drei Dimensionen des Raumes hat der Vektor eine bestimmte Ausdehnung, die auch 0 sein kann. Prinzipiell kann ein Vektor n -dimensional sein, wobei n eine natürliche Zahl ist. Ein eindimensionaler Vektor hat beispielsweise nur eine Komponente, und dabei handelt es sich mehr oder weniger um die Zahlen, die wir auch im normalen Alltag verwenden. Stellen wir uns eine eindimensionale Welt vor. Damit würden die Wesen, die in dieser Welt leben, jede Position in ihrem Raum mit einer Angabe beschreiben können, da ihre Welt ja nur eine Art von Linie ist. Eine Position lässt sich dann eindeutig als beispielsweise [5] beschreiben. Das ist dann genau bei der Stelle 5 nach dem Nullpunkt der Linie.

Was ist ein Vektor?

In einer zweidimensionalen Welt leben die entsprechenden Wesen nicht mehr auf einer Linie, sondern auf einer unendlich ausgedehnten Fläche. Ein zwar etwas hinkender, dafür aber anschaulicher Vergleich ist unsere Erde. Wenn wir die Höheninformationen ignorieren, kann man jede Position auf der Erde durch die Angabe von zwei Werten eindeutig definieren, nämlich durch die Höhen- und Breitengrade. Rein zweidimensionale Wesen müssten nicht mehr wissen als die Werte, die sie in Relation zum Nullpunkt (nullter Höhen- und Breitengrad) auf den Höhen- und Breitengraden laufen müssten, um diese Position zu erreichen. Ein zweidimensionaler Vektor ist also in so einer Welt vollkommen ausreichend. Wir arbeiten aber in einer dreidimensionalen Welt, daher benötigen wir zur eindeutigen Identifizierung einer bestimmten Position genau drei Angaben: die Länge, die Breite und die Höhe der Position.

n -dimensional

In unserem Kontext sind solche Vektoren eine Struktur aus drei `float`-Werten für Länge, Breite und Höhe einer Position. Ein Vektor ist also nichts weiter als eine Positionsangabe im Raum, relativ zu einem Bezugspunkt, der in der Regel der Nullpunkt auf der jeweiligen Achse ist. Der Vektor [-3, 8, 5] beispielsweise beschreibt den Punkt im 3D-Raum an der Position (-3, 8, 5).

Vektoren versus Punkte

Bleibt die Frage, warum wir zwischen Vektoren und Punkten unterscheiden. Ein Punkt ist, rein mathematisch gesehen, ein unendlich kleines Objekt an einer bestimmten Position im 3D-Raum. Ein Vektor hingegen definiert sich durch zwei Eigenschaften:

1. Er hat eine bestimmte Länge.
2. Er hat eine bestimmte Richtung.

Grafisch kann man sich einen Vektor als ein Stück Linie mit einer Pfeilspitze vorstellen. Der Vektor läuft vom Ursprung des Koordinatensystems in eine bestimmte Richtung über eine bestimmte Länge und endet dann dort. Die Stelle, an der ein Vektor endet, entspricht genau der Stelle, an der ein Punkt mit korrespondierenden Koordinaten sitzt. Ein Vektor zeigt also quasi vom Nullpunkt aus auf den entsprechenden Punkt.

Warum so kompliziert?

Diese Unterscheidung zwischen Vektoren und Punkten ist deswegen so wichtig, weil Vektoren eben nicht ausschließlich auf bestimmte Punkte zeigen. Ein Vektor dient auch dazu, eine bestimmte Richtung anzuzeigen. Beispielsweise kann man einen Vektor dazu verwenden, die Bewegung eines Objekts anzugeben. Die Länge des Vektors definiert die Geschwindigkeit, mit der sich das Objekt in einem bestimmten Zeitabschnitt bewegt (beispielsweise Kilometer pro Stunde), und die Richtung des Vektors bestimmt die Richtung, in die sich das Objekt bewegt.

Da wir in der Regel sehr viel mit Vektoren rechnen müssen, ist es sehr bequem, daraus eine saubere C++-Klasse zu machen. Auf diese Weise können wir die verschiedenen Operatoren, wie beispielsweise $+$, $-$ und $*$, überladen und mit den entsprechenden Vektoroperationen belegen. Ich gehe hier davon aus, dass ihr ausreichende Kenntnisse in linearer Algebra habt und das Rechnen mit Vektoren und Matrizen beherrscht. Der Schwerpunkt dieses Kapitels liegt daher auf der Implementierung dieser 3D-Mathematik in funktionalen und vor allem vergleichsweise schnellen C++-Klassen.

Wo wir gerade bei Klassen sind: Hier ist die Definition für die Vektoren-Klasse, die wir für die *ZFXEngine* verwenden werden:



```
// In Datei: zfx3d.h
class __declspec(dllexport) ZFXVector {
public:
    float x, y, z, w;           // Koordinaten

    ZFXVector(void) { x=0, y=0, z=0, w=1.0f; }
    ZFXVector(float _x, float _y, float _z)
        { x = _x; y = _y; z = _z; w = 1.0f; }

    inline void Set(float _x, float _y, float _z,
```

```

        float _w=1.0f);
inline float GetLength(void);
inline float GetSqrLength(void) const;
inline void Negate(void);
inline void Normalize(void);
inline float AngleWith(ZFXVector &v);
inline void Difference(const ZFXVector &u,
                      const ZFXVector &v);
void operator += (const ZFXVector &v);
void operator -= (const ZFXVector &v);
void operator *= (float f);
void operator /= (float f);
float operator * (ZFXVector &v) const;
ZFXVector operator * (float f) const;
ZFXVector operator * (ZFXMatrix &m) const;
ZFXVector operator + (ZFXVector &v) const;
ZFXVector operator - (ZFXVector &v) const;

inline void Cross(const ZFXVector &u,
                 const ZFXVector &v);
}; // class

```

Unsere Klasse namens `ZFXVector` vereint bereits die wichtigsten Vektor-Operationen in sich. Wir können hier solche grundlegenden Operationen wie die Addition und Subtraktion zweier Vektoren ebenso über Operatoren durchführen wie die Berechnung des Punktproduktes oder die Multiplikation eines Vektors mit einer Matrix. Die Klasse `ZFXMatrix` lernen wir aber erst im nächsten Abschnitt kennen.

Über das C++-Schlüsselwort `operator` können wir beliebige Operatoren der Programmiersprache überladen, also mit unseren eigenen Funktionen überschreiben. Zu solchen Operatoren gehören die Symbole `+`, `-`, `*`, `/`, aber beispielsweise auch `=` und `==`. So wäre es zum Beispiel möglich, wenn auch wenig sinnvoll, das Minuszeichen `-` in unserer Klasse einfach durch eine Additionsrechnung zu überladen.

Wir nutzen diese Option jedoch sinnvoll und definieren uns beispielsweise ein `+` für eine Funktion für die Addition zweier Vektoren. Auch diese überladenen Operatoren kann man mehrfach mit verschiedenen Parameterlisten implementieren. Das sehen wir am Beispiel von `*`, das wir einmal für die Multiplikation zweier Vektoren und einmal für die Multiplikation eines Vektors mit einer Matrix verwenden.

Das Attribut `w` unserer Klasse hatte ich ja weiter oben schon einmal implizit angesprochen. Wenn wir Vektoren mit 4x4-Matrizen multiplizieren, dann benötigen wir den Vektor auch vierdimensional. Damit diese vierte Dimension aber keinen Schaden anrichtet, also den Ergebnisvektor nicht verändert,



müssen wir ihn immer auf den Wert 1 setzen und dürfen ihn beispielsweise bei der Addition und Subtraktion zweier Vektoren nicht berücksichtigen.

Damit sind wir auch schon wieder genau beim Thema. Schauen wir uns einmal an, wie wir die grundlegenden arithmetischen Operationen in unserer Klasse implementieren. Ich gehe zwar davon aus, dass die Vektormathematik bekannt ist, das Überladen von Operatoren ist dem einen oder anderen aber eventuell neu. Daher zeige ich die Implementierung dieser Funktionen hier trotzdem.



Bei der Übergabe von Parametern ist die Arbeit mit Referenzen (Call-by-Reference) zu bevorzugen, weil so keine Kopie des Objekts über den Stack übergeben werden muss. Dann kann jedoch die Funktion den Wert des übergebenen Objekts dauerhaft verändern, was nicht immer erwünscht ist. Es gibt nun aber zwei verschiedene Möglichkeiten der Übergabe von Parametern als Referenz. Bei der einen Möglichkeit sind die Parameter der Funktion bereits ein entsprechender Pointer-Typ, also werden die Variablen beim Funktionsaufruf durch den Adressoperator & nur als Adresse übergeben. Diese Möglichkeit werden wir immer dann verwenden, wenn die Funktion Zugriff auf die entsprechenden Variablen haben soll.

Möchten wir trotzdem nur Zeiger auf Variablen-Objekte als Parameter übergeben, ohne dass die Funktion schreibenden Zugriff hat, so verwenden wir die andere Methode, bei der wir zwar die Objekte als Variablen übergeben, in der Parameterliste der Funktion diese jedoch über den Adressoperator zu Referenzen machen. Das Schlüsselwörtchen `const` versichert hierbei dem Aufrufer, dass an seinem Objekt dennoch nichts verändert wird.

Grundlegende (arithmetische) Operationen

Überladen von Operatoren

Bei den überladenen Operatoren gibt es immer grundsätzlich zwei Varianten: beispielsweise den Operator `+` und seinen Cousin `+=`. Die Unterscheidung zwischen den beiden ist nicht nur rein kosmetischer oder anwendungsspezifischer Natur. Das Zeichen `+` kann man ja verwenden, um einem dritten Objekt die Summe zweier anderen Objekte zuzuweisen. Das Zeichen `+=` verwendet man, um zu einem Objekt noch ein anderes hinzuzuaddieren. Allerdings haben diese beiden Methoden auch einen klaren Geschwindigkeitsunterschied, den wir gleich sehen werden. Bei dem Zeichen `+=` kann man die Attribute des betroffenen Objekts direkt verändern. Bei dem Zeichen `+` muss man zunächst ein neues Objekt anlegen, in dessen Attributen das Ergebnis der Addition speichern und das Objekt dann zurückgeben. Sowohl das Anlegen eines Objekts (impliziter Aufruf des Standard-Konstruktors) als auch das Zurückgeben des Objekts (Kopieren über den Stack) kosten Zeit. Die Variante `+=` ist daher schneller.

Syntax der Überladung

Aber jetzt noch ein wenig zur Syntax bei der Überladung von Operatoren. Verwenden wir beispielsweise einfache Operatoren wie

```
// gegeben: ZFXVector u, v, w;
w = u + v;
```

dann wird unsere Überladung des Operators mit dem Objekt `u` aufgerufen. Als Parameter der Funktion wird das Objekt `v` angegeben. Das `return` der Funktion liefert das Ergebnis direkt auf die rechte Seite des `=`-Zeichens, wo es dann ganz normal in das Objekt `w` kopiert wird. Verwenden wir jedoch das Zeichen `+=` ...

```
// gegeben: ZFXVector u, v;
u += v;
```

... dann wird die Funktion zwar auch mit dem Objekt `u` aufgerufen und `v` wird auch als Parameter angegeben. Aber das Ergebnis der Funktion soll ja auch in `u` gespeichert werden. Also brauchen wir keine Rückgabewerte, sondern können direkt die entsprechenden Attribute von `u` verändern.

Ich denke, das reicht als kleine Einweisung in die Programmierung von überladenen Operatoren. So schwer ist das ja nicht, und alle Unklarheiten werden durch den folgenden Code beseitigt. Das Folgende ist die Implementierung der einfachen Funktionen unserer Vektoren-Klasse.

```
// _fabs-Funktion, die wir später noch verwenden
float _fabs(float f) {if (f<0.0f) return -f; return f;}
```

```
inline void ZFXVector::Set(float _x, float _y, float _z,
                           float _w) {
    x=_x; y=_y; z=_z; w=_w;
}
/*-----*/
```

```
void ZFXVector::operator += (const ZFXVector &v) {
    x += v.x;   y += v.y;   z += v.z;
}
/*-----*/
```

```
ZFXVector ZFXVector::operator + (const ZFXVector &v)
const {
    return ZFXVector(x+v.x, y+v.y, z+v.z);
}
/*-----*/
```

```
void ZFXVector::operator -= (const ZFXVector &v) {
    x -= v.x;   y -= v.y;   z -= v.z;
}
/*-----*/
```



```

ZFXVector ZFXVector::operator - (const ZFXVector &v)
const {
    return ZFXVector(x-v.x, y-v.y, z-v.z);
}
/*-----*/

void ZFXVector::operator *= (float f) {
    x *= f;  y *= f;  z *= f;
}
/*-----*/

void ZFXVector::operator /= (float f) {
    x /= f;  y /= f;  z /= f;
}
/*-----*/

ZFXVector ZFXVector::operator * (float f) const {
    return ZFXVector(x*f, y*f, z*f);
}
/*-----*/

float ZFXVector::operator * (const ZFXVector &v) const {
    return (v.x*x + v.y*y + v.z*z);
}
/*-----*/

inline float ZFXVector::GetSqrLength(void) const {
    return (x*x + y*y + z*z);
}
/*-----*/

inline void ZFXVector::Negate(void) {
    x = -x;  y = -y;  z = -z;
}
/*-----*/

inline void ZFXVector::Difference(const ZFXVector &u,
                                const ZFXVector &v) {
    x = v2.x - v1.x;
    y = v2.y - v1.y;
    z = v2.z - v1.z;
    w = 1.0f;
}
/*-----*/

inline float ZFXVector::AngleWith(ZFXVector &v) {
    return (float)acos( ((*this) * v) /
                       (this->GetLength()*v.GetLength()) );
}
/*-----*/

```

Überladene Operatoren haben wir jetzt im Griff. Nun stellt sich aber noch eine Frage: Warum haben wir uns oben so lange durch das Thema SIMD gequält und gesehen, welche riesigen Vorteile uns das beim Rechnen mit vierdimensionalen Vektoren bietet, wenn wir es hier gar nicht benutzen? Natürlich bietet SSE für 3D-Mathematik eine enorme Beschleunigungsmöglichkeit – allerdings nicht überall und um jeden Preis. Der Preis für die Anwendung von SSE wäre im Falle dieser einfachen Rechenoperationen sehr hoch und müsste in barer CPU-Zeit bezahlt werden. Die Addition zweier Vektoren über SSE würde langsamer vonstatten gehen als die reine C++-Implementierung. Das liegt daran, dass wir die beiden Vektoren erst in die entsprechenden XMM-Register schieben müssten. Dann würden sie dort addiert werden, was wiederum sehr schnell ginge, aber dann muss das Ergebnis aus dem XMM-Register auch wieder zurück in die entsprechende Variable geschoben werden. Bei einer so kleinen Rechenoperation dauert das `MOV`en der Daten schon länger, als die ganze Operation in C++-Implementierung.

Und wo bitte bleibt SIMD?

Für die etwas umfangreicheren Rechenoperationen bei Vektoren eignet sich SSE aber ganz ausgezeichnet. Hier erzielen wir durch die SSE Version der Funktion auf alle Fälle einen Performance-Gewinn. Im folgenden Abschnitt werden wir also unsere ersten echten SIMD-Funktionen schreiben, die SSE verwenden – natürlich immer vorausgesetzt, unsere Funktion zur Prozessor-Analyse hat das Vorhandensein von SSE bestätigt.

Jetzt aber!

Komplexere Operationen mit SSE-Unterstützung

Bei den einfachen Operationen für unsere Vektor-Klasse haben wir schon eine Funktion namens `GetSqrtLength()` geschrieben. Diese gibt als Rückgabewert nicht die Länge, sondern die quadrierte Länge eines Vektors wieder. Das ist deshalb so wichtig, weil das Ziehen einer Wurzel mit zu den langsamsten mathematischen Operationen gehört, die ein Computer ausführen kann. Schließlich muss der Computer die Wurzel quasi durch Annäherungsrechnung erraten. Wo dies möglich ist, sollte man also immer den Referenzwert, mit dem man die Länge eines Vektors vergleichen muss, quadrieren und mit der quadrierten Länge vergleichen. Es gibt aber auch Fälle, in denen man die Länge des Vektors wirklich genau braucht. Das Ziehen einer Wurzel in normalem C++-Code ist hier zu langsam, daher erledigen wir die Aufgabe über SSE.

Betrag eines Vektors

```
inline float ZFXVector::GetLength(void) {
    float f;

    if (!g_bsSE) {
        f = (float)sqrt(x*x + y*y + z*z);
    }
}
```



```

else {
    float *pf=&f;
    w = 0.0f;
    __asm {
        mov     ecx,   pf           ; Ergebnis merken
        mov     esi,   this        ; Vektor U
        movups  xmm0,  [esi]       ; Vektor U in XMM0
        mulps   xmm0,  xmm0        ; Mul. mit 2. Vektor
        movaps  xmm1,  xmm0        ; Ergebnis kopieren
        shufps  xmm1,  xmm1, 4Eh   ; shuffle: f1,f0,f3,f2
        addps   xmm0,  xmm1
        movaps  xmm1,  xmm0        ; Ergebnis kopieren
        shufps  xmm1,  xmm1, 11h
        addps   xmm0,  xmm1
        sqrtss  xmm0,  xmm0        ; Wurzel skalar ziehen
        movss   [ecx], xmm0        ; Ergebnis nach EDI
    }
    w = 1.0f;
}

return f;
}

```

Verzweigung der Funktion

Zunächst entscheidet die Funktion über die globale Variable, ob wir SSE-Unterstützung gefunden haben. Dazu ist es notwendig, dass der Benutzer unserer 3D-Mathematik-Bibliothek zuerst die Funktion `ZFX3DInitCPU()` einmalig bei der Initialisierung seines Programms aufgerufen hat. Diese `if`-Abfrage ist die simplere Lösung des Verzweigungsproblems zu der Funktionsvariante mit oder ohne SSE. Allerdings ist sie auch die langsamste. Wer hier einen schnelleren Weg gehen möchte, der implementiert ein Prinzip analog der D3DX-Bibliothek von Microsoft. Hier arbeitet man mit Funktionspointern, die erst dann auf die entsprechende Funktion gesetzt werden, wenn der Prozessor und dessen Features identifiziert wurden.

Die Funktion im Einzelnen

Im SSE-Teil der Funktion schieben wir erst zwei Pointer in zwei normale Register. Das ist notwendig, weil wir die Ergebnisse der Berechnung aus einem XMM-Register nicht direkt in eine unserer Variablen kopieren können. Dann verschieben wir den Vektor, also das Objekt auf das `this` zeigt, über den `MOVUPS`-Befehl in ein XMM-Register. Durch Verwendung dieses Befehls weiß der Assembler, dass er es hier mit 128 Bit gepackten Daten zu tun hat. Das entspricht also unseren vier 32-Bit-`float`-Werten des Vektors.



Um auf die Daten aus einem Speicherbereich und nicht nur auf die Adresse des Speicherbereichs zugreifen zu können, muss man das Register in eckige Klammern [] setzen.

Nun multiplizieren wir dasselbe Objekt zu dem Register hinzu. Damit haben wir also die Rechnung

$$[x*x \quad y*y \quad z*z \quad w*w]$$

ausgeführt. Nun kopieren wir das Ergebnis in ein anderes Register. Dort vertauschen wir die einzelnen Komponenten des Vektors mit dem Shuffle-Wert 4Eh, was nach Tabelle 4.1 auf Seite 131 der Dualzahl 01 00 11 10 entspricht. Damit haben wir also je zwei Komponenten des Vektors (die bereits die multiplizierten Koordinaten enthalten) vertauscht. Diesen geschuffelten Vektor addieren wir nun wieder zu dem Register XMM0. Dieses enthält damit den folgenden Vektor:

$$[x*x + z*z \quad y*y + w*w \quad z*z + x*x \quad w*w + y*y]$$

Ich glaube, ihr seht, worauf das hinausläuft. Jetzt kopieren wir diesen Vektor wiederum nach XMM1 und spielen das Shuffle-Spiel noch einmal. Diesmal shufflen wir mit 11h und addieren das Ergebnis wieder zu XMM0. Jetzt haben wir in jedem der vier Elemente dasselbe stehen, und zwar genau die quadrierte Länge des Vektors:

$$x*x + y*y + z*z + w*w$$

Es ist zwar überflüssig, dass wir diesen Wert nun viermal haben. Aber ihn auf diese Weise zu berechnen ist schneller, als wenn wir die einzelnen Komponenten des Vektors erst in andere Register bewegt und dort addiert hätten. Durch den Befehl `SQRTSS` ziehen wir jetzt die Wurzel des Wertes in dem Register (skalar, also nur auf die erste Komponente des Registers, nicht auf alle vier), und verschieben die nun korrekt ermittelte Länge durch `MOVSS` in die lokale Variable, die dann per `return` zurückgegeben wird. Zu beachten ist hier, dass wir den Wert von `w` erst mal temporär auf 0 stellen müssen, damit er bei der Berechnung der Länge das Ergebnis nicht verfälscht. Schließlich ist er nur ein Stützrad und keine echte Komponente des Vektors.

Damit haben wir unsere erste SSE-Implementierung erfolgreich gemeistert. So schlimm war es doch gar nicht, oder? Und da wir jetzt die Länge eines Vektors so schön in SSE berechnen können, wenden wir uns sofort der nächsten Funktion zu, die fast identisch ist.

Juhu!

Die Normalisierung eines Vektors bedeutet, dass wir ihn auf eine Länge von genau 1.0 bringen wollen. Das ist an verschiedenen Stellen notwendig, da andere Vektorberechnungen normalisierte Vektoren voraussetzen. Auch für lokale Achsen eines sich bewegenden Objekts ist es notwendig, die Vektoren auf Einheitslänge zu korrigieren. Um einen Vektor zu normalisieren, müssen wir lediglich jede seiner Komponenten durch seine Länge teilen. Wir ver-

*Normalisierung
eines Vektors*

wenden daher zunächst dieselbe Berechnung wie eben bei der Längenberechnung. Am Ende allerdings, bevor wir die Wurzel ziehen, nehmen wir eine Änderung vor und ergänzen die Rechnung dann noch entsprechend.



```
inline void ZFXVector::Normalize(void) {
    if (!g_bsSE) {
        float f = (float)sqrt(x*x + y*y + z*z);

        if (f != 0.0f) {
            x/=f; y/=f; z/=f;
        }
    }
    else {
        w = 0.0f;
        __asm {
            mov     esi,     this           ; Vektor U
            movups  xmm0,    [esi]         ; Vektor U nach XMM0
            movaps  xmm2,    xmm0          ; Original kopieren
            mulps   xmm0,    xmm0          ; Mul. mit 2. Vektor
            movaps  xmm1,    xmm0          ; Ergebnis kopieren
            shufps  xmm1,    xmm1, 4Eh     ; shuffle
            addps   xmm0,    xmm1
            movaps  xmm1,    xmm0          ; Ergebnis kopieren
            shufps  xmm1,    xmm1, 11h     ; shuffle
            addps   xmm0,    xmm1

            rsqrtps  xmm0,    xmm0         ; reziproke Wurzel
            mulps   xmm2,    xmm0         ; Mul. mit reziprok
            movups  [esi],    xmm2
        }
        w = 1.0f;
    }
}
```

Nachdem wir in jeder Komponente das Ergebnis der Längenberechnung noch ohne das Wurzelziehen stehen haben, ziehen wir hier nicht die Wurzel. Wir bilden die reziproke Wurzel, also den Umkehrbruch, und multiplizieren diesen Wert mit dem bisherigen Ergebnis. Das ist dasselbe, als wenn wir durch die Wurzel teilen würden, hat aber einen entscheidenden Vorteil: Die reziproke Wurzel wird nicht berechnet, sondern in einer Look-Up-Table im Prozessor nachgeschlagen. Damit ist sie schneller als die Berechnung der Wurzelfunktion, jedoch im letzten Nachkommastellen-Bereich ein wenig ungenau. Hier ist uns aber die Geschwindigkeit wichtiger.



Bei der Normalisierung eines Vektors könnte man auch die reziproke Wurzel mit der Funktion RSQRTSS berechnen, also für nur eine Komponente statt für alle vier. Über die SHUFPS-Anweisung könnte man das Ergebnis dann auf alle vier Komponenten von XMM0 broadcasten. Das ist jedoch bei der rezi-

proken Wurzel langsamer, als wenn man diese gleich für alle vier Komponenten nachschlägt. Wer jedoch mit der richtigen Wurzel arbeiten möchte, sollte den Geschwindigkeitsunterschied dieser Methode auf alle Fälle testen.

Eine andere häufig gebrauchte Operation mit Vektoren ist die Berechnung des so genannten Kreuzprodukts. Nimmt man zwei beliebig im Raum ausgerichtete, nicht parallele Vektoren, dann kann man mittels des Kreuzprodukts einen dritten Vektor berechnen. Dieser dritte Vektor hat die Eigenschaft, dass er zu den beiden anderen Vektoren rechtwinklig angeordnet ist. Diese Funktion braucht man insbesondere dann, wenn man den Normalenvektor einer Ebene oder eines Polygons bestimmen will. Auch hier ist die SSE-Variante der Funktion schneller, als es reiner C++-Code wäre.

**Kreuzprodukt
zweier Vektoren**

```
inline void ZFXVector::Cross(const ZFXVector &u,
                             const ZFXVector &v) {
    if (!g_bSSE) {
        x = v1.y * v2.z - v1.z * v2.y;
        y = v1.z * v2.x - v1.x * v2.z;
        z = v1.x * v2.y - v1.y * v2.x;
        w = 1.0f;
    }
    else {
        __asm {
            mov esi, v1
            mov edi, v2

            movups xmm0, [esi]
            movups xmm1, [edi]
            movaps xmm2, xmm0
            movaps xmm3, xmm1

            shufps xmm0, xmm0, 0xc9
            shufps xmm1, xmm1, 0xd2
            mulps  xmm0, xmm1

            shufps xmm2, xmm2, 0xd2
            shufps xmm3, xmm3, 0xc9
            mulps  xmm2, xmm3

            subps  xmm0, xmm2

            mov  esi, this
            movups [esi], xmm0
        }
        w = 1.0f;
    }
}
```



Wir kopieren hier je einen der beiden Vektoren in je zwei Register. Eines der Register verwenden wir zum Shufflen, in dem jeweils anderen Register multiplizieren wir dann den originalen Vektor mit dem geschuffelten. Dann haben wir in den beiden Registern XMM0 und XMM2 die beiden Vektoren stehen, die bereits die korrekte Multiplikation der einzelnen Komponenten (siehe auch C++-Code in der Funktion) darstellen. Im letzten Berechnungsschritt führen wir noch die Subtraktion durch die Anweisung `SUBPS` aus und haben den endgültigen Ergebnisvektor berechnet. Dieser wird dann an die Stelle im Speicherbereich kopiert, auf die der `this`-Zeiger zeigt. Die Shuffle-Werte schreiben wir hier mit dem Präfix `0x` anstelle des Suffix `h`, weil der Compiler die mit einem Buchstaben beginnenden Werte sonst nicht als Hexadezimalzahl erkennt. Wer noch Probleme mit dem Verständnis des Shufflens hat, dem empfehle ich, sich diese Funktion Schritt für Schritt mit den jeweiligen Komponenten der Vektoren aufzumalen.

Vektor-Matrix-Multiplikation

Die letzte Funktion, die unsere Klasse `ZFXVector` definiert, ist die Multiplikation eines Vektors mit einer Matrix. Auch diese Rechnung sollte uns hinreichend bekannt sein. Schließlich müssen wir sie immer dann verwenden, wenn wir beispielsweise einen Vektor mit einer Rotationsmatrix multiplizieren, um den Vektor im Raum zu drehen. Auch hier beschleunigt SSE die Rechnung um einiges, und so häufig, wie wir diese Funktion verwenden, werden lohnt sich die Optimierung hier auf alle Fälle. Als Parameter der Funktion verwenden wir ein Objekt der Klasse `ZFXMatrix`, die wir im nächsten Abschnitt behandeln werden.



```
ZFXVector ZFXVector::operator * (const ZFXMatrix &m)
const {
    ZFXVector vcResult;

    if (!g_bSSE) {
        vcResult.x = x*m._11 + y*m._21 + z*m._31 + m._41;
        vcResult.y = x*m._12 + y*m._22 + z*m._32 + m._42;
        vcResult.z = x*m._13 + y*m._23 + z*m._33 + m._43;
        vcResult.w = x*m._14 + y*m._24 + z*m._34 + m._44;

        vcResult.x = vcResult.x/vcResult.w;
        vcResult.y = vcResult.y/vcResult.w;
        vcResult.z = vcResult.z/vcResult.w;
        vcResult.w = 1.0f;
    }
    else {
        float *ptrRet = (float*)&vcResult;
        __asm {
            mov     ecx,    this        ; Vektor
            mov     edx,    m           ; Matrix
            movss   xmm0,    [ecx]
            mov     eax,    ptrRet      ; Ergebnis Vektor
            shufps  xmm0,    xmm0, 0
        }
    }
}
```

```

    movss xmm1, [ecx+4]
    mulps xmm0, [edx]
    shufps xmm1, xmm1, 0
    movss xmm2, [ecx+8]
    mulps xmm1, [edx+16]
    shufps xmm2, xmm2, 0
    movss xmm3, [ecx+12]
    mulps xmm2, [edx+32]
    shufps xmm3, xmm3, 0
    addps xmm0, xmm1
    mulps xmm3, [edx+48]
    addps xmm2, xmm3
    addps xmm0, xmm2
    movups [eax], xmm0      ; Speicher als Ergebnis
    mov    [eax+3], 1      ; w = 1
}
}
return vcResult;
}

```

An dieser Funktion gibt es nichts spektakulär Neues. Mit dem bisher Gelernten sollte man also auch mit dem SSE-Code keine Probleme mehr haben. Hier wird zwar auch wieder viel geshuffelt, dabei handelt es sich aber nur um reine Broadcastings.

Zugegeben, bisher haben wir ziemlich viel mit SSE gearbeitet. Man sollte dabei immer bedenken, dass Vektoren mit die kleinste Einheit sind, in der wir rechnen. Optimiert wird zunächst grundsätzlich nur dort, wo wir Code antreffen, der oft ausgeführt wird. »Oft« bedeutet mindestens mehrere hundert Male in einem Frame des Programms. Die grundlegenden Vektor-Operationen durch SSE zu beschleunigen, macht also sehr viel Sinn, und wie wir gesehen haben, ist das noch nicht einmal viel Aufwand. Im Folgenden werden wir solche Optimierungen aber weniger verwenden, denn wir kommen nun zu Rechen-Operationen, die wir seltener verwenden. Doch auch dort bietet es sich an, SSE-Code einzubauen. Das wird dann dort aber etwas umfangreicher, weshalb wir hier darauf verzichten wollen. Wer jedoch noch mehr Speed aus seiner Engine herauskitzeln möchte, findet dort einen guten Ansatzpunkt.

So viel SSE?

4.3 Rechnen mit Matrizen

Viele Geschichten und Geheimnisse ranken sich um dieses mysteriöse mathematische Gebilde. Ist es bei Vektoren noch recht einfach, ihren Sinn in der Anwendung zu erkennen, weil man sie mit bestimmten physikalischen Eigenschaften assoziieren kann, ist das bei den Matrizen leider ein wenig anders. Man sollte auch nicht lange nachgrübeln und versuchen, etwas in Matrizen hineinzuiinterpretieren. Bei Matrizen handelt es sich lediglich um

Was ist eine Matrix?

ein großes Meeting von vielen Zahlen. Diese werden in Reihen und Spalten angeordnet, ähnlich einer Tabelle von Zahlen. Dabei muss jede Reihe eine identische Anzahl von Einträgen haben, ebenso wie jede Spalte. Matrizen werden in der Mathematik zu ganz unterschiedlichen Operationen verwendet, beispielsweise zum Lösen von Gleichungssystemen und zur Berechnung von Extrema n-dimensionaler Funktionen.

*Haste mal 'ne
Matrix?*

Sind die Vektoren die kleinsten Einheiten unsere 3D-Engine, dann gehören die Matrizen gleich auf die nächste Stufe hinter den Vektoren. Wenn wir beispielsweise Objekte transformieren wollen, können wir dies für jede Achse separat tun und die entsprechenden Vektoren umrechnen. Matrizen bieten uns die Möglichkeit, die verschiedenen Transformationen alle in einer Matrix zusammenzufassen und dann in einem statt in multiple Arbeitsschritten auf einen Vektor anzuwenden. Ebenso wie Rotations-, Translations- und Skalierungsmatrizen gibt es noch eine Vielzahl anderer Matrizen, mit denen wir in der 3D-Grafik relevante Berechnungen ausführen können, beispielsweise Kamera-, Projektions-, Billboard- und LookAt-Matrizen.

Die Matrizenrechnung ist also integraler Bestandteil einer schnellen 3D-Engine. Damit wir die Matrizen ebenso handlich bedienen können wie unseren `ZFXVector`, definieren wir die folgenden Klasse für Matrizen.



```
// In Datei: zfx3d.h
class __declspec(dllexport) ZFXMatrix {
public:
    float _11, _12, _13, _14;
    float _21, _22, _23, _24;
    float _31, _32, _33, _34;
    float _41, _42, _43, _44;

    ZFXMatrix(void) { /* nothing to do */ ; }

    inline void Identity(void);      // Einheitsmatrix
    inline void RotaX(float a);      // X-Achse
    inline void RotaY(float a);      // Y-Achse
    inline void RotaZ(float a);      // Z-Achse
    inline void RotaArbi(ZFXVector vcAxis, float a);
    inline void Translate(float dx, float dy,
                          float dz);

    inline void TransposeOf(const ZFXMatrix &m);
    inline void InverseOf(const ZFXMatrix &m);

    ZFXMatrix operator * (const ZFXMatrix &m) const;
    ZFXVector operator * (const ZFXVector &vc) const;
}; // class
```

Grundlegende Operationen

Unser Matrix-Typ ist nichts weiter als eine Ansammlung von vier mal vier float-Werten und Funktionen, um diese Struktur nach bestimmten Kriterien zu berechnen und mit Werten zu füllen. Da ich voraussetze, dass alle diese Matrixoperationen bekannt sind, liste ich im Folgenden nur kurz die Funktionen auf. Eine Erklärung oder Herleitung der entsprechenden Funktionen findet ihr in nahezu jeder Quelle über 3D-Grafik, beispielsweise auch in einem Internet-Tutorial über DirectX3D von mir.⁶

Beschreibung

```
inline void ZFXMatrix::Identity(void) {
    float *f = (float*)&this->_11;
    memset(f, 0, sizeof(ZFXMatrix));
    _11 = _22 = _33 = _44 = 1.0f;
}

/*-----*/

// Rotationsmatrix um die X-Achse
inline void ZFXMatrix::RotaX(float a) {
    float fCos = cosf(a);
    float fSin = sinf(a);
    _22 = fCos;
    _23 = fSin;
    _32 = -fSin;
    _33 = fCos;
    _11 = _44 = 1.0f;
    _12=_13=_14=_21=_24=_31=_34=_41=_42=_43=0.0f;
}

/*-----*/

// Rotationsmatrix um die Y-Achse
inline void ZFXMatrix::RotaY(float a) {
    float fCos = cosf(a);
    float fSin = sinf(a);

    _11 = fCos;
    _13 = -fSin;
    _31 = fSin;
    _33 = fCos;

    _22 = _44 = 1.0f;
    _12 = _23 = _14 = _21 = _24 = _32 = _34 = _41 = _42 = _43 = 0.0f;
}

/*-----*/
```

⁶ www.zfx.info

```

// Rotationsmatrix um die Z-Achse
inline void ZFXMatrix::RotaZ(float a) {
    float fCos = cosf(a);
    float fSin = sinf(a);
    _11 = fCos;
    _12 = fSin;
    _21 = -fSin;
    _22 = fCos;
    _33 = _44 = 1.0f;
    _13=_14=_23=_24=_31=_32=_34=_41=_42=_43=0.0f;
}

/*-----*/

// Rotationsmatrix von a Radian um eine beliebige Achse
inline void ZFXMatrix::RotaArbi(ZFXVector vcAxis,
                                float a) {
    float fCos = cosf(a);
    float fSin = sinf(a);
    float fSum = 1.0f - fCos;

    vcAxis.Normalize();

    _11 = (vcAxis.x*vcAxis.x) * fSum + fCos;
    _12 = (vcAxis.x*vcAxis.y) * fSum - (vcAxis.z*fSin);
    _13 = (vcAxis.x*vcAxis.z) * fSum + (vcAxis.y*fSin);

    _21 = (vcAxis.y*vcAxis.x) * fSum + (vcAxis.z*fSin);
    _22 = (vcAxis.y*vcAxis.y) * fSum + fCos ;
    _23 = (vcAxis.y*vcAxis.z) * fSum - (vcAxis.x*fSin);

    _31 = (vcAxis.z*vcAxis.x) * fSum - (vcAxis.y*fSin);
    _32 = (vcAxis.z*vcAxis.y) * fSum + (vcAxis.x*fSin);
    _33 = (vcAxis.z*vcAxis.z) * fSum + fCos;

    _14 = _24 = _34 = _41 = _42 = _43 = 0.0f;
    _44 = 1.0f;
}

/*-----*/

inline void ZFXMatrix::Translate(float dx, float dy, float dz) {
    _41 = dx;
    _42 = dy;
    _43 = dz;
}

/*-----*/

inline void ZFXMatrix::TransposeOf(const ZFXMatrix &m) {
    _11 = m._11;

```



```

    _21 = m._12;
    _31 = m._13;
    _41 = m._14;

    _12 = m._21;
    _22 = m._22;
    _32 = m._23;
    _42 = m._24;

    _13 = m._31;
    _23 = m._32;
    _33 = m._33;
    _43 = m._34;

    _14 = m._41;
    _24 = m._42;
    _34 = m._43;
    _44 = m._44;
}
/*-----*/

```

Die Funktion für die Multiplikation eines Vektors mit einer Matrix haben wir ja weiter oben, im Abschnitt über Vektoren, bereits gesehen. Ich habe sie aus dem Grund noch einmal in die Matrix-Klasse integriert, damit einerseits die Reihenfolge der Operatoren egal ist, man also jetzt auch $M*V$ statt $V*M$ schreiben kann. Andererseits ist diese Variante der Funktion aber schneller als die der Vektorenklasse. Das liegt daran, dass hier nur ein Vektor (vier `float`-Werte) über den Stack an die Funktion übergeben werden muss. In der Vektorklasse mussten wir eine größere Struktur, nämlich eine Matrix (sechzehn `float`-Werte) über den Stack schieben. Die Funktion zur Multiplikation einer Matrix mit SSE-Code ist viel zu lang, um sie hier abzu-
drucken, Gleiches gilt für die Invertierung einer Matrix (ob nun mit SSE oder ohne). Für beides bietet aber Intel Tutorials auf seiner Internetseite⁷ an, und der Quelltext unserer Implementierung befindet sich auf der CD.

Und die anderen?

Dies sind die wichtigsten Funktionalitäten, die wir von einer handlichen Matrix erwarten. Insbesondere die Operatoren machen unseren Code, ebenso wie die Operatoren bei unserer Vektor-Klasse, sehr viel leichter lesbar – immer vorausgesetzt, man verwendet eindeutige Namen mit entsprechenden Präfixen für seine Variablen.

Es gibt sicherlich noch viele andere Matrizen-Operationen, die man implementieren könnte. Wir wollen unseren Code jedoch so schlank und effektiv wie möglich halten. Der Sinn und Zweck unserer Implementierung ist es ja nicht, eine möglichst umfassende Matrix-Klasse zu entwerfen, sondern eine Matrix-Klasse, die uns all das bietet, was wir im Bereich der 3D-Mathema-

*Objekt-
Orientierung; Pros
und Cons*

⁷ Streaming SIMD Extensions – Matrix Multipl.; Order No. 245045-001

tik oft brauchen werden. Dass die Attribute der Klasse öffentlich zugänglich sind, eröffnet einem Anwender auch die Möglichkeit, eigene Routinen für unsere Matrix-Klasse zu implementieren. Auch wenn dies dem strikt objekt-orientierten Ansatz widerspricht, so ist es hier doch zweckmäßig, die Attribute frei zugänglich zu machen. Selbiges gilt übrigens auch für die Vektor-Klasse. Man sollte nämlich auch nicht vergessen, dass Assessor-Methoden wie `Get()` und `Set()` mehr Zeit kosten als ein direkter Zugriff auf die Attribute.

4.4 Rechnen mit Strahlen

*Unsere
3D-Bibliothek*

Neben Vektoren und Matrizen gibt es noch diverse andere mathematische oder geometrische Objekte, mit denen wir bei einer 3D-Engine häufig zu tun haben. Diese treten zwar unter Umständen nur selten in Erscheinung, aber dennoch wäre es nett, sie im Fundus unserer 3D-Bibliothek zu haben. Im folgenden Teil dieses Kapitels begeben wir uns in den Bereich der Kollisionsabfragen und des Objekt-Cullings in Weltkoordinaten. Aber ich möchte nicht zu weit vorgreifen. Als Nächstes implementieren wir eine Klasse für Strahlen (Rays), mit deren Hilfe wir später so einiges anstellen können. Insbesondere können wir mit ihnen Kollisionsabfragen für kleine, schnelle Objekte (wie beispielsweise Geschosse) und natürlich ein Picking von Objekten im 3D-Raum mit der Maus durchführen.

*Siebzehn Strahlen
sind eine Sonne*

Ein Strahl ist im mathematischen Sinne Teil einer Linie. Und was ist eine Linie? Nun, eine Linie ist ein Gebilde, das sich eindimensional von **-symbUnendlich** bis **+symbUnendlich** ausdehnt. Ein Strahl hingegen beginnt erst ab einer bestimmten, definierten Stelle im n-dimensionalen Raum, die man *Ursprung* (Origin) nennt. Von dort aus läuft der Strahl eindimensional in eine bestimmte Richtung (Direction) über seinen Richtungsvektor bis in alle Unendlichkeit. Man kann sich das wie den unsichtbaren Laserstrahl einer Laserkanone mit unendlicher Reichweite vorstellen. Unsere Klasse für Strahlen wird aber nicht sehr umfangreich sein. Wir benötigen lediglich eine Methode, um den Ursprung und den Richtungsvektor eines Strahls festlegen zu können. Als weitere Funktionalität brauchen wir dann eine Funktion, die den Strahl (der normalerweise im Weltkoordinatensystem angegeben ist) in das lokale Koordinatensystem eines beliebigen Objekts transformiert. Schließlich geben wir unserer Klasse noch eine mehrfach überladene Funktion mit auf den Weg, die auf eine Kollision zwischen dem Strahl und verschiedenen anderen Objekten hin prüfen kann. Strahlen verwendet man in der 3D-Mathematik nämlich hauptsächlich für Kollisionsabfragen. Doch dazu später mehr.



```

class __declspec(dllexport) ZFXRay {
public:
    ZFXVector m_vcOrig, // Ursprung
               m_vcDir;  // Richtung

    ZFXRay(void) { /* nothing to do */ ; }

    inline void Set(ZFXVector vcOrig,
                   ZFXVector vcDir);
    inline void DeTransform(const ZFXMatrix &m);

    // Schnittpunkt mit Dreieck
    bool Intersects(const ZFXVector &vc0,
                   const ZFXVector &vc1,
                   const ZFXVector &vc2, bool bCull,
                   float *t);
    bool Intersects(const ZFXVector &vc0,
                   const ZFXVector &vc1,
                   const ZFXVector &vc2, bool bCull,
                   float fL, float *t);

    // Schnittpunkt mit Ebene
    bool Intersects(const ZFXPlane &plane, bool bCull,
                   float *t, ZFXVector *vcHit);
    bool Intersects(const ZFXPlane &plane, bool bCull,
                   float fL, float *t,
                   ZFXVector *vcHit);

    // Schnittpunkt mit AABB
    bool Intersects(const ZFXAabb &aabb,
                   ZFXVector *vcHit);

    // Schnittpunkt mit OBB
    bool Intersects(const ZFXObb &obb, float *t);
    bool Intersects(const ZFXObb &obb, float fL,
                   float *t);
}; // class

```

Grundlegende Operationen

Wir beginnen mit der einfachsten Operation für diese Klasse, nämlich dem Festlegen der beiden Attribute über eine Assessor-Funktion. Man beachte hierbei, dass der Richtungsvektor durch diese Funktion nicht extra normalisiert wird. Viele Operationen bei den Strahlen setzen voraus, dass der Richtungsvektor normalisiert ist; dafür hat dann der Konstrukteur des Strahls vor der Übergabe der Werte zu sorgen. Auf diese Weise ist es aber auch möglich, nicht-normalisierte Richtungsvektoren zu erzeugen, und die Funktion führt auch keine *verdeckten* Operationen durch, die der Aufrufer eventuell nicht erwartet.

Attribute festlegen



```
inline void ZFXRay::Set(ZFXVector vcOrig,
                       ZFXVector vcDir) {
    m_vcOrig = vcOrig;
    m_vcDir  = vcDir;
}
```

Transformation zwischen Koordi- natensystemen

Um den Strahl in das lokale Koordinatensystem eines Objekts zu transformieren, müssen wir natürlich dessen Weltmatrix an die Funktion `ZFXRay::DeTransform` übergeben. Normalerweise sind die Objekte ja im Speicher des Programms in lokalen Koordinaten abgelegt und werden von der Grafikkarte nur zum Rendern in Weltkoordinaten umgerechnet (und dann weiter transformiert und projiziert). Wenn wir nun beispielsweise eine Kollisionsabfrage mit einem Strahl und einem Objekt durchführen wollen, dann müssten wir jeden einzelnen Vertex des Objekts in Weltkoordinaten transformieren, damit das Objekt im selben Bezugssystem wie der Strahl vorliegt – schließlich berechnen wir die Strahlen in der Regel im Weltkoordinatensystem. Die schnellere Alternative ist es natürlich, den Strahl in das lokale Koordinatensystem des Objekts zu transformieren. Hier müssen wir lediglich zwei Vektoren umrechnen und nicht n Vektoren wie bei der vorherigen Methode (wobei n die Anzahl der Vertices des Objekts ist). Eine solche *Rück-Transformation* erreichen wir, indem wir den Strahl mit der inversen Weltmatrix des Objekts transformieren. Man beachte also, dass diese Funktion den Strahl nicht mit der als Parameter übergebenen Matrix transformiert, sondern den Strahl im Gegenteil so umrechnet, dass er im selben lokalen Koordinatenraum vorliegt wie diejenigen Objekte, die mittels dieser Matrix in den Koordinatenraum des Strahls (Weltkoordinatensystem) transformiert würden.



Ein wenig knifflig ist das natürlich, weil wir hier die Verschiebung des Strahls im Auge behalten müssen. Wenn wir eine Transformationsmatrix invertieren, so wird sie um einen falschen Punkt rotiert, wenn sie noch eine Verschiebung beinhaltet. Wir müssen also die Verschiebung aus der Matrix extrahieren, gesondert auf den Strahl anwenden und dann die Matrix, die nur noch Rotationen enthält, invertieren und auf den Strahl anwenden.



```
inline void ZFXRay::DeTransform(const ZFXMatrix &m) {
    ZFXMatrix mInv;
    ZFXMatrix m=m;
    // invertiere Verschiebung
    m_vcOrig.x -= m._41;
    m_vcOrig.y -= m._42;
    m_vcOrig.z -= m._43;

    // lösche Verschiebung aus der Matrix
    m._41=0.0f;  m._42=0.0f;  m._43=0.0f;

    // invertiere Matrix
```

```

mInv.Inverse(&m);
m_vcOrig = m_vcOrig * mInv;
m_vcDir  = m_vcDir  * mInv;
}

```

Bereits hier sehen wir auch schon den Vorteil, den uns die überladenen Operatoren der `ZFXVector`- und `ZFXMatrix`-Klassen liefern. Wir können in dieser Funktion ganz bequem mit dem `*`-Zeichen für die Multiplikation zweier Vektoren oder eines Vektors mit einer Matrix arbeiten.

Kollision mit Dreiecken

Kommen wir damit zur ersten Schnittpunkt-(Intersection-)Funktion unserer Engine. Im 3D-Raum lässt sich relativ einfach und schnell herausfinden, ob ein Strahl ein Dreieck geschnitten hat. Mehr noch, wir können diese Funktion sehr leicht so erweitern, dass wir die Rückseiten der Dreiecke entweder mit berücksichtigen können oder nicht, und wir können uns die genaue Entfernung vom Strahlursprung zum Schnittpunkt geben lassen, sofern es einen solchen gab. Die folgende Methode stammt von Möller und Trumbore. Dabei wird das Dreieck, vereinfacht gesagt, an den Ursprung des Weltkoordinatensystems verschoben und zu einem Einheitsdreieck in der *y*- und *z*-Ebene transformiert, während der Strahl auf die *x*-Ebene ausgerichtet wird. Die Implementierung sieht wie folgt aus:

*Schnittpunkt
eines Strahls mit
einem Dreieck*

```

bool ZFXRay::Intersects(const ZFXVector vc0,
                        const ZFXVector vc1,
                        const ZFXVector vc2,
                        bool bCull, float *t) {
    ZFXVector pvec, tvec, qvec;

    ZFXVector edge1 = vc1 - vc0;
    ZFXVector edge2 = vc2 - vc0;

    pvec.Cross(m_vcDir, edge2);

    // Wenn nahe 0 ist Strahl parallel
    float det = edge1 * pvec;
    if ( (bCull) && (det < 0.0001f) )
        return false;
    else if ( (det < 0.0001f) && (det > -0.0001f) )
        return false;

    // Entfernung zur Ebene, < 0 = hinter der Ebene
    tvec = m_vcOrig - vc0;
    float u = tvec * pvec;
    if (u < 0.0f || u > det)
        return false;
}

```



```

qvec.Cross(tvec, edge1);
float v = m_vcDir * qvec;
if (v < 0.0f || u+v > det)
    return false;

if (t) {
    *t = edge2 * qvec;
    float fInvDet = 1.0f / det;
    *t *= fInvDet;
}
return true;
} // Intersects(Tri)

```

Wie man sieht, ist die Funktion vom Rechenaufwand her nicht sonderlich anspruchsvoll. Vielleicht wäre es eine gute Übung, die gesamte Funktion in SSE zu programmieren? Aber das überlasse ich euch als Hausaufgabe.

**Schnittpunkt
eines Strahlseg-
ments mit einem
Dreieck**

Dieselbe Funktion können wir in fast identischer Weise nochmals verwenden. In bestimmten Situationen kann es notwendig werden, nicht auf den Schnittpunkt eines Strahls, sondern nur auf den Schnittpunkt mit einem Teil des Strahls zu prüfen. Einen solchen Teil, gemessen ab dem Ursprung des Strahls über eine bestimmte Länge hinweg, nennt man *Segment*. Beispielsweise kann man so die Reichweite der Maus beim Anklicken von Objekten (Picking) einer bestimmten Maximalreichweite unterwerfen. Um das zu realisieren, müssen wir die Funktion lediglich um einen Parameter für die gewünschte Länge erweitern:

```

bool ZFXRay::Intersects(const ZFXVector vc0,
                        const ZFXVector vc1,
                        const ZFXVector vc2,
                        bool bCull,
                        float fL, float *t);

```

Am Ende der Funktion berechnen wir, wie oben gesehen, die Distanz vom Strahlursprung zu dem Schnittpunkt, sofern es einen gab. Ist diese Distanz größer als der Wert der Länge `fL`, dann gab es zwar eine Kollision, allerdings nicht auf dem betrachteten Segment des Strahls. Diese Funktion findet ihr natürlich auch komplett auf der CD zu diesem Buch, aber die notwendigen Änderungen betreffen gerade mal zwei Zeilen am Ende der Funktion, daher drucke ich sie an dieser Stelle nicht ab.

Kollision mit Ebenen

**Schnittpunkt
eines Strahls mit
einer Ebene**

Nachdem wir uns nun um die Dreiecke gekümmert haben, schauen wir uns die Kollision von Strahlen mit Ebenen an. Auch hierfür gibt es einige Verwendungen. Sollte ihr mal in die Verlegenheit kommen, mit *Binary Space Partitioning*-(BSP-)Bäumen arbeiten zu müssen, werdet ihr alles zu schätzen wissen, was euch die Arbeit mit Strahlen und Ebenen erleichtert.

Binary Space Partitioning nennt man die Technik, bei der die Geometrie eines Levels rekursiv immer an einer bestimmten Ebene durch ein Polygon der Geometrie aufgeteilt wird, um den Level in konvexe Teile zu zerlegen. Diese Technik fand in Computerspielen erstmals durch die Firma id Software in den Klassikern Castle Wolfenstein und Doom Anwendung und wird auch in aktuellen Engines immer noch verwendet.

Es ist zwar nicht unsere Hauptintention, auf eine bestimmte Technik hinzuwirken, aber je allgemeiner verwendbar wir unsere Engine halten, desto besser ist das. Glücklicherweise gibt es bereits seit Urzeiten das Raytracing – na ja, ganz so lange ist das noch nicht her. Glück ist das für uns deshalb, weil für den Raytracer ein Strahl quasi genau das ist, was für uns ein Vektor ist – nämlich die wichtigste Einheit, mit der er arbeitet.

Raytracer berechnen aus 3D-Szenen fotorealistische Bilder, indem sie von einer oder mehreren Lichtquellen aus Strahlen in die Szene schießen, um den Verlauf des Lichts real nachzubilden. Trifft ein solcher Lichtstrahl ein Objekt in der Szene, dann wird dort, basierend auf der Objektoberfläche, dem Lichteinfallswinkel, der Lichtintensität usw., die Helligkeit des Objekts an der Stelle berechnet.

Daher sind aus dem Raytracing auch sehr viele Algorithmen bekannt, die Schnittpunkte mit anderen Objekten berechnen. Unter anderem auch einer, mit dem wir den Schnittpunkt eines Strahls mit einer Ebene testen können. Die folgende Funktion berechnet diesen Schnittpunkt – je nach Wahl des Aufrufers entweder nur die Vorderseite der Ebene oder beide Seiten. Als Rückgabewert erhält man `true` oder `false` und auf Wunsch auch die Entfernung vom Strahlursprung zu dem Schnittpunkt und auch den Schnittpunkt selbst. Je mehr Informationen man von dieser Funktion abfragt, desto langsamer wird die Berechnung natürlich. Diese Zusatzinformationen sollte man sich also nur dann ausgeben lassen, wenn man sie wirklich benötigt. Sonst gibt man für den entsprechenden Parameter einfach `NULL` an.

```
bool ZFXRay::Intersects(const ZFXPlane &plane,
                        bool bCull, float *t,
                        ZFXVector *vcHit) {
    float Vd = plane.m_vcN * m_vcDir;

    // Strahl parallel zur Ebene
    if (_fabs(Vd) < 0.00001f)
        return false;

    // Normalenvektor zeigt von der Strahl-Richtung weg
    // => Schnitt mit Rückseite falls überhaupt
    if (bCull && (Vd > 0.0f))
        return false;
```



```

float Vo = -( plane.m_vcN * m_vcOrig) + plane.m_fD);

float _t = Vo / Vd;

// Schnitt jenseits des Strahl-Ursprungs
if (_t < 0.0f)
    return false;

if (vcHit) {
    (*vcHit) = m_vcOrig + (m_vcDir * _t);
}

if (t)
    (*t) = _t;

return true;
} // Intersects(Plane)

```

*Schnittpunkt
eines Strahlseg-
ments mit einer
Ebene*

Durch diese Funktion können wir also bequem entscheiden, ob ein Strahl eine Ebene getroffen hat oder nicht. So langsam mausert sich unsere ZFXRay-Klasse zu einem sehr wichtigen Werkzeug, weil sie einen großen Umfang an Kollisionsmethoden definiert. Natürlich können wir diese Funktion ebenfalls wieder als Funktion für ein Strahlsegment umschreiben. Der Prototyp dieser Funktion sieht wie folgt aus (die Implementierung findet sich auf der CD zu diesem Buch):

```

bool ZFXRay::Intersects(const ZFXPlane &plane,
                        bool bCull, float fL,
                        float *t, ZFXVector *vcHit);

```

Im Falle einer Kollision prüfen wir die Entfernung vom Kollisionspunkt zum Ursprung des Strahls einfach gegen den Parameter `fL` für die Länge, die unser Segment haben soll. Liegt der Punkt weiter entfernt, so kollidiert der Strahl zwar mit der Ebene, aber nicht mehr in dem Bereich, der uns interessiert. Dies bauen wir einfach in die Abfrage `_t < 0.0f` ein, indem wir die `if`-Abfrage durch ein logisches ODER `_t > fL` ergänzen.

Wir wenden uns nun der nächsten wichtigen Kollisionsfunktion zu. Bevor wir nämlich einen Strahl beispielsweise auf eine Kollision mit einem Dreieck in der Level-Geometrie prüfen, ist es in der Regel sinnvoller, in einer hierarchischen Struktur den Strahl auf Kollision mit einer Bounding-Box zu testen, die um bestimmte, größere Teile der Geometrie des Levels gespannt ist.

Kollision mit Bounding-Boxen

*Schnittpunkt
eines Strahls mit
einer AABB*

Die einfachere Variante einer Bounding-Box ist eine an den Weltachsen ausgerichtete Bounding-Box oder kurz *AABB* (Axis-Aligned Bounding-Box) genannt. Diese ist sehr einfach zu berechnen, weil man für jede Art von Geo-

metrie nur die gesamte Vertexliste durchlaufen muss und sich auf den jeweiligen Achsen die minimal und maximal auftretenden Werte merkt. Im Gegensatz zu den orientierten Boxen (OBB) sind solche AABB aber ungenauer, da sie die unterliegende Geometrie in der Regel weniger eng umschließen. Dennoch sind sie aufgrund ihrer Einfachheit immer noch sehr nützlich. Also entwickeln auch wir eine Klasse `ZFXAabb` für solche Boxen. Allerdings folgt diese erst im übernächsten Abschnitt, also noch ein klein wenig Geduld.

Mit der folgenden Funktion testen wir auf eine Kollision eines Strahls mit einer ABB. Sie ist eine leicht veränderte Variante der Funktion von Andrew Woo (*Graphics Gems*; Academic Press; 1990) .

```
bool ZFXRay::Intersects(const ZFXAabb &aabb,
                       ZFXVector *vcHit)
{
    bool bInside = true;
    ZFXVector MaxT;

    MaxT.Set(-1.0f, -1.0f, -1.0f);

    // Finde die x-Komponente
    if (m_vcOrig.x < aabb.vcMin.x) {
        (*vcHit).x = aabb.vcMin.x;
        bInside = false;
        if (m_vcDir.x != 0.0f)
            MaxT.x = (aabb.vcMin.x - m_vcOrig.x) /
                    m_vcDir.x;
    }
    else if (m_vcOrig.x > aabb.vcMax.x) {
        (*vcHit).x = aabb.vcMax.x;
        bInside = false;
        if (m_vcDir.x != 0.0f)
            MaxT.x = (aabb.vcMax.x - m_vcOrig.x) /
                    m_vcDir.x;
    }
    // Findw die y-Komponente
    if (m_vcOrig.y < aabb.vcMin.y) {
        (*vcHit).y = aabb.vcMin.y;
        bInside = false;
        if (m_vcDir.y != 0.0f)
            MaxT.y = (aabb.vcMin.y - m_vcOrig.y) /
                    m_vcDir.y;
    }
    else if (m_vcOrig.y > aabb.vcMax.y) {
        (*vcHit).y = aabb.vcMax.y;
        bInside = false;
        if (m_vcDir.y != 0.0f)
            MaxT.y = (aabb.vcMax.y - m_vcOrig.y) /
                    m_vcDir.y;
    }
}
```



```

    }
    // Finde die z-Komponente
    if (m_vcOrig.z < aabb.vcMin.z) {
        (*vcHit).z = aabb.vcMin.z;
        bInside = false;
        if (m_vcDir.z != 0.0f)
            MaxT.z = (aabb.vcMin.z - m_vcOrig.z) /
                    m_vcDir.z;
    }
    else if (m_vcOrig.z > aabb.vcMax.z) {
        (*vcHit).z = aabb.vcMax.z;
        bInside = false;
        if (m_vcDir.z != 0.0f)
            MaxT.z = (aabb.vcMax.z - m_vcOrig.z) /
                    m_vcDir.z;
    }

    // Strahl-Ursprung innerhalb der Box
    if (bInside) {
        (*vcHit) = m_vcOrig;
        return true;
    }

    // Größter Wert von MaxT
    int nPlane = 0;

    if (MaxT.y > ((float*)&MaxT)[nPlane]) nPlane = 1;
    if (MaxT.z > ((float*)&MaxT)[nPlane]) nPlane = 2;

    if ( ((float*)&MaxT)[nPlane] < 0.0f) return false;

    if (nPlane != 0) {
        (*vcHit).x = m_vcOrig.x + MaxT.x * m_vcDir.x;
        if ( ((*vcHit).x < aabb.vcMin.x-0.00001f) ||
            ((*vcHit).x < aabb.vcMax.x+0.00001f) )
            return false;
    }
    if (nPlane != 1) {
        (*vcHit).y = m_vcOrig.y + MaxT.y * m_vcDir.y;
        if ( ((*vcHit).y < aabb.vcMin.y-0.00001f) ||
            ((*vcHit).y < aabb.vcMax.y+0.00001f) )
            return false;
    }
    if (nPlane != 0) {
        (*vcHit).z = m_vcOrig.z + MaxT.z * m_vcDir.z;
        if ( ((*vcHit).z < aabb.vcMin.z-0.00001f) ||
            ((*vcHit).z < aabb.vcMax.z+0.00001f) )
            return false;
    }
    return true;
} // Intersects(Aabb)

```

Die Funktion sieht deshalb ein wenig langgestreckt aus, weil ich die Schleifen über die drei Komponenten x, y und z eines Vektors hier auseinander gezogen habe, anstatt auf die Komponenten über ein `float`-Array und einen Schleifenindex zuzugreifen. Dadurch wird der Code ein wenig länger, aber man spart sich den Overhead für eine Schleife, die bei nur drei Durchläufen nicht unbedingt sinnvoll ist. Nun verlassen wir die AABB auch schon wieder und wenden uns den etwas eleganteren Boxen zu – nämlich den OBB.

In unserer Engine werden wir, insbesondere bei den einzelnen 3D-Modellen, mit orientierten Bounding-Boxen arbeiten, die eine beliebige Ausrichtung in Bezug auf die Weltkoordinaten-Achsen in unserer 3D-Engine annehmen können. Dafür schreiben wir nachher noch eine Klasse `ZFXObb`. Als Kollisionstest zwischen diesen beiden Boxen verwenden wir die so genannte Slaps-Methode, die hinreichend bekannt sein sollte. Eine Beschreibung und Implementierung findet ihr unter anderem in dem Buch von Möller und Haines⁸ sowie in meinem zweiten Buch.⁹ Die Klasse `ZFXObb` beschreibt eine solche orientierte Bounding-Box, und ihre Definition und Implementierung werden wir später noch sehen. Hier seht ihr erst einmal die Kollisionsfunktion. Diese übernimmt eine solche OBB und einen Zeiger auf einen `float`-Wert. Falls der Strahl die OBB schneidet, wird dieser Wert mit der Distanz von dem Strahlursprung bis zu dem Schnittpunkt gefüllt.

*Schnittpunkt
eines Strahls mit
einer OBB*

```
bool ZFXRay::Intersects(const ZFXObb *pObb, float *t) {
    float e, f, t1, t2, temp;
    float tmin = -99999.9f,
           tmax = +99999.9f;

    ZFXVector vcP = pObb->vcCenter - m_vcOrig;

    // 1. Slap
    e = pObb->vcA0 * vcP;
    f = pObb->vcA0 * m_vcDir;
    if (_fabs(f) > 0.00001f) {

        t1 = (e + pObb->fA0) / f;
        t2 = (e - pObb->fA0) / f;

        if (t1 > t2) { temp=t1; t1=t2; t2=temp; }
        if (t1 > tmin) tmin = t1;
        if (t2 < tmax) tmax = t2;
        if (tmin > tmax) return false;
        if (tmax < 0.0f) return false;
    }
    else if ( ((-e - pObb->fA0) > 0.0f) ||
              ((-e + pObb->fA0) < 0.0f) )
        return false;
}
```

}
LIST

⁸ Real-Time Rendering; A K Peters, Ltd., 1999

⁹ 3D Spieleprogrammierung mit DirectX in C/C++ – Band II; BOD, 2002

```

// 2. Slap
e = p0bb->vcA1 * vcP;
f = p0bb->vcA1 * m_vcDir;
if (_fabs(f) > 0.00001f) {

    t1 = (e + p0bb->fA1) / f;
    t2 = (e - p0bb->fA1) / f;

    if (t1 > t2) { temp=t1; t1=t2; t2=temp; }
    if (t1 > tmin) tmin = t1;
    if (t2 < tmax) tmax = t2;
    if (tmin > tmax) return false;
    if (tmax < 0.0f) return false;
}
else if ( ((-e - p0bb->fA1) > 0.0f) ||
          ((-e + p0bb->fA1) < 0.0f) )
    return false;

// 3. Slap
e = p0bb->vcA2 * vcP;
f = p0bb->vcA2 * m_vcDir;
if (_fabs(f) > 0.00001f) {

    t1 = (e + p0bb->fA2) / f;
    t2 = (e - p0bb->fA2) / f;

    if (t1 > t2) { temp=t1; t1=t2; t2=temp; }
    if (t1 > tmin) tmin = t1;
    if (t2 < tmax) tmax = t2;
    if (tmin > tmax) return false;
    if (tmax < 0.0f) return false;
}
else if ( ((-e - p0bb->fA2) > 0.0f) ||
          ((-e + p0bb->fA2) < 0.0f) )
    return false;

if (tmin > 0.0f) {
    if (t) *t = tmin;
    return true;
}
if (t) *t = tmax;
return true;
}

```

**Schnittpunkt
eines Strahlseg-
ments mit einer
OBB**

Fast analog zu dieser Methode können wir auch eine Prüfung auf einen Schnitt zwischen einer OBB und einem Segment durchführen. Ein Segment haben wir ja ein paar Zeilen weiter oben schon kennen gelernt. Es ist sozusagen ein Strahl, der nur eine bestimmte Länge hat. Von seinem Ursprung aus läuft er in eine bestimmte Richtung über eine bestimmte Distanz und

endet dort, während ein normaler Strahl bis in die Unendlichkeit weiterlaufen würde. Den Kollisionstest für ein solches Segment mit einer OBB erstellen wir nach demselben Prinzip wie bei der Kollision mit einem Dreieck. Wir nehmen die vorhandene Funktion für die Kollision eines kompletten Strahls mit einer OBB und integrieren am Ende der Funktion noch den Test, ob die berechnete Distanz zu dem Schnittpunkt des Strahls mit der OBB, sofern vorhanden, maximal so groß ist wie die angegebene Länge des zu betrachtenden Strahlsegments. Also ergänzen wir auch hier im Prototyp einen Wert für die Länge des Segments:

```
bool ZFXRay::Intersects(const ZFXObb *pObb,
                        float fL, float *t);
```

Wie gewohnt findet ihr auch diese Implementierung komplett auf der CD zu diesem Buch. Wir hingegen verlassen hier die Strahlen und beschäftigen uns ab jetzt mit Ebenen.

4.5 Rechnen mit Ebenen

Eine Ebene (Plane) ist wieder so ein mathematisches *Ding*, dessen Konzept einem am Anfang etwas verwirrend erscheinen mag. Dazu benötigen wir wieder die ohnmächtig gewordene Zahl Acht: **symbUnendlich**. Stellen wir uns eine unbiegsame Plexiglasplatte vor. Diese Platte symbolisiert eine Ebene, nur mit dem Unterschied, dass Ebenen keine begrenzte Fläche haben. Vielmehr ist ihre Fläche unendlich groß, denn die Ebene hat nur im Unendlichen einen Rand. Aufgespannt wird eine Ebene durch zwei beliebige Vektoren, die jedoch nicht parallel sein dürfen, sonst würde die Ebene zu einer Linie kollabieren.

Ebenen

So gesehen ist eine Ebene quasi eine Art Scheibchen des Raumes, das unendlich dünn ist, also keine messbare Stärke hat. Wie können wir ein solches Gebilde fassen und beschreiben? Dazu gibt es in der Mathematik mehrere Möglichkeiten. Für uns im Bereich der 3D-Mathematik ist es am zweckmäßigsten, die folgende Darstellung zu verwenden:

Ebenen-Formel

$$\mathbf{V} * \mathbf{N} + d = 0$$

Zäumen wir das Pferd, sprich: die Formel, mal von hinten auf: Das Symbol d ist ein einfacher Fließkommazahl-Wert, der den Abstand der Ebene vom Nullpunkt des Bezugskoordinatensystems beschreibt. Das Symbol $\mathbf{N}$ steht in der Formel für den (normalisierten) Normalenvektor der Ebene. Eine Ebene hat eine Vorderseite und eine Rückseite, und um die beiden auseinander zu halten, verwendet man auch den Normalenvektor. Die Seite der Ebene, die in dieselbe Richtung wie der Normalenvektor *schaut*, ist die Vorderseite der Ebene.

Normalenvektoren

Ein Normalenvektor ist ein Vektor, der rechtwinklig zu einem Objekt steht und die Länge 1.0 hat, also normalisiert ist. Dabei hat der Begriff Normalenvektor nichts mit dem Attribut normalisiert zu tun, die Ähnlichkeit der beiden Begriffe lässt vielleicht anderes vermuten.

Ein Normalenvektor zu einer Ebene beispielsweise lässt sich sehr leicht finden. Wir bilden einfach das Kreuzprodukt der beiden Vektoren, die die Ebene aufspannen, oder zweier beliebiger verschiedener, nicht-paralleler Vektoren, die in der Ebene liegen. Die Distanz d einer Ebene zum Ursprung ist übrigens immer dann negativ, wenn der Normalenvektor der Ebene vom Ursprung wegzeigt, sonst hat sie ein positives Vorzeichen.

Die Variable v in der Ebenen-Formel steht für den Vektor zu einem beliebigen Punkt in/auf der Ebene. Sind wir uns beispielsweise bei einem Punkt unsicher, ob er in der Ebene liegt, setzen wir ihn als v in die Ebenen-Formel ein. Dann bilden wir das Punktprodukt mit dem Normalenvektor und addieren den Wert von d . Ergibt diese Rechnung den Wert 0, so liegt der Punkt in der Ebene, bei jedem anderen Ergebnis nicht.

Klassifizierung von Punkten

Aus jedem anderen Ergebnis können wir auch noch weitere wichtige Schlüsse ziehen. Ist das Ergebnis kleiner als 0, so liegt der Punkt irgendwo hinter der Ebene; ist das Ergebnis größer als 0, so liegt der Punkt irgendwo vor der Ebene. Diese Klassifizierung von Punkten brauchen wir beispielsweise sehr häufig bei der Arbeit mit BSP-Bäumen. Jetzt haben wir aber erst mal genug Theorie über diese Ebene gehört. Also beginnen wir mit der Implementierung.

Definition der Klasse ZFXPlane



```
class __declspec(dllexport) ZFXPlane {
public:
    ZFXVector m_vcN,           // Normalenvektor
               m_vcPoint;      // Punkt auf der Ebene
    float      m_fD;           // Entfernung Ursprung

    ZFXPlane(void) { /* nothing to do */ ; }

    inline void Set(const ZFXVector &vcN,
                   const ZFXVector &vcPoint);
    inline void Set(const ZFXVector &vcN,
                   const ZFXVector &vcPoint,
                   float fD);
    inline void Set(const ZFXVector &v0,
                   const ZFXVector &v1,
                   const ZFXVector &v2);

    // Entfernung eines Punktes zur Ebene
```

```

inline float Distance(const ZFXVector &vcPoint);

// Klassifizierung eines Punktes
inline int Classify(const ZFXVector &vcPoint);

// Kollision mit einem Dreieck
bool Intersects(const ZFXVector &vc0,
               const ZFXVector &vc1,
               const ZFXVector &vc2);

// Schnittgerade zweier Ebenen
bool Intersects(ZFXPlane &plane,
               ZFXRay *pIntersection);

// Kollision mit AABB oder OBB
bool Intersects(const ZFXAabb &aabb);
bool Intersects(const ZFXObb &obb);
}; // class

```

An Funktionalität benötigen wir auch bei einer Ebene nicht viel. Neben der Klassifizierung eines Punktes zur Ebene möchten wir in einigen Fällen auch den Abstand eines Punktes zur Ebene berechnen. Die letzte, mehrfach überladene Funktion dient dazu, auf Schnitte zwischen einer Ebene und anderen Objekten zu testen. Diese Funktionalität implementieren wir für Ebene-Dreieck, Ebene-Ebene und Ebene-Bounding-Box (AABB und OBB, siehe unten). Man hätte natürlich auch hier die Funktion für die Kollision zwischen Ebenen und Strahlen implementieren können. Diese haben wir jedoch schon als Member der Klasse `ZFXRay` und werden sie hier nicht noch einmal zeigen.

Grundlegende Operationen

Die grundlegenden Operationen für Ebenen ergeben sich zumeist wie von selbst aus der oben aufgeführten Ebenen-Formel. Eine Herleitung erspare ich mir hier, da wir uns ja bereits mit 3D-Mathematik gut auskennen.

Ohne Worte

```

#define ZFXFRONT    0
#define ZFXBACK     1
#define ZFXPLANAR   2

inline void ZFXPlane::Set(const ZFXVector &vcN,
                        const ZFXVector &vcPoint) {
    m_fD      = - ( vcN * vcPoint);
    m_vcN     = vcN;
    m_vcPoint = vcPoint;
}

/*-----*/

```



```

inline void ZFXPlane::Set(const ZFXVector &vcN,
                          const ZFXVector &vcPoint,
                          float fD) {
    m_vcN    = vcN;
    m_fD     = fD;
    m_vcPoint = vcPoint;
}
/*-----*/

inline void ZFXPlane::Set(const ZFXVector &v0,
                          const ZFXVector &v1,
                          const ZFXVector &v2) {
    ZFXVector vcEdge1 = v1 - v0;
    ZFXVector vcEdge2 = v2 - v0;

    m_vcN.Cross(vcEdge1, vcEdge2);
    m_fD = m_vcN * v0;
}
/*-----*/

// Berechne die Entfernung Punkt-Ebene; der Ebenen-
// Normalenvektor muss normalisiert sein.
inline float ZFXPlane::Distance(const ZFXVector &vcP) {
    return ( _fabs((m_vcN*vcP) - m_fD) );
}
/*-----*/

// Klassifiziere Punkt zur Ebene.
inline int ZFXPlane::Classify(const ZFXVector &vcP) {
    float f = (vcP * m_vcN) + m_fD;

    if (f > 0.00001) return ZFXFRONT;
    if (f < -0.00001) return ZFXBACK;
    return ZFXPLANAR;
}
/*-----*/

```

Gehen wir daher gleich weiter zu den etwas komplexeren Operationen für Ebenen im 3D-Raum, die man nicht ganz so häufig antrifft. Das sind natürlich die Kollisionsfunktionen für Ebenen mit anderen Objekten im 3D-Raum. Diese Operationen brauchen wir beispielsweise dann sehr oft, wenn wir unsere Level-Geometrie in einem Scene-Management mit einem bestimmten Algorithmus anordnen.

Kollision mit Dreiecken

Banal! Die Kollision eines Dreiecks mit einer Ebene ist sehr einfach auszurechnen. Wir haben bereits eine Funktion, mit der wir Punkte in Bezug auf eine Ebene klassifizieren können, also ob ein Punkt vor oder hinter oder auf

einer Ebene liegt. Eine Ebene schneidet ein Dreieck logischerweise nur dann, wenn nicht alle drei Punkte des Dreiecks in der Ebene oder auf derselben Seite der Ebene liegen.

```
bool ZFXPlane::Intersects(const ZFXVector &vc0,
                          const ZFXVector &vc1,
                          const ZFXVector &vc2) {
    int n = this->Classify(vc0);

    if ( (n == this->Classify(vc1)) &&
        (n == this->Classify(vc2)) )
        return false;
    return true;
} // Intersects(Tri)
```



Diese Funktion interpretiert eine Berührung eines oder zweier Punkte des Dreiecks mit der Ebene auch schon als Kollision. Das ist schneller zu berechnen, als wenn wir die einzelnen Rückgabewerte auch noch gegen das `#define ZFXPLANAR` prüfen müssten. Unter Umständen können andere Anwendungen hier andere Implementierungen benötigen. Möchte man auch noch die Schnittgerade der beiden Objekte haben, dann muss man die Ebene des Dreiecks bestimmen und die Funktion aus dem folgenden Abschnitt verwenden.

Aber Vorsicht!

Kollision zwischen Ebenen

Die Kollision zwischen zwei Ebenen lässt sich auch relativ simpel berechnen. Der folgende Code basiert auf einer Implementierung von David Eberly, auf dessen Internetseiten¹⁰ sich viele interessante Funktionen finden. Der eigentliche Test auf einen Schnitt ist dabei sehr simpel: Da sich Ebenen im Raum unendlich weit ausdehnen, gibt es nur einen Fall, in dem sich zwei Ebenen *nicht* schneiden – und zwar genau dann, wenn sie parallel sind. Das können wir über die Berechnung des Kreuzprodukts der beiden Normalenvektoren herausfinden, das den Nullvektor ergibt, wenn beide parallel sind. Etwas aufwändiger ist die Berechnung der Schnittgeraden im Falle einer Kollision.

```
bool ZFXPlane::Intersects(const ZFXPlane &plane,
                          ZFXRay *pIntersection) {
    ZFXVector vcCross;
    float      fSqrLength;

    // Wenn Kreuzprodukt=Nullvektor, dann Ebenen parallel
    vcCross.Cross(this->m_vcN, plane.m_vcN);
    fSqrLength = vcCross.GetSqrLength();
```



¹⁰ www.magic-software.com

```

if (fSqrLength < 1e-08f)
    return false;

// Schnittlinie, falls gewünscht
if (pIntersection) {
    float fN00 = this->m_vcN.GetSqrLength();
    float fN01 = this->m_vcN * plane.m_vcN;
    float fN11 = plane.m_vcN.GetSqrLength();
    float fDet = fN00*fN11 - fN01*fN01;

    if (_fabs(fDet) < 1e-08f)
        return false;

    float fInvDet = 1.0f/fDet;
    float fC0 = (fN11*this->m_fD - fN01*plane.m_fD) *
                fInvDet;
    float fC1 = (fN00*plane.m_fD - fN01*this->m_fD) *
                fInvDet;

    (*pIntersection).m_vcDir = vcCross;
    (*pIntersection).m_vcOrig = this->m_vcN * fC0
                               + plane.m_vcN * fC1;
}
return true;
} // Intersects(Plane)

```

Wie bereits weiter oben erwähnt, sollte man die Schnittgerade wirklich nur dann berechnen lassen, wenn man sie wirklich benötigt. Anderenfalls bremst sie die Funktion unnötig aus. Und wir wollen ja so schnell wie möglich bleiben.

Kollision mit Bounding-Boxen

AABB und OBB

Die beiden letzten Objekte, die wir auf eine Kollision mit einer Ebene hin testen wollen, sind Bounding-Boxen – und zwar achsen-ausgerichtete (AABB) und orientierte Boxen (OBB). Wie so oft ist die Funktion für die eigentlich simpleren AABB um einiges länger als die für orientierte Boxen. Davon sollte man sich aber nicht täuschen lassen, da der Code für die achsen-ausgerichteten Boxen selten aus mehr als nur `if`-Vergleichen besteht. Dass der Code für orientierte Boxen so kurz ausfällt, verdanken wir nicht zuletzt unseren überladenen Operatoren der `ZFXVector`-Klasse. Aber die dahinter stehenden Berechnungen sind viel zeitaufwändiger.

Beginnen wir mit dem Code für die Kollision einer Ebene mit einer achsen-ausgerichteten Box. Eine Implementierung dazu findet ihr im Buch von Möller und Haines.¹¹

¹¹ Real-Time Rendering; A K Peters, Ltd., 1999



```

bool ZFXPlane::Intersects(const ZFXAabb &aabb) {
    ZFXVector Vmin, Vmax;

    // x-Komponente
    if (m_vcN.x >= 0.0f) {
        Vmin.x = aabb.vcMin.x;
        Vmax.x = aabb.vcMax.x;
    }
    else {
        Vmin.x = aabb.vcMax.x;
        Vmax.x = aabb.vcMin.x;
    }

    // y-Komponente
    if (m_vcN.y >= 0.0f) {
        Vmin.y = aabb.vcMin.y;
        Vmax.y = aabb.vcMax.y;
    }
    else {
        Vmin.y = aabb.vcMax.y;
        Vmax.y = aabb.vcMin.y;
    }

    // z-Komponente
    if (m_vcN.z >= 0.0f) {
        Vmin.z = aabb.vcMin.z;
        Vmax.z = aabb.vcMax.z;
    }
    else {
        Vmin.z = aabb.vcMax.z;
        Vmax.z = aabb.vcMin.z;
    }

    if ( ((m_vcN * Vmin) + m_fd) > 0.0f)
        return false;

    if ( ((m_vcN * Vmax) + m_fd) >= 0.0f)
        return true;

    return false;
} // Intersects(AABB)

```

Wir müssen hier einfach anhand des Normalenvektors der Ebene komponentenweise einen minimalen Punkt und einen maximalen Punkt aus den beiden Extrempunkten (maximale und minimale Ausdehnung auf den beiden Achsen) der AABB zusammenbasteln. Diese beiden Punkte testen wir dann abschließend gegen die Ebenen-Formel. Liegt der minimale Punkt schon vor der Ebene, dann gilt dies für den maximalen Punkt auch, und es kann keinen Schnitt geben. Trifft dies aber nicht zu, dann liegt der minimale

*Lang, aber doch
einfach*

Punkt hinter der Ebene. Liegt der maximale Punkt dann vor der Ebene, muss es einen Schnitt geben. Anderenfalls ist das Ergebnis auch wieder `false`, weil dann beide temporären Extrempunkte der Box hinter der Ebene liegen.

Und OBB? Der Code für den Kollisionstest einer orientierten Box mit einer Ebene ist erschreckend kurz. Aber das liegt, wie bereits gesagt, an den überladenen Operatoren unserer Vektoren:

```
bool ZFXPlane::Intersects(const ZFXObb &obb) {
    float fRadius = _fabs( obb.fA0 * (m_vcN*obb.vcA0) )
        + _fabs( obb.fA1 * (m_vcN*obb.vcA1) )
        + _fabs( obb.fA2 * (m_vcN*obb.vcA2) );

    float fDistance = this->Distance(obb.vcCenter);
    return (fDistance <= fRadius);
} // Intersects(OBB)
```

Was wir hier tun, das entspricht, geometrisch betrachtet, einer Projektion der Box und der Ebene auf eine Linie. Dabei berechnen wir den Radius, den die Box bei dieser Projektion haben würde, und berechnen die Entfernung des Box-Mittelpunktes zu der Ebene. Ist diese Entfernung größer als der Radius der Box, so kann es keine Kollision geben. Dieser Test, ebenso wie der für die achsen-ausgerichteten Boxen, wird uns gleich wiederbegegnen. Nämlich dann, wenn wir Funktionen für das Culling von Bounding-Boxen gegen den View-Frustum schreiben. Ein Frustum ist schließlich auch nur eine Ansammlung von Ebenen.

4.6 Rechnen mit AABB und OBB

In unserer *ZFXEngine* werden wir sowohl achsen-ausgerichtete als auch orientierte Bounding-Boxen verwenden. Ich setze hier voraus, dass jeder mit diesen Begriffen etwas anfangen kann. Wir benötigen also zwei Klassen für diese beiden Objekt-Arten, denn die Repräsentation einer AABB unterscheidet sich grundlegend von der einer OBB. Bei der OBB müssen wir die drei orthogonalen Achsen der Box angeben, die ja von der Orientierung der Box im Raum abhängen. Zusätzlich benötigen wir natürlich noch zu jeder Achse die Ausdehnung, die die Box in beide Richtungen auf dieser Achse hat. Für einige Berechnungen ist es zudem erforderlich, den Mittelpunkt der Box zu kennen.

Extrempunkte Bei einer AABB hingegen reicht es aus, wenn wir die beiden so genannten Extrempunkte angeben. Dazu bestimmen wir einfach auf jeder Komponente (x, y und z) den jeweils kleinsten und größten Wert und speichern diese beiden Punkte für die Box. Hier ist die Klassen-Definition für die achsen-ausgerichteten Boxen:

```

class __declspec(dllexport) ZFXAabb {
public:
    ZFXVector vcMin, vcMax; // Extrem-Punkte
    ZFXVector vcCenter;    // Mittelpunkt

    //-----

    ZFXAabb(void) { /* nothing to do */ ; }

    void Construct(const ZFXObb *pObb);
    int  Cull(const ZFXPlane *pPlanes,
              int nNumPlanes);

    void GetPlanes(ZFXPlane *pPlanes);
    bool Contains(const ZFXRay &Ray, float fL);
}; // class

```



Und hier für die orientierten Boxen. Grundsätzlich ist es zwar möglich, achsen-ausgerichtete Boxen auch als OBB zu speichern, dann würden wir aber unnötig viele Daten im Speicher halten, die wir für AABB nicht benötigen. Umgekehrt ist dies nicht möglich, denn eine OBB kann man nicht als AABB repräsentieren.

*AABB auch als
OBB*

```

class __declspec(dllexport) ZFXObb {
public:
    float    fA0, fA1, fA2; // Halbe Achsen Länge
    ZFXVector vcA0, vcA1, vcA2; // Achsen
    ZFXVector vcCenter;    // Mittelpunkt

    //-----

    ZFXObb(void) { /* nothing to do */ ; }

    inline void DeTransform(const ZFXObb &obb,
                           const ZFXMatrix &m);

    bool Intersects(const ZFXRay &Ray, float *t);
    bool Intersects(const ZFXRay Ray, float fL,
                    float *t);

    bool Intersects(const ZFXObb &Obb);
    bool Intersects(const ZFXVector v0,
                    const ZFXVector v1,
                    const ZFXVector v2);

    int  Cull(const ZFXPlane *pPlanes,
              int nNumPlanes);

private:

```



```

void ObbProj(const ZFXObb &Obb,
             const ZFXVector &vcV,
             float *pfMin, float *pfMax);
void TriProj(const ZFXVector &v0,
             const ZFXVector &v1,
             const ZFXVector &v2,
             const ZFXVector &vcV,
             float *pfMin, float *pfMax);

}; // class

```

Die Klasse für achsen-ausgerichtete Bounding-Boxen bietet uns keine neue Funktionalität, was die Kollisionsabfragen betrifft. Der Grund dafür ist ganz einfach, dass diese Funktionalität, soweit sie benötigt wird, in anderen Klassen abgedeckt ist, in diesem Fall in `ZFXRay` und `ZFXPlane`. Ein Großteil unserer Engine wird ja mit den orientierten Boxen arbeiten, daher steckt in der `ZFXObb`-Klasse wieder eine entsprechend überladene Methode für die Berechnung von Kollisionen. Neu sind hier die Kollision einer orientierten Box mit einem Dreieck und die Kollision zweier orientierter Boxen. Die Kollision mit einem Strahl kennen wir bereits aus der Klasse `ZFXRay`. Hier ist sie nochmals implementiert, um über beiden Datenstrukturen auf diese Funktionalität zugreifen zu können.



Als Culling bezeichnet man das Entfernen von Teilen aus dem Rahmen der Betrachtung, die für eine gegebene Situation nicht relevant sind. In diesem Kontext meint man damit das Aussortieren von Objekten aus der Berechnung eines Frames, die aufgrund des Sichtwinkels des Betrachters nicht in seinem Blickfeld liegen.

Grundlegende Operationen und Culling

AABB

Beginnen wir mit den einfacheren achsen-ausgerichteten Boxen. Hier haben wir nur zwei Funktionen. Die erste berechnet aus einer OBB eine AABB. Dazu muss man nur die Werte der AABB ein wenig herumschubsen und die Ausdehnung der Box auf den drei Standard-Weltachsen berechnen. Das kann nötig sein, wenn wir eine orientierte Box in einer bestimmten Rechnung zur Vereinfachung als achsen-ausgerichtet behandeln wollen.



```

void ZFXAabb::Construct(const ZFXObb &Obb) {
    ZFXVector vcA0, vcA1, vcA2;
    ZFXVector _vcMax, _vcMin;

    vcA0 = Obb.vcA0 * Obb.fA0;
    vcA1 = Obb.vcA1 * Obb.fA1;
    vcA2 = Obb.vcA2 * Obb.fA2;

    if (vcA0.x > vcA1.x) {
        if (vcA0.x > vcA2.x)

```

```

        { vcMax.x = vcA0.x; vcMin.x = -vcA0.x; }
    else { vcMax.x = vcA2.x; vcMin.x = -vcA2.x; }
    }
else {
    if (vcA1.x > vcA2.x)
        { vcMax.x = vcA1.x; vcMin.x = -vcA1.x; }
    else { vcMax.x = vcA2.x; vcMin.x = -vcA2.x; }
    }
if (vcA0.y > vcA1.y) {
    if (vcA0.y > vcA2.y) {
        { vcMax.y = vcA0.y; vcMin.y = -vcA0.y; }
    else { vcMax.y = vcA2.y; vcMin.y = -vcA2.y; }
    }
else {
    if (vcA1.y > vcA2.y)
        { vcMax.y = vcA1.y; vcMin.y = -vcA1.y; }
    else { vcMax.y = vcA2.y; vcMin.y = -vcA2.y; }
    }
if (vcA0.z > vcA1.z) {
    if (vcA0.z > vcA2.z)
        { vcMax.z = vcA0.z; vcMin.z = -vcA0.z; }
    else { vcMax.z = vcA2.z; vcMin.z = -vcA2.z; }
    }
else {
    if (vcA1.z > vcA2.z)
        { vcMax.z = vcA1.z; vcMin.z = -vcA1.z; }
    else { vcMax.z = vcA2.z; vcMin.z = -vcA2.z; }
    }
} // construct

```

Eine elementare Funktion für jedes Bounding-Volume ist eine Funktion für das Culling gegen den View-Frustum oder eine andere Konstruktion aus Clipping-Planes im 3D-Raum. Diese Funktion ist fast analog zu dem Kollisionstest zwischen Ebenen und achsen-ausgerichteten Boxen. Aber eben nur fast. Hier testen wir nicht nur gegen eine Ebene, sondern gegen eine Vielzahl von Ebenen. Damit sich das Objekt innerhalb des Raumes befindet, den die Ebenen abstecken, darf die AABB sich nicht auf der Vorderseite auch nur einer einzigen Ebene befinden.

AABB-Culling

Wir gehen beim Culling gegen eine Anzahl von Clipping-Planes immer davon aus, dass die Normalenvektoren der Ebenen nach außen zeigen, wenn man den Raum, den die Ebenen abgrenzen, als Innen beschreibt.

**!!
STOP**

Damit eine AABB nicht dem Culling unterworfen wird, muss sie mindestens eine der Ebenen schneiden und darf auf keinen Fall auf der Vorderseite auch nur einer der Ebenen sein. Die folgende Funktion berechnet wie gehabt die temporären Extrempunkte in Relation zu der jeweiligen Ebene und testet dann die Kollision.



```

#define ZFXCLIPPED 3
#define ZFXCULLED 4
#define ZFXVISIBLE 5

int ZFXAabb::Cull(const ZFXPlane *pPlanes,
                 int nNumPlanes) {
    ZFXVector vcMin, vcMax;
    bool bIntersects = false;

    // Bastle und teste Extrempunkte
    for (int i=0; i<nNumPlanes; i++) {
        if (pPlanes[i].m_vcN.x >= 0.0f) {
            vcMin.x = this->vcMin.x;
            vcMax.x = this->vcMax.x;
        }
        else {
            vcMin.x = this->vcMax.x;
            vcMax.x = this->vcMin.x;
        }
        if (pPlanes[i].m_vcN.y >= 0.0f) {
            vcMin.y = this->vcMin.y;
            vcMax.y = this->vcMax.y;
        }
        else {
            vcMin.y = this->vcMax.y;
            vcMax.y = this->vcMin.y;
        }
        if (pPlanes[i].m_vcN.z >= 0.0f) {
            vcMin.z = this->vcMin.z;
            vcMax.z = this->vcMax.z;
        }
        else {
            vcMin.z = this->vcMax.z;
            vcMax.z = this->vcMin.z;
        }

        if ( ((pPlanes[i].m_vcN*vcMin) + pPlanes[i].m_fd)
             > 0.0f)
            return ZFXCULLED;

        if ( ((pPlanes[i].m_vcN*vcMax) + pPlanes[i].m_fd)
             >= 0.0f)
            bIntersects = true;
    } // for
    if (bIntersects) return ZFXCLIPPED;
    return ZFXVISIBLE;
} // cull

```

**OBb transfor-
mieren**

Damit verlassen wir das Gebiet der achsen-ausgerichteten Boxen auch schon wieder und wenden uns dem thematisch letzten Teil dieses Kapitels

zu, nämlich der Arbeit mit orientierten Boxen. Beginnen werden wir mit der Funktion, die eine Box in ein bestimmtes Koordinatensystem transformiert. In der Regel werden wir es zwar umgekehrt machen und beispielsweise Strahlen in das Koordinatensystem einer OBB transformieren. Aber wenn man den umgekehrten Weg auch zur Auswahl hat, kann das ja nicht schaden. Die Berechnung läuft eigentlich analog der Berechnung bei den Strahlen ab, nur eben auf ein paar Vektoren mehr angewendet.

```
inline void ZFXOBB::DeTransform(const ZFXOBB &obb,
                                const ZFXMatrix &m)
{
    ZFXMatrix mat = m;
    ZFXVector vcT;

    // lösche Verschiebung aus mat
    vcT.Set(mat._41, mat._42, mat._43);
    mat._41 = mat._42 = mat._43 = 0.0f;

    // Rotiere Vektoren zu Matrix-Koord.Sys.
    this->vcCenter = mat * obb.vcCenter;
    this->vcA0      = mat * obb.vcA0;
    this->vcA1      = mat * obb.vcA1;
    this->vcA2      = mat * obb.vcA2;

    // Verschiebung für Mittelpunkt
    this->vcCenter += vcT;

    // Werte übernehmen
    fA0 = obb.fA0;
    fA1 = obb.fA1;
    fA2 = obb.fA2;
} // Transform
```



Auch das Culling der orientierten Boxen ist für uns nichts wirklich Neues mehr. Schließlich ist das Culling der OBB nichts anderes als eine Kollisionsabfrage der OBB gegen eine Anzahl von Clipping-Planes, die einen Bereich des Raumes (beispielsweise das View-Frustum) eingrenzen. Allerdings gibt es hier, im Vergleich zu dem Kollisionstest zwischen einer Ebene und einer OBB, eine Sache zu berücksichtigen. Aber dazu gleich.

OBB-Culling

```
int ZFXOBB::Cull(const ZFXPlane *pPlanes,
                 int nNumPlanes) {
    ZFXVector vN;
    int      nResult = ZFXVISIBLE;
    float     fRadius, fTest;

    // Für alle Ebenen
    for (int i=0; i<nNumPlanes; i++) {
        // Normalen nach innen biegen
```



```

vN = pPlanes[i].m_vcN * -1.0f;

// Box-Radius berechnen
fRadius = _fabs(fA0 * (vN * vcA0))
          + _fabs(fA1 * (vN * vcA1))
          + _fabs(fA2 * (vN * vcA2));

// Referenzwert: (N*C - d) (#)
fTest = vN * this->vcCenter - pPlanes[i].m_fd;

// OBB jenseits der Ebene: (#) < -r
if (fTest < -fRadius) return ZFXCULLED;

// oder Schnitt mit Ebene?
else if (!(fTest > fRadius)) nResult = ZFXCLIPPED;
} // for

return nResult;
}

```

Hier dürfen wir nicht die Funktion `ZFXPlane::Distance` verwenden. Diese würde uns die Entfernung zwischen der Ebene und dem Mittelpunkt der Box als absoluten Wert zurückgeben. Bei der Kollisionsfunktion, wo wir das so verwendet haben, war das in Ordnung, weil es uns nur darauf ankam, ob die Box die Ebene schneidet. Hier ist es wichtig, auf welcher Seite der Ebene die Box liegt, falls sie die Box nicht schneidet. Nur wenn die Box bei wenigstens einer Ebene komplett auf der Rückseite liegt, können wir ein Culling durchführen.

Kollision mit Dreiecken

Quellen Die Kollision zwischen einer orientierten Box und einem Dreieck lässt sich am einfachsten über die Methode der Separationsachsen herausfinden. Diese Implementierung findet ihr unter anderem auf der Internetseite¹² von David Eberly und in seinem Buch.¹³

Aus diesem Grund werde ich die Funktion hier nicht weiter kommentieren, sondern einfach in leichter Variation ablichten. Auf der eben genannten Internetseite findet sich auch ein Papier über die Grundlagen der Technik von Separationsachsen, das ich nur wärmstens empfehlen kann.

¹² www.magic-software.com

¹³ 3D Game Engine Design; Morgan Kaufmann Publishers, 2000



```
// Hilfsfunktion
void ZFXObb::ObbProj(const ZFXObb &Obb,
                    const ZFXVector &vcV,
                    float *pfMin, float *pfMax) {
    float fDP = vcV * Obb.vcCenter;
    float fR = Obb.fA0 * _fabs(vcV * Obb.vcA0) +
              Obb.fA0 * _fabs(vcV * Obb.vcA1) +
              Obb.fA1 * _fabs(vcV * Obb.vcA2);
    *pfMin = fDP - fR;
    *pfMax = fDP + fR;
} // ObbProj

/*-----*/
```

```
// Hilfsfunktion
void ZFXObb::TriProj(const ZFXVector &v0,
                    const ZFXVector &v1,
                    const ZFXVector &v2,
                    const ZFXVector &vcV,
                    float *pfMin, float *pfMax) {
    *pfMin = vcV * v0;
    *pfMax = *pfMin;

    float fDP = vcV * v1;
    if (fDP < *pfMin) *pfMin = fDP;
    else if (fDP > *pfMax) *pfMax = fDP;

    fDP = vcV * v2;
    if (fDP < *pfMin) *pfMin = fDP;
    else if (fDP > *pfMax) *pfMax = fDP;
} // TriProj

/*-----*/
```

```
// Kollisionsfunktion
bool ZFXObb::Intersects(const ZFXVector &v0,
                       const ZFXVector &v1,
                       const ZFXVector &v2) {
    float fMin0, fMax0, fMin1, fMax1;
    float fD_C;
    ZFXVector vcV, vcTriEdge[3], vcA[3];

    // zum Loopen
    vcA[0] = this->vcA0;
    vcA[1] = this->vcA1;
    vcA[2] = this->vcA2;

    // Richtung der Tri-Normalen
    vcTriEdge[0] = v1 - v0;
    vcTriEdge[1] = v2 - v0;

    vcV.Cross(vcTriEdge[0], vcTriEdge[1]);
```

```

fMin0 = vcV * v0;
fMax0 = fMin0;

this->ObbProj((*this), vcV, &fMin1, &fMax1);
if ( fMax1 < fMin0 || fMax0 < fMin1 )
    return true;

// Richtung der OBB-Ebenen
// =====
// Achse 1:
vcV = this->vcA0;
this->TriProj(v0, v1, v2, vcV, &fMin0, &fMax0);
fD_C = vcV * this->vcCenter;
fMin1 = fD_C - this->fA0;
fMax1 = fD_C + this->fA0;
if ( fMax1 < fMin0 || fMax0 < fMin1 )
    return true;

// Achse 2:
vcV = this->vcA1;
this->TriProj(v0, v1, v2, vcV, &fMin0, &fMax0);
fD_C = vcV * this->vcCenter;
fMin1 = fD_C - this->fA1;
fMax1 = fD_C + this->fA1;
if ( fMax1 < fMin0 || fMax0 < fMin1 )
    return true;

// Achse 3:
vcV = this->vcA2;
this->TriProj(v0, v1, v2, vcV, &fMin0, &fMax0);
fD_C = vcV * this->vcCenter;
fMin1 = fD_C - this->fA2;
fMax1 = fD_C + this->fA2;
if ( fMax1 < fMin0 || fMax0 < fMin1 )
    return true;

// Richtung der Tri-OBB-Kanten-Kreuzprodukte
vcTriEdge[2] = vcTriEdge[1] - vcTriEdge[0];
for (int j=0; j<3; j++) {
    for (int k=0; k<3; k++) {
        vcV.Cross(vcTriEdge[j], vcA[k]);

        this->TriProj(v0, v1, v2, vcV,
                      &fMin0, &fMax0);
        this->ObbProj((*this), vcV, &fMin1, &fMax1);

        if ( (fMax1 < fMin0) || (fMax0 < fMin1) )
            return true;
    }
}

```

```

    }
    }
    return true;
} // Intersects(Tri)
/*-----*/

```

Damit nähern wir uns mit großen Schritten dem Ende dieses Kapitels. Eine *kleine* Kollisionsfunktion steht für diesen Abschnitt noch aus, dann sind wir mit dem Stoff der Kollisionen im 3D-Raum erst mal durch.

Kollision zweier orientierter Boxen

Ganz so klein ist die Kollisionsfunktion allerdings nicht. Auch für den Test auf Kollision zweier orientierter Boxen verwendet man die Methode der Separationsachsen. Diese ist in dem original Paper über OBB-Bäume von Gottschalk et al. aus den SIGGRAPH Proceedings von 1996 beschrieben und auch in den eben bei der Kollisionsfunktion für Dreiecke genannten Quellen. Aus diesem Grund spare ich mir hier auch das Abdrucken des seitenlangen Quelltextes. Dieser befindet sich aber natürlich komplett auf der CD des Buches.

*Noch mehr
Quellen*

Ebenen einer AABB

Diese Methoden zum Abfragen der Ebenen einer AABB können wir kurz und schmerzlos gestalten. Wir benötigen unter Umständen (ganz sicher sogar) die Möglichkeit, die Ebenen einer AABB abzufragen. Dazu haben wir die folgende Methode, die die Ebenen zu einer AABB konstruiert. Das ist keine Magie, sondern einfachste Mathematik. Wir bestimmen einfach den Normalenvektor und geben einen Punkt an, der auf der Ebene liegt. Den Rest erledigt die Ebenen-Klasse für uns. Die folgende Methode erwartet die Startadresse eines Arrays von sechs Instanzen der ZFXPlane-Klasse, die sie dann mit den entsprechenden Ebenen betankt.

```

void ZFXAabb::GetPlanes(ZFXPlane *pPlanes) {
    ZFXVector vcN;

    if (!pPlanes) return;

    // rechts
    vcN.Set(1.0f, 0.0f, 0.0f);
    pPlanes[0].Set(vcN, vcMax);

    // links
    vcN.Set(-1.0f, 0.0f, 0.0f);
    pPlanes[1].Set(vcN, vcMin);

    // vorne

```



```

vcN.Set(0.0f, 0.0f, -1.0f);
pPlanes[2].Set(vcN, vcMin);

// hinten
vcN.Set(0.0f, 0.0f, 1.0f);
pPlanes[3].Set(vcN, vcMax);

// oben
vcN.Set(0.0f, 1.0f, 0.0f);
pPlanes[4].Set(vcN, vcMax);

// unten
vcN.Set(0.0f, -1.0f, 0.0f);
pPlanes[5].Set(vcN, vcMin);
} // Intersects(point)

```

Strahl in AABB

Eine weitere handliche Methode, die wir beispielsweise bei unserem Octree später brauchen werden, ist die Möglichkeit herauszufinden ob ein Linien-segment, also ein Strahl mit einer begrenzten Länge, vollkommen innerhalb einer AABB liegt. So kann man beispielsweise erkennen, ob ein Strahl aus einem Node eines Octrees herausreicht und man den entsprechenden Nachbarn testen muss. Aber darauf kommen wir viel später in diesem Buch noch einmal zurück.



```

bool ZFXAabb::Contains(const ZFXRay &Ray, float fL)
{
    ZFXVector vcEnd = Ray.m_vcOrig + (Ray.m_vcDir*fL);
    return ( Intersects(Ray.m_vcOrig) &&
            Intersects(vcEnd) );
} // Contains

```

Wir prüfen hier einfach, ob der Ursprung des Strahls und der Endpunkt in der AABB enthalten sind. Ist dies der Fall, so ist der Strahl vollkommen in der Bounding-Box enthalten. Die `ZFXAabb::Intersects`-Methode für einen Vektor zeige ich hier nicht, denn man muss dort einfach prüfen, ob der übergebene Punkt zwischen den beiden Extrempunkten der Box liegt. Der Code befindet sich natürlich auf der CD-ROM zu diesem Buch.

4.7 Rechnen mit Polygonen

*Erstens kommt es
anders ...*

... und zweitens als man denkt. So dachte ich mir doch: Eine Klasse für Polygone brauchst du für dieses Buch nicht. So viel werden wir nicht mit Polygonen mathematisch arbeiten. Doch es kommt ja immer anders, als man denkt. Ich habe mich nämlich soeben durch eine temporale, dimensionsparallele Subraumsalte bewegt, nachdem ich den Quelltext von *Pando-*

ras Legacy fertig gestellt habe. Ohne eine Klasse für Polygone wäre der Code nicht wirklich so schön gewesen, also hole ich die Arbeit an dieser Stelle nach. Oder ich ziehe sie vor – je nachdem aus welcher temporalen Normalität man das hier gerade betrachtet.

Unsere Polygon-Klasse werden wir entsprechend so designen, dass wir sie multifunktional einsetzen können. Es gibt viele Operationen, die man mit Polygonen durchführen können muss, wenn man beispielsweise einen Octree aus einer Geometrie von Polygonen konstruieren will oder einen BSP-Baum und solche Sachen. Diese Dinge sind normalerweise eine Hämorrhoid im Hintern, weil es dabei so viele Lowlevel-Aufgaben zu erledigen gibt, obwohl die Algorithmen an sich vor eleganter Einfachheit nur so strotzen. Und genau diese lästigen Lowlevel-Aufgaben schaffen wir uns durch das Anlegen einer Polygon-Klasse vom Hals. In späteren Kapiteln werden wir diese intensiv anwenden, und ihr werdet mehr als drei Kreuze machen, dass wir uns dieses Stück Arbeit bereits an dieser Stelle vom Hals geschafft haben. Auf geht's!

*Octree, Clipping,
BSP, ESP, FDH
und vieles mehr.*

Grundlegende Operationen

An dieser Stelle machen wir uns einfach noch mal deutlich, was genau ein Polygon ist und was wir von ihm erwarten. Ein Polygon in unserem Sinne ist eine Menge von Punkten, die eine ebene Fläche konvex begrenzen. Die Einschränkungen, dass alle Punkte in einer Ebene liegen und dass wir nur mit konvexen Polygonen arbeiten, macht uns die Sache etwas leichter. Aber es sei jeder angehalten, diese Klasse zu verallgemeinern und z.B. ebene und konvexe Polygone als eine Ableitung einer solchen generalisierten Polygon-Klasse zu gestalten.

Polygonale Basis

Unsere Hauptanforderung an eine solche Polygon-Klasse ist ihre Eignung für den Einsatz in der Umgebung der 3D-Computergrafik. Das läuft insbesondere darauf hinaus, dass ein solches Polygon sehr einfach in Standard-Algorithmen wie beispielsweise dem BSP-Baum oder einem Octree verwendbar ist. Dazu benötigen wir insbesondere Methoden, um ein Polygon an einer Bounding-Box oder an einer Ebene zu *clippen*, also zu beschneiden. Gut sind natürlich auch hier – wie bei unseren anderen Mathe-Klassen – Kollisionsmethoden, insbesondere mit Strahlen.

Anforderungen

Daneben gibt es noch die eine oder andere Utility-Methode, wie beispielsweise das Abfragen der Anzahl an Punkten des Polygons usw., die wir als direkte Accessor-Methoden implementieren. Aber betrachten wir erst mal die Definition der Klasse.



```

class __declspec(dllexport) ZFXPolygon {
    friend class ZFXPlane;

private:
    ZFXPlane      m_Plane;    // Ebene des Polygons
    int            m_NumP;     // Anzahl Punkte
    int            m_NumI;     // Anzahl Indices
    ZFXAabb        m_Aabb;     // Bounding-Box
    unsigned int   m_Flag;     // Beliebig nutzbar
    ZFXVector      *m_pPoints; // Punkte
    unsigned int   *m_pIndis;  // Indices

    void CalcBoundingBox(void);

public:
    ZFXPolygon(void);
    ~ZFXPolygon(void);

    void      Set(const ZFXVector*, int,
                  const unsigned int*, int);
    void      Clip(const ZFXPlane &Plane,
                  ZFXPolygon *pFront,
                  ZFXPolygon *pBack);
    void      Clip(const ZFXAabb &aabb);
    int       Cull(const ZFXAabb &aabb);
    void      CopyOf(const ZFXPolygon &Poly);

    void      SwapFaces(void);
    bool      Intersects(const ZFXRay&,
                        bool, float*);
    bool      Intersects(const ZFXRay&, bool
                        float fL, float *t);

    int       GetNumPoints(void){return m_NumP;}
    int       GetNumIndis(void){return m_NumI;}
    ZFXVector* GetPoints(void) {return m_pPoints;}
    unsigned int* GetIndices(void) {return m_pIndis;}
    ZFXPlane   GetPlane(void) {return m_Plane;}
    ZFXAabb     GetAabb(void) {return m_Aabb;}
    unsigned int GetFlag(void) {return m_Flag;}
    void        SetFlag(unsigned int n)
                {m_Flag = n;}

}; // class

```

**Was immer du
willst!**

Erwähnenswert, weil nicht selbsterklärend, ist hier das Attribut `m_Flag`. Es hat innerhalb der Klasse auch gar keine Bedeutung. Es dient dazu, dem Anwender unserer Klasse die Möglichkeit zu geben, jeder Instanz einen beliebigen Wert zuzuweisen und diesen wieder abzufragen. Was der Anwender dann damit macht, das ist uns vollkommen egal. Ganz praktisch ist dies z.B. beim Zerle-

gen einer Menge von Polygonen in einen BSP-Baum oder einen Octree, wenn man einzelne Polygone als bereits benutzt markieren will.

Der Konstruktor und der Destruktor der Klasse sind ebenso simpel, weil es hier nicht viel zu tun gibt, außer die Attribute mit Initialwerten zu belegen.

*Konstruktor und
Destruktor*

```
ZFXPolygon::ZFXPolygon(void) {
    m_pPoints = NULL;
    m_pIndis  = NULL;
    m_NumP    = 0;
    m_NumI    = 0;
    m_Flag    = 0;
    memset(&m_Aabb, 0, sizeof(ZFXAabb));
} // constructor
```

```
ZFXPolygon::~ZFXPolygon(void) {
    if (m_pPoints) {
        delete [] m_pPoints;
        m_pPoints = NULL;
    }
    if (m_pIndis) {
        delete [] m_pIndis;
        m_pIndis = NULL;
    }
} // destructor
```



Punkte für das Polygon festlegen

Wenn wir nun eine Instanz der Polygon-Klasse bilden, dann enthält diese noch nichts. Dazu müssen wir über die `ZFXPolygon::Set`-Methode noch die Daten angeben, aus welchen Punkten ein Polygon besteht. Das klingt zunächst recht einfach, doch wir müssen hierbei eines beachten: Wir möchten zu einem Polygon auch die Ebene haben, in der das Polygon liegt, da wir diese für diverse Algorithmen ständig benötigen.

Vorsicht Falle!

Nun wissen wir ja, dass wir die Ebene recht leicht bestimmen können, wenn wir den Normalenvektor der Ebene, also des Polygons, und einen Punkt in der Ebene, also einen Punkt des Polygons, kennen. Cool, dann berechnen wir einfach den Normalenvektor des Polygons, indem wir aus den ersten drei Punkten des Polygons zwei Kanten des Polygons erstellen und deren Vektoren kreuzen, um den Normalenvektor zu erhalten, richtig?

Eigentlich ja, wenn das kleine Wörtchen »wenn« nicht wäre. Bei Dreiecken ist das in der Tat so einfach. Doch wir arbeiten hier mit beinahe beliebigen Polygonen, und da kommt es schon vor, dass einige Punkte des Polygons auf derselben Kante liegen. Wir müssen also die Situation ausschließen, in der wir den Normalenvektor des Polygons aus zwei parallel laufenden Kan-

Parallele Kanten

ten zu bilden versuchen. Aus genau diesem Grund ist die folgende Methode ein wenig aufgebläht, denn wir durchlaufen alle Kanten des Polygons so lange, bis wir zwei nicht-parallele gefunden haben, mit denen wir dann die Ebene berechnen können.



```
void ZFXPolygon::Set(const ZFXVector *pPoints,
                    int nNumP,
                    const unsigned int *pIndis,
                    int nNumI) {
    ZFXVector vcEdge0, vcEdge1;
    bool bGotEm = false;

    if (m_pPoints) delete [] m_pPoints;
    if (m_pIndis) delete [] m_pIndis;

    m_pPoints = new ZFXVector[nNumP];
    m_pIndis = new unsigned int[nNumI];

    m_NumP = nNumP;    m_NumI = nNumI;

    memcpy(m_pPoints,pPoints,sizeof(ZFXVector)*nNumP);
    memcpy(m_pIndis,pIndis,sizeof(unsigned int)*nNumI);

    vcEdge0 = m_pPoints[m_pIndis[1]] -
              m_pPoints[m_pIndis[0]];

    // berechne die Ebene
    for (int i=2; bGotEm==false; i++) {
        if ((i+1) > m_NumI) break;

        vcEdge1 = m_pPoints[m_pIndis[i]] -
                  m_pPoints[m_pIndis[0]];

        vcEdge0.Normalize();
        vcEdge1.Normalize();

        // Kanten dürfen nicht parallel sein
        if (vcEdge0.AngleWith(vcEdge1) != 0.0)
            bGotEm = true;
    } // for

    m_Plane.m_vcN.Cross(vcEdge0, vcEdge1);
    m_Plane.m_vcN.Normalize();
    m_Plane.m_fD = -(m_Plane.m_vcN * m_pPoints[0]);
    m_Plane.m_vcPoint = m_pPoints[0];

    CalcBoundingBox();
} // Set
```

Eine Sache ist hier noch zu erwähnen: Natürlich besteht ein Polygon nur aus einer Menge von Punkten. Doch wir speichern in einem Polygon auch eine Liste von Indices, die die Triangulation des Polygons repräsentiert. In der Computergrafik ist es nun mal oftmals notwendig, mit solchen Indices zu arbeiten, und daher fließt das auch in unsere Polygon-Klasse mit ein.

*Indizierte
Polygone*

Und wo wir gerade schon mal dabei sind, zeige ich auch gleich noch den Code, wie wir eine achsen-ausgerichtete-Bounding-Box für das Polygon erstellen.

AABB berechnen

```
void ZFXPolygon::CalcBoundingBox(void) {
    ZFXVector vcMax, vcMin;
    vcMax = vcMin = m_pPoints[0];

    for (int i=0; i<m_NumP; i++) {
        if ( m_pPoints[i].x > vcMax.x )
            vcMax.x = m_pPoints[i].x;
        else if ( m_pPoints[i].x < vcMin.x )
            vcMin.x = m_pPoints[i].x;

        if ( m_pPoints[i].y > vcMax.y )
            vcMax.y = m_pPoints[i].y;
        else if ( m_pPoints[i].y < vcMin.y )
            vcMin.y = m_pPoints[i].y;

        if ( m_pPoints[i].z > vcMax.z )
            vcMax.z = m_pPoints[i].z;
        else if ( m_pPoints[i].z < vcMin.z )
            vcMin.z = m_pPoints[i].z;
    } // for
    m_Aabb.vcMax    = vcMax;
    m_Aabb.vcMin    = vcMin;
    m_Aabb.vcCenter = (vcMax + vcMin) / 2.0f;
} // CalcBoundingBox
```



Nein, es ist noch nicht Karneval, und wir sind auch nicht in Rio. Trotzdem ist es manchmal für ein Polygon notwendig, sein Gesicht zu vertauschen. Eine Münze, sprich: ein Polygon, hat ja immer zwei Seiten. Eine davon bezeichnet man als die Vorderseite (engl. Frontface) und eine als die Rückseite (engl. Backface). Erstere ist diejenige, auf deren Seite der Normalenvektor steht. In der Computergrafik rendert man für gewöhnlich nur die Frontfaces der Geometrie, und daher macht es Sinn, für die Kollisionsabfrage auch nur die Frontfaces zu verwenden.

*Gesichter
vertauschen*



An dieser Stelle möchte ich noch einmal explizit darauf hinweisen, dass wir die Klasse für Polygone nachher nur für Berechnungen im 3D-Raum benötigen. Instanzen dieser Klasse werden ausdrücklich nicht zum Rendern verwendet. Daher genügt es auch, die Positionen der Punkte des Polygons zu kennen. Wir brauchen hier nicht etwa auch Vertex-Normalen, Texturkoordinaten usw., die wir für die Triangle-Listen angeben müssen, die wir tatsächlich zum Rendern verwenden.

Um die entsprechende Ausrichtung eines Polygons, oder genauer gesagt: seines Normalenvektors, zu verändern, definieren wir die folgende Methode:



```
void ZFXPolygon::SwapFaces(void) {
    unsigned int *pIndis = new unsigned int[m_NumI];

    // Indersortierung ändern
    for (int i=0; i<m_NumI; i++)
        pIndis[m_NumI-i-1] = m_pIndis[i];

    // Richtung des Normalenvektors umdrehen
    m_Plane.m_vcN *= -1.0f;
    m_Plane.m_fD *= -1.0f;

    delete [] m_pIndis;
    m_pIndis = pIndis;
} // SwapFaces
```

*Distanz der Ebene
nicht vergessen*

Zum einen vertauschen wir die Reihenfolge der Indices, denn nur, wenn wir diese in umgekehrter Reihenfolge definieren, stellen sie die Veränderung von Frontface und Backface korrekt dar. Wenn man nun wieder automatisiert den Normalenvektor berechnen würde, dann würde dieser nun korrekt umgedreht berechnet. Zum anderen brauchen wir das aber gar nicht zu tun, denn wir können den Normalenvektor einfach so verwenden, wie er ist, und ihn einfach umdrehen. Wir dürfen nur nicht vergessen, auch das Vorzeichen des Distanzwertes der Ebenengleichung zu negieren.

Clippen eines Polygons

*Beschneidung auf
Polygonesisch*

Das Clipping eines Polygons ist eine Operation, die bei dem Scene-Management einer Geometrie von Polygonen recht häufig auftreten wird. Wir werden hier die zwei typischsten Fälle implementieren, nämlich das Clippen eines Polygons mit einer AABB und mit einer Ebene. Ersteres werden wir beispielsweise für einen Octree gut verwenden können, und Letzteres braucht man an sehr vielen Stellen. Als Beispiel sei hier nur der BSP-Algorithmus genannt, wo sämtliche Geometrie an so genannten Splitter-Ebenen geclippt wird.

Clipping an einer AABB

Das Clipping eines Polygons an einer achsen-orientierten Bounding-Box ist vergleichsweise trivial, wenn man seine Hausaufgaben gemacht hat. Streng genommen, kann man eine AABB auch als ein Volumen im Raum auffassen, das durch sechs Ebenen begrenzt wird. Das ist auch schon der ganze Trick. Wir holen uns die sechs Ebenen der AABB, dazu haben wir ja schon eine praktische Methode, die das tut, und clippen das Polygon nach und nach an allen sechs Ebenen. Genau das tut die folgende Methode:

Mit Köpfchen

```
void ZFXPolygon::Clip(const ZFXAabb &aabb) {
    ZFXPolygon BackPoly, ClippedPoly;
    ZFXPlane Planes[6];
    bool bClipped=false;

    // Tom Hanks casts away const
    ZFXAabb *pAabb = ((ZFXAabb*)&aabb);

    // Ebenen der AABB holen, die Normalen-
    // vektoren zeigen nach außen
    pAabb->GetPlanes(Planes);

    // kopiere das Polygon
    ClippedPoly.CopyOf( *this );

    // und nun Clipping
    for (int i=0; i<6; i++) {
        if (Planes[i].Classify(ClippedPoly) == ZFXCLIPPED)
        {
            ClippedPoly.Clip(Planes[i], NULL, &BackPoly);
            ClippedPoly.CopyOf(BackPoly);
            bClipped = true;
        }
    }

    if (bClipped) CopyOf(ClippedPoly);
} // Clip
```



Ich gebe zu, das ist einfacher gesagt als gecodet, denn die Hauptarbeit des Clippens an einer Ebene fehlt uns ja noch. Aber dazu kommen wir gleich. Es sei nur so viel gesagt, dass die Clipping-Methode für Polygone an Ebenen natürlich zwei Polygone erzeugt – vorausgesetzt, das Polygon schneidet die Ebene. Wir haben dann ein Polygon auf der Frontface-Seite der Ebene und ein Polygon auf der Backface-Seite der Ebene. Es ist also wichtig zu wissen, in welche Richtung die Normalenvektoren zeigen und was wir eigentlich wollen.

*Richtung der
Normalen
beachten!*

Die Methode `ZFXAabb::GetPlanes` ist so implementiert, dass die Normalenvektoren der sechs Ebenen nach außen zeigen. Wenn ein Polygon eine AABB

schneidet, dann müssen wir es entsprechend so an den Ebenen der AABB clippen, dass wir den Backface-Teil des Polygons behalten und den Rest, der dann außerhalb der AABB liegt, verwerfen.

*Klassifizierung
von Polygonen mit
Ebenen*

Die oben verwendete Methode `ZFXPlane::Classify` mit einem Polygon als Parameter habe ich oben noch nicht gezeigt, da wir die Polygon-Klasse noch nicht hatten. Sie ist aber natürlich trivial umzusetzen, denn man muss hier lediglich alle Punkte des Polygons durchlaufen und an der Ebene klassifizieren. Liegen alle Punkte auf derselben Seite der Ebene, so erhält man die entsprechende Seite als Rückgabewert. Liegt mindestens einer der Punkte auf der anderen Seite der Ebene, dann erhält man `ZFXCLIPPED` als Rückgabewert und weiß, dass das Polygon die Ebene schneidet.

Clipping an einer Ebene

*Augen zu, Hirn an,
und durch!*

Na schön. Wir haben uns bis hierhin vorgekämpft, doch nun stehen wir unserem Angstgegner gegenüber: dem allseits gefürchteten Polygon-Ebenen-Clipper! But we're still Marines and got a job to do. Eigentlich ist es auch gar nicht so schwer, ein Polygon an einer Ebene zu splitten oder zu clippen. Man muss sich nur gut überlegen, was man eigentlich tun muss. Im Endeffekt haben wir folgende Situation: Ein Polygon wird durch eine Ebene geschnitten. Wir haben also einen Teil des Polygons auf der Frontseite der Ebene und einen Teil des Polygons auf der Backseite der Ebene. Bei Clipping an der Ebene verwerfen wir also das Originalpolygon und erstellen zwei neue – ein Frontside-Polygon und ein Backside-Polygon.

Algorithmus

Nun könnten wir ja einfach alle Punkte des Polygons durchlaufen, sie entsprechend klassifizieren und in zwei entsprechende Listen einordnen. Das ist auch schon nah dran an dem, was zu tun ist. Doch leider wird ein Polygon selten so liegen, dass der Schnitt der Ebene genau durch Punkte des Polygons läuft. Wir müssen also an den Schnittpunkten neue Punkte in das Polygon einfügen, damit wir es sauber an dieser Stelle trennen können. Das ist eigentlich ein recht smarter Prozess und auch nicht so schwer zu verstehen. Die Methode ist nur etwas lang. Daher machen wir uns einen kurzen Ablaufplan, was wirklich zu tun ist:

1. Erstelle zwei Listen für Punkte der neuen Polygone (Frontpolygon und Backpolygon).
2. Klassifiziere den ersten Punkt des ursprünglichen Polygons, und sortiere ihn in die entsprechende Liste ein (Frontliste bzw. Backliste). Liegt der Punkt genau in der Ebene, dann sortiere ihn in beide Listen ein.
3. Nimm den nächsten Punkt des Polygons, und klassifiziere ihn. Liegt er in der Ebene, füge ihn in beide Listen ein. Ansonsten bilde mit dem zuvor betrachteten Punkt eine Kante des Polygons, und prüfe, ob diese Kante die Ebene schneidet. Wenn es einen Schnittpunkt gibt, dann füge

diesen als neuen Punkt in beide Listen ein. Nun sortiere den aktuellen Punkt in die Listen ein, falls er nicht in der Ebene lag.

4. Hat das ursprüngliche Polygon noch weitere Punkte, dann wiederhole Schritt 3, sonst gehe zu Schritt 5.
5. Enthalten die beiden Listen mehr als zwei Punkte (sind also gültige Polygone), dann konstruiere eine neue Indexliste für die Triangulation. Dazu wählt man den ersten Punkt des Polygons und erstellt mit den jeweils zwei folgenden Punkten ein Dreieck für die Triangulation ([0,1,2], [0,2,3], [0,3,4], ...).
6. Erzeuge aus den beiden Punktlisten und den beiden Indexlisten zwei neue Polygone. Prüfe dabei mittels der Normalenvektoren der Ebenen, ob die neuen Polygone in dieselbe Richtung schauen wie das ursprüngliche Polygon. Die komplette Neuerstellung der Indexliste kann die Ausrichtung umgedreht haben. Ist dies der Fall, dann vertausche die Faces des Polygons.

Das liest sich doch wirklich sehr einfach, oder? Und die Implementierung ist ganz genauso trivial, wie es hier aussieht. Man hat nur unheimlich viel in einer Methode zu tun, was man nicht unbedingt in Hilfsroutinen auslagern kann, ohne ein Stückelwerk von verschiedenen Listen in verschiedenen Methoden zu erzeugen. Also lasst euch nicht von der Länge der Methode abschrecken. Nichts wird so heiß ausgeführt, wie es programmiert wird.

Heißer Quellcode

```
void ZFXPolygon::Clip(const ZFXPlane &Plane,
                    ZFXPolygon *pFront,
                    ZFXPolygon *pBack) {
    if (!pFront && !pBack) return;

    ZFXVector vcHit, vcA, vcB;
    ZFXRay    Ray;

    // cast away const
    ZFXPlane *pPlane = ((ZFXPlane*)&Plane);

    unsigned int nNumFront=0, // Anzahl Punkte Frontside
                nNumBack=0,   // Anzahl Punkte Backside
                nLoop=0, nCurrent=0;

    ZFXVector *pvcFront = new ZFXVector[m_NumP*3];
    ZFXVector *pvcBack  = new ZFXVector[m_NumP*3];

    // klassifiziere den ersten Punkt
    switch (pPlane->Classify(m_pPoints[0])) {
        case ZFXFRONT:
            pvcFront[nNumFront++] = m_pPoints[0];
            break;
```



```

case ZFXBACK:
    pvcBack[nNumBack++] = m_pPoints[0];
    break;
case ZFXPLANAR:
    pvcBack[nNumBack++] = m_pPoints[0];
    pvcFront[nNumFront++] = m_pPoints[0];
    break;
default: return;
}

// laufe durch alle Punkte des Polygons
for (nLoop=1; nLoop < (m_NumP+1); nLoop++) {

    if (nLoop == m_NumP) nCurrent = 0;
    else nCurrent = nLoop;

    // nimm zwei benachbarte Punkte des Polygons
    vcA = m_pPoints[nLoop-1];
    vcB = m_pPoints[nCurrent];

    // klassifiziere sie mit der Ebene
    int nClass = pPlane->Classify(vcB);
    int nClassA = pPlane->Classify(vcA);

    // falls planar, füge sie zu beiden Seiten hinzu
    if (nClass == ZFXPLANAR) {
        pvcBack[nNumBack++] = m_pPoints[nCurrent];
        pvcFront[nNumFront++] = m_pPoints[nCurrent];
    }

    // anderenfalls prüfe, ob eine Kante die Ebene
    // schneidet
    else {
        Ray.m_vcOrig = vcA;
        Ray.m_vcDir = vcB - vcA;

        float fLength = Ray.m_vcDir.GetLength();

        if (fLength != 0.0f) Ray.m_vcDir /= fLength;

        if ( Ray.Intersects(Plane, false, fLength, 0,
                            &vcHit)
            && (nClassA != ZFXPLANAR)) {

            // dann füge den Schnittpunkt als neuen
            // Punkt in beide Listen ein
            pvcBack[nNumBack++] = vcHit;
            pvcFront[nNumFront++] = vcHit;
        }

        // sortieren den aktuellen Punkt ein
        if (nCurrent == 0) continue;
    }
}

```



```

        if (nClass == ZFXFRONT) {
            pvcFront[nNumFront++] =
                m_pPoints[nCurrent];
        }
        else if (nClass == ZFXBACK) {
            pvcBack[nNumBack++] =
                m_pPoints[nCurrent];
        }
    }
} // for [NumP]

// Jetzt haben wir die Vertices für beide
// Polygone und kümmern uns um die Indices.

unsigned int I0, I1, I2;
unsigned int *pnFront = NULL;
unsigned int *pnBack = NULL;

if (nNumFront > 2) {
    pnFront = new unsigned int[(nNumFront-2)*3];

    for (nLoop=0; nLoop < (nNumFront-2); nLoop++) {
        if (nLoop==0) { I0=0; I1=1; I2=2; }
        else { I1=I2; I2++; }

        pnFront[(nLoop*3) ] = I0;
        pnFront[(nLoop*3) +1] = I1;
        pnFront[(nLoop*3) +2] = I2;
    }
}

if (nNumBack > 2) {
    pnBack = new unsigned int[(nNumBack-2)*3];

    for (nLoop=0; nLoop < (nNumBack-2); nLoop++) {
        if (nLoop==0) { I0=0; I1=1; I2=2; }
        else { I1=I2; I2++; }

        pnBack[(nLoop*3) ] = I0;
        pnBack[(nLoop*3) +1] = I1;
        pnBack[(nLoop*3) +2] = I2;
    }
}

// erzeuge die neuen Polygone aus den Daten
if (pFront && pnFront) {
    pFront->Set(pvcFront, nNumFront, pnFront,
        (nNumFront-2)*3);
}

```

```

// achte darauf, dass wir dieselbe Ausrichtung
// beibehalten wie das Originalpolygon
if (pFront->GetPlane().m_vcN * m_Plane.m_vcN
    < 0.0f)
    pFront->SwapFaces();
}

if (pBack && pnBack) {
    pBack->Set(pvcBack, nNumBack, pnBack,
              (nNumBack-2)*3);

    if (pBack->GetPlane().m_vcN * m_Plane.m_vcN
        < 0.0f)
        pBack->SwapFaces();
}

if (pvcFront) { delete [] pvcFront; }
if (pvcBack) { delete [] pvcBack; }
if (pnFront) { delete [] pnFront; }
if (pnBack) { delete [] pnBack; }
} // Clip

```

Organisatorisches

Wie man sehen kann, lässt diese Methode das eigentliche Objekt, über das wir die Methode aufrufen, vollkommen intakt. Die beiden neu generierten Polygone, die auf der Frontseite und der Backseite der betrachteten Ebene liegen, sind zwei neue Instanzen der `ZFXPolygon`-Klasse. Was diese Methode auch explizit nicht tut, ist die Überprüfung, ob die Ebene das Polygon überhaupt zerschneidet. Der Aufrufer muss dies im Vorfeld abklären und ein Polygon entsprechend erst dann clippen, wenn er die Notwendigkeit dazu eruiert hat. Das könnte wie folgt aussehen:

```

ZFXPolygon FrontPoly, BackPoly;

if ( Plane.Classify( Poly ) == ZFXCLIPPED )
    Poly.Clip(Plane, &FrontPoly, &BackPoly);

```

Hier würden wir das ursprüngliche Polygon verwerfen und mit den beiden neuen Polygonen weiterarbeiten. Wenn wir ein Polygon jedoch an einer AABB clippen wollen, dann ist das noch einfacher. Denn dort haben wir unseren Clipper ja bereits so geschrieben, dass die Methode das Objekt selbst nimmt, es gegen die Ebenen der AABB clippt und dann jeweils den in der ABB liegenden Teil des Polygons auf sich selbst kopiert. Wenn wir ein Polygon so beschneiden wollen, dass wir nur noch den Teil des Polygons haben, der innerhalb einer AABB liegt, dann sieht das wie folgt aus:

```

Plane.Clip( aabb );

```

Das Ergebnis dieses Aufrufes ist, dass die Instanz `PPlane` der Klasse `ZFXPlane` so beschnitten ist (falls das notwendig gewesen ist), dass das entsprechende Polygon vollkommen innerhalb der Instanz `aabb` der Klasse `ZFXAabb` liegt. Hier rieche ich schon förmlich den Octree.

Insbesondere hier wird es deutlich, warum wir uns in unserer Klasse auf konvexe Polygone beschränkt haben. Die Generierung einer komplett neuen Indexliste nach dem oben gezeigten primitiven Algorithmus lässt sich nur für konvexe Polygone verwenden. Bei konkaven Polygonen wäre dies um einiges komplizierter.



**!!
STOP**

Culling mit Bounding-Boxen

Eine weitere Hilfsmethode, die wir insbesondere für Scene-Management, Algorithmen gut verwenden können, ist die Überprüfung, ob ein Polygon in einer Bounding-Box eingeschlossen ist. Wir nehmen dazu das Innere der Bounding-Box als sichtbaren Bereich an und möchten wissen, ob das Polygon total sichtbar ist (also vollkommen in der AABB liegt), ob es nur teilweise sichtbar ist (also die AABB schneidet) oder ob es total unsichtbar ist (also vollkommen außerhalb der AABB liegt).

Den ersten Fall können wir sehr leicht untersuchen. Es müssen einfach alle Punkte des Polygons innerhalb der AABB liegen. Der letzte Fall ist jedoch nicht so einfach, wie es zunächst scheint. Selbst wenn alle Punkte außerhalb der Box liegen, so kann das Polygon einfach größer sein als die Box und komplett durch diese durchgehen. Wir definieren einfach den letzten Fall so, dass keiner der beiden ersten Fälle eingetreten ist. Der mittlere Fall ist schon ein wenig interessanter. Wir werden uns auch hier wieder die Ebenen der AABB vornehmen und die Kanten des Polygons auf einen Schnittpunkt mit den Ebenen prüfen. Finden wir einen solchen Schnittpunkt, so geben wir einfach als Ergebnis zurück, dass das Polygon die Box schneidet.

Methoden

Das ist mathematisch aber nicht wirklich korrekt, sondern nur eine grobe Abschätzung des Ergebnisses. Ein Polygon kann ja durchaus eine unendlich ausgedehnte Ebene der AABB schneiden, ohne selbst die AABB zu schneiden. Etwas besser wäre die Annäherung des Ergebnisses, wenn wir mindestens zwei Schnittpunkte des Polygons mit zwei verschiedenen Ebenen der AABB erwarten. Doch auch diese Voraussetzung können Polygone erfüllen, die die AABB nicht schneiden, aber sehr nah an ihnen liegen. Genau wird der Test dann, wenn man einen gefundenen Schnittpunkt daraufhin untersucht, ob er auch in der Seitenfläche der zugehörigen Ebene liegt, und nicht nur in der Ebene. Bei Rechtecken ist dies vergleichsweise einfach, denn da die Box achsen-orientiert ist, lässt sich das Problem zweidimensional lösen.

Achtung!

Die Implementierung dieser Genauigkeit überlasse ich euch als kleine Übung. Wir kommen im Folgenden mit der groben Abschätzung aus, wenn wir um die Ungenauigkeit wissen und sie entsprechend beachten.



```
int ZFXPolygon::Cull(const ZFXAabb &aabb) {
    ZFXPlane Planes[6];
    int    nClass=0;
    int    nInside=0, nCurrent=0;
    bool    bFirst=true;
    ZFXRay    Ray;

    ZFXAabb *pAabb = ((ZFXAabb*)&aabb);

    // Ebenen der AABB, Normalen zeigen nach außen
    pAabb->GetPlanes(Planes);

    // Überlappen sich die AABB überhaupt?
    if ( !m_Aabb.Intersects(aabb) )
        return ZFXCULLED; // no way

    // für alle Ebenen
    for (int p=0; p<6; p++) {

        // prüfe einmalig, ob alle Punkte in AABB liegen
        if (bFirst) {
            for (int i=0; i<m_NumP; i++) {
                if ( pAabb->Intersects(m_pPoints[i]) )
                    nInside++;
            }
            bFirst = false;

            // ja => Polygon total innerhalb der AABB
            if (nInside == m_NumP) return ZFXVISIBLE;
        }

        // teste auf Schnittpunkt der aktuellen Ebene mit
        // den Kanten des Polygons
        for (int nLoop=1; nLoop < (m_NumP+1); nLoop++) {
            if (nLoop == m_NumP) nCurrent = 0;
            else nCurrent = nLoop;

            // Kante aus zwei benachbarten Punkten
            Ray.m_vcOrig = m_pPoints[nLoop-1];
            Ray.m_vcDir  = m_pPoints[nCurrent]
                - m_pPoints[nLoop-1];

            float fLength = Ray.m_vcDir.GetLength();
            if (fLength != 0.0f) Ray.m_vcDir /= fLength;
```

```

        // falls Schnittpunkt, dann scheidet die Ebene
        // und damit die AABB (u.U.) das Polygon
        if (Ray.Intersects(Planes[p], false, fLength,
            0, NULL))
            return ZFXCLIPPED;
    }
}

// Polygon nicht in der AABB und kein Schnittpunkt
return ZFXCULLED;
} // Cull

```

Wenn man einen Schnittpunkt gefunden hat, kann man natürlich auch die `ZFXAabb::Intersects`-Methode für einen Punkt bemühen, die überprüft, ob der angegebene Punkt innerhalb der AABB liegt. Die Voraussetzung dafür ist, dass diese Methode auch Punkte genau auf einer Seitenfläche der AABB erkennt. Man muss also den Fall berücksichtigen, dass die Koordinaten gleich den mit den Extrempunkt-Koordinaten sind und dort auch für `float`-Rechen-Ungenauigkeiten einen gewissen Spielraum lassen.

*Kleiner Tipp zur
Übungsaufgabe*

Kollision mit Strahlen

Das mit Abstand Wichtigste an unserer 3D-Mathe-Bibliothek sind die Kollisionsabfragen. Diese werden wir im Game-Code ständig brauchen. Natürlich braucht dann auch unsere `ZFXPolygon`-Klasse solche Kollisionsmethoden. Das eben gezeigte Culling mit einer Bounding-Box ist ja schon so eine Art Kollisionsabfrage, und wer mag, kann diese Klasse auch um eine Methode `ZFXPolygon::Intersects(ZFXAabb)` erweitern, die dann einfach intern die `ZFXPolygon::Cull`-Methode aufruft und nur im Fall `ZFXCULLED` den Wert `false` zurückgibt. Hier ist aber die Frage, ob wirklich jeder Aufrufer den Fall, dass das Polygon komplett in der AABB liegt, als Kollision erkannt haben möchte.

*Immer wieder
ecken wir an*

Wir implementieren hier jedenfalls erst einmal nur die wichtigsten Fälle der `ZFXPolygon::Intersects`-Methode für einen Strahl und ein Liniensegment, wobei wir das Liniensegment als Strahl mit einer bestimmten Länge auffassen. Je öfter mir aber ein Liniensegment unterkommt, desto eher bin ich geneigt, in der nächsten Iteration der `zfx3d.lib` auch eine Klasse für Liniensegmente einzuführen.

In der folgenden Methode sehen wir auch wieder ein, warum wir die Triangulation für das Polygon mitgespeichert haben. Wir haben ja schon eine schöne Methode der `ZFXRay`-Klasse, um auf eine Kollision zwischen einem Dreieck und einem Strahl zu prüfen. Und genau diese verwenden wir hier einfach für alle Dreiecke, aus denen das Polygon besteht.

*Strahl gegen
Dreiecke*



Wie bereits weiter oben gesagt wurde, müssen wir an einigen Stellen einen Pointer auf einen als `const` übergebenen Parameter der Methode richten. Das ist notwendig, um das `const` wegzucasten, weil wir sonst keine Memberfunktionen des Parameters aufrufen könnten. Da würde sich der Compiler beschweren. Damit der Aufrufer unserer Methode aber gewiss ist, dass seinem übergebenen Objekt nichts passiert, ist es dennoch schön, das `const` im Funktionskopf stehen zu haben.

Kollision mit Backface?

Der Methode kann man auch gleich als Parameter angeben, ob man auch wünscht, dass auf eine Kollision mit der Rückseite des Polygons getestet wird. Dies wird oftmals nicht gewünscht werden. Ist das aber dennoch der Fall, drehen wir einfach die Reihenfolge der Indices eines Dreiecks beim Test um, so dass die Backseite zur Frontseite wird.



```
bool ZFXPolygon::Intersects(const ZFXRay &Ray,
                           bool bCull, float *t) {
    ZFXRay *pRay = (ZFXRay*)&Ray;

    for (int i=0; i<m_NumI; i+=3) {
        if (pRay->Intersects(m_pPoints[m_pIndis[i]],
                           m_pPoints[m_pIndis[i+1]],
                           m_pPoints[m_pIndis[i+2]],
                           false, t))
        {
            return true;
        }
        if (!bCull) {
            if (pRay->Intersects(m_pPoints[m_pIndis[i+2]],
                               m_pPoints[m_pIndis[i+1]],
                               m_pPoints[m_pIndis[i]],
                               false, t))
            {
                return true;
            }
        }
    }
    return false;
} // Intersects
```

Liniensegmente

Der Test für das Liniensegment ist ganz analog implementiert. Hier wird jedoch noch die Länge des Segmentes zusätzlich übergeben. Wurde eine Kollision mit dem Strahl gefunden, so testen wir die Distanz zum Kollisionspunkt noch gegen die Länge des Segmentes. Der Code dieser Methode befindet sich natürlich auf der CD-ROM.

4.8 Rechnen mit Quaternionen

Bevor wir dieses Kapitel nun endgültig schließen können, fehlt uns aber noch die Vorstellung eines grundlegenden Konzeptes von 3D-Mathematik. Oder vielleicht sollte ich besser 4D-Mathematik sagen. Ich spreche hier natürlich von den so genannten *Quaternionen*, wie man bereits an der Überschrift zu diesem Abschnitt erkennen kann. Wer in der Schule ab und an im Mathematik-Unterricht die Augen und Ohren offen hatte, der wird wenigstens vom Namen her schon mit Vektoren und Matrizen vertraut sein. Quaternionen sind jedoch eine Sache, die man für gewöhnlich nicht in der Schule anspricht.

Wer sich aber mit 3D-Computergrafik beschäftigt, wird den Begriff Quaternion wenigstens schon einmal gehört haben. Damit wir eine grundsätzliche Idee davon haben, wozu wir ein Quaternion überhaupt brauchen, versuche ich es hier einmal einfach auszudrücken: Wenn man die Rotation eines Objekts im 3D-Raum beschreiben will, kann man dies auf (mindestens) drei verschiedene Arten tun. Zum einen kann man die oben gezeigten *Rotationsmatrizen* verwenden. Dabei wird die beliebige Rotation eines Punktes im 3D-Raum in einer Matrix gespeichert, mit der man dann Punkte transformieren kann. Die zweite Möglichkeit sind die so genannten *Euler-Winkel*. Hier speichert man den Rotationswinkel und dazu die Rotationsachse zu einem Objekt. Beispielsweise könnte man die drei Rotationswinkel um die lokale X-, Y- und Z-Achse eines Objekts speichern und es dadurch transformieren.

Matrizen, Euler-Winkel und Quaternionen

Die dritte Möglichkeit sind Quaternionen. In einem Quaternion kann man durch wesentlich weniger Daten (d.h. mit weniger Speicheraufwand) dieselben Informationen über die Rotation und die Achse eines Objekts speichern. Auch wenn die Standard-APIs (OpenGL und Direct3D) mit Rotationsmatrizen arbeiten, sind Quaternionen an vielen Stellen in einer 3D-Engine sehr hilfreich. Doch dazu gleich mehr.

Einführung in den 4D-Raum

Bevor wir eine Klasse für Quaternionen implementieren, schauen wir uns in aller Kürze die Grundlagen der Quaternion-Mathematik an. Ein Quaternion ist im Grunde genommen eine vierdimensionale Zahl, nicht zu verwechseln beispielsweise mit einem Vektor aus eindimensionalen Zahlen im vierdimensionalen Raum, wie wir sie bisher gesehen haben. Im Mathematik-Leistungskurs an der Schule habt ihr es eventuell schon mit den zweidimensionalen Zahlen (auch komplexe Zahlen genannt) zu tun bekommen. Im Jahre 1843 stellte Sir William Rowan Hamilton die Grundlagen der Quaternionen als Erweiterung für die komplexen Zahlen auf.



Nach den eindimensionalen Zahlen (den Skalaren) und den zweidimensionalen Zahlen (den komplexen Zahlen) gibt es eine dreidimensionale Lücke. Erst bei den vierdimensionalen Zahlen (eben jenen Quaternionen) konnte man bisher vernünftige mathematische Gesetze formulieren.

Komplexe Zahlen

Komplexe Zahlen werden durch eine »normale« Zahl beschrieben, zu der ein imaginärer Teil addiert wird. Diesen kürzt man in der Regel durch den Buchstaben i ab, woraus sich folgende Formel ergibt:

$$k = (a + bi)$$

Die beiden Variablen a und b sind dabei normale Skalare, und i ist quasi eine Art Einheit. Allerdings hat i auch einen definierten Wert. Es gilt:

$$(i * i) = -1 \quad \Leftrightarrow \quad i = \text{Wurzel}(-1)$$

An dieser Stelle verweise ich auf ein gutes Buch zur Analysis, aber ihr könnt hier bereits sehen, welche Implikationen die komplexen Zahlen haben. Man kann nun auch Wurzeln aus negativen Zahlen ziehen und das Ergebnis als komplexe Zahl darstellen.

Quaternionen

Die Quaternionen erweitern die komplexen Zahlen um gleich zwei weitere imaginäre Werte, die üblicherweise j und k genannt werden:

$$\begin{aligned} (j * j) &= -1 & \Leftrightarrow & \quad j = \text{Wurzel}(-1) \\ (k * k) &= -1 & \Leftrightarrow & \quad k = \text{Wurzel}(-1) \end{aligned}$$

Entsprechend kann man ein Quaternion als vierdimensionale Zahl aus einem Realteil und drei imaginären Teilen schreiben:

$$q = (a + bi + cj + dk)$$

Diese Schreibweise kann man noch ein wenig umformulieren, damit wir sehen können wohin das Ganze führt:

$$q = [w, v] \quad \text{mit } v = (x, y, z)$$

Ein Quaternion kann also durch einen skalaren Wert, hier w genannt, und einen Vektor, hier v genannt, dargestellt werden. In dieser Betrachtungsweise stellt w quasi den Winkel dar, um den rotiert worden ist, und der Vektor v repräsentiert die Achse, um die rotiert wurde. In so einem Quaternion kann man jede beliebige Rotation darstellen, die man auch über Euler-Winkel und eine Rotationsmatrix ausdrücken könnte. Die notwendige Mathematik, mit der man entsprechende Berechnungen zur Erzeugung eines Quaternionen durchführen kann, ist sogar noch einfacher als beispielsweise bei der Erstellung einer komplexen Rotationsmatrix um drei Achsen mit drei verschiedenen Winkeln.

Die Herleitung der verschiedenen Quaternion-Operationen spare ich mir an dieser Stelle, da es allein im Internet zahlreiche Quellen dazu gibt. Ich werde hier auch nur die notwendigsten Operationen in unsere Quaternion-Klasse einbauen, die wir später benötigen werden.

Was bieten Quaternions, das die anderen beiden oben genannten Methoden nicht bieten? Wie eben erwähnt, lassen sich Quaternions zwar schneller berechnen, aber die Standard-APIs arbeiten in der Regel mit Rotationsmatrizen. Wenn wir in unserer Engine mit Quaternions arbeiten würden, dann müssten wir jedes Quaternion entsprechend in eine Matrix umwandeln. Allerdings gibt es eine Stelle in unserer Engine, an der auch wir nicht auf den Komfort von Quaternions verzichten wollen, selbst wenn wir dadurch Konvertierungsaufwand erzeugen. Diese Stelle ist die Implementierung einer freien Kamera, mit der sich der Spieler durch die virtuelle 3D-Welt bewegen kann. In einem späteren Kapitel kommen wir zu der tatsächlichen Implementierung der Kamera (in der Basisklasse `ZFXMovementController`), und dort werde ich die Notwendigkeit zur Verwendung von Quaternions aufzeigen. An dieser Stelle sei nur so viel gesagt, dass man dadurch den so genannten *Gimbal Lock* (dt. Kardanring-Blockade) einer Kamera verhindern kann.

*Warum
Quaternions?*

Gimbal Lock

Alte Trägheitsnavigationssysteme verwendeten eine fixierte Plattform, die durch Kreiselkomasse unabhängig von der Bewegung des (Raum- oder Luft-)Fahrzeugs immer die gleiche Orientierung im Raum einhielt. Damit dies ermöglicht werden konnte, benötigte diese Plattform eine flexible Verbindung zu dem Fahrzeug, über die sie auf allen drei Achsen (X, Y und Z) frei rotiert werden konnte.

Diese Verbindung bestand aus mehreren ineinander liegenden Kardanringen (engl. Gimbals) verschiedener Größe, die über Gelenke miteinander verbunden waren. Dabei gab die Befestigung der Gelenke des größten Rings die Rotationsachse für den nächstkleineren Ring vor. Um eine Bewegung auf drei Achsen zu ermöglichen, benötigte man logischerweise mindestens drei solcher Ringe. Nun ist es aber möglich, den innersten von diesen drei Ringen in einer bestimmten (Flug-)Lage ungewollt so auszurichten, dass seine Befestigungsgelenke genau mit denen des äußersten Rings übereinstimmen. Die dritte Rotationsachse ist also identisch mit der ersten Achse, wodurch das Fahrzeug die Fähigkeit verliert, sich auf drei Achsen frei auszurichten. Dies bezeichnet man als Gimbal Lock.

Das Problem lässt sich durch die Verwendung eines zusätzlichen vierten Rings lösen. Der Mondflug von Apollo 13 hatte beispielsweise auch dieses Problem, nachdem die Steuerung nach einer Explosion im Tank

beschädigt war. Damals hatte man nur drei Kardanringe benutzt und die Piloten entsprechend ausgebildet, nur Flugmanöver durchzuführen, die nicht zu einem Gimbal Lock führen würden.

In der Computergrafik kennt man dieses Problem auch, da man hier ähnlich den drei Kardanringen normalerweise mit den drei Weltachsen als eine Art fixierter Referenzplattform arbeitet. Rotiert man eine Kamera beispielsweise um 90 Grad auf der Z-Achse nach links, so entspricht der Vektor der lokalen X-Achse der Kamera nach der Rotation dem Vektor der Y-Achse, womit die folgenden Rotationen um die X- und Y-Achse identisch sind. So hat die Kamera einen Freiheitsgrad der Rotation verloren – sie leidet nun ebenfalls am Gimbal Lock.

Grundlegende Operationen

Erstaunlicherweise sind Quaternions für das, was sie leisten können, relativ simpel aufgebaut. Ich zeige hier zunächst einmal die Klassendefinition, dann gehen wir etwas mehr ins Detail.



```
class __declspec(dllexport) ZFXQuat {
public:
    float x, y, z, w;

    ZFXQuat(void) { x=0.0f, y=0.0f, z=0.0f, w=1.0f; }

    ZFXQuat(float _x, float _y, float _z, float _w)
        { x=_x; y=_y; z=_z; w=_w; }

    void GetMatrix(ZFXMatrix *m);
    void MakeFromEuler(float fPitch,
                       float fYaw,
                       float fRoll);

    ZFXQuat operator * (const ZFXQuat &q)const;
    void operator *= (const ZFXQuat &q);
}; // class
```

Konstruktoren

Wie unschwer zu erkennen ist, kommen wir hier mit gerade mal drei Funktionen neben den beiden Konstruktoren aus. Der Standard-Konstruktor erstellt das Quaternion als *Identity-Quaternion* (neutrales Element bezüglich der Multiplikation), der zweite Konstruktor initialisiert es mit den angegebenen Werten.

Ein Quaternion mit einem Nullvektor und der Zusatzzahl 1 bezeichnet man als Identity-Quaternion. Es sieht wie folgt aus:

$$q_I = [1, (0, 0, 0)]$$

Dann haben wir in unserer Klasse noch die Definition des Multiplikations-Operators. Analog gibt es natürlich auch Funktionen für die anderen Grundrechenarten, also beispielsweise für die Addition und Subtraktion zweier Quaternions. Allerdings benötigen wir in unserer einfachen Engine lediglich die Multiplikation.

Die verbleibenden beiden Funktionen dienen dazu, ein Quaternion aus drei Euler-Winkeln zu erstellen bzw. eine Rotationsmatrix aus einem Quaternion zu konstruieren. Hier gibt es auch entsprechende inverse Funktionen bzw. noch ganz andere Funktionen, die man mit Quaternions ausführen kann. Hier beschränken wir uns aber auf das notwendige Minimum.

Multiplikation zweier Quaternions

Die Multiplikation zweier Quaternions ist nicht kommutativ. Das bedeutet, dass Quaternion C ungleich Quaternion D ist, wenn $C=A*B$ und $D=B*A$ gilt, wobei A und B ebenfalls Quaternions sind. Allerdings erkennt man hier auch, dass bei der Multiplikation zweier Quaternions ebenfalls ein Quaternion das Ergebnis bildet. Die Multiplikation zweier Quaternions q und r ist wie folgt definiert:

$$\text{Geg.: } q = [q_w, (q_v)], r = [r_w, (r_v)]$$

$$\text{Ges.: } q * r = [q_w r_w - q_v r_v, (q_v \times r_v + r_w q_v + q_w r_v)]$$

Der Realteil des neuen Quaternions setzt sich also aus der Differenz des Produkts der beiden ursprünglichen Realteile und des Punktprodukts der beiden ursprünglichen Vektoren zusammen. Der Vektor des neuen Quaternions entsteht aus der Summe des Kreuzprodukts der beiden ursprünglichen Vektoren und der Skalierung der beiden ursprünglichen Vektoren mit dem Realteil des jeweils anderen ursprünglichen Quaternions. In unserer Implementierung sieht das dann wie folgt aus:

```
void ZFXQuat::operator *= (const ZFXQuat &q) {
    float _x, _y, _z, _w;

    _w = w*q.w - x*q.x - y*q.y - z*q.z;
    _x = w*q.x + x*q.w + y*q.z - z*q.y;
    _y = w*q.y + y*q.w + z*q.x - x*q.z;
    _z = w*q.z + z*q.w + x*q.y - y*q.x;

    x = _x;
```



Operatoren

Methoden

Kommutativ-
Gesetz



```

    y = _y;
    z = _z;
    w = _w;
}

/*-----*/

ZFXQuat ZFXQuat::operator * (const ZFXQuat &q) const {
    ZFXQuat qResult;

    qResult.w = w*q.w - x*q.x - y*q.y - z*q.z;
    qResult.x = w*q.x + x*q.w + y*q.z - z*q.y;
    qResult.y = w*q.y + y*q.w + z*q.x - x*q.z;
    qResult.z = w*q.z + z*q.w + x*q.y - y*q.x;

    return qResult;
}

```

Konstruktion aus Euler-Winkeln

Um mit einem Quaternion sinnvoll arbeiten zu können, müssen wir es auch mit sinnvollen Informationen füttern können. Am greifbarsten sind für uns natürlich die Euler-Winkel. Sie sind nach dem Schweizer Mathematiker Leonard Euler (1707-1783) benannt. Sie beschreiben eine Rotation im 3D-Raum auf den drei lokalen Achsen, die man auch aus dem kubischen Koordinatensystem kennt. Man hat also drei Winkel, die entsprechend die Rotation auf der X-Achse (engl. Pitch), der Y-Achse (engl. Yaw) und der Z-Achse (engl. Roll) eines Objekts definieren.

Ist eine solche Rotation für ein Objekt bekannt, dann können wir aus diesen drei Rotationswinkeln ein Quaternion erstellen, das eine gleichwertige Ausrichtung eines Objekts repräsentiert.

Unit-Quaternions

Um jedoch eine Rotation durch ein Quaternion repräsentieren zu können, muss es sich dabei um ein Einheits-Quaternion (engl. Unit-Quaternion) mit dem Betrag 1 handeln. Für einen beliebigen dreidimensionalen Vektor v gilt dabei für ein Unit-Quaternion:

$$q = [\cos(a), \sin(a) \ v]$$

Die folgende Funktion erstellt aus drei gegebenen Euler-Winkeln eine gleichwertige Repräsentation der Rotation im 3D-Raum durch ein Quaternion:



```

void ZFXQuat::MakeFromEuler(float fPitch, float fYaw,
                           float fRoll) {
    float cX, cY, cZ, sX, sY, sZ, cYcZ, sYsZ, cYsZ, sYcZ;

    fPitch *= 0.5f;
    fYaw    *= 0.5f;
    fRoll   *= 0.5f;

    cX = cosf(fPitch/2);
    cY = cosf(fYaw/2);
    cZ = cosf(fRoll/2);

    sX = sinf(fPitch/2);
    sY = sinf(fYaw/2);
    sZ = sinf(fRoll/2);

    cYcZ = cY * cZ;
    sYsZ = sY * sZ;
    cYsZ = cY * sZ;
    sYcZ = sY * cZ;

    w = cX * cYcZ + sX * sYsZ;
    x = sX * cYcZ - cX * sYsZ;
    y = cX * sYcZ + sX * cYsZ;
    z = cX * cYsZ - sX * sYcZ;
} // MakeFromEuler

```

Rotationsmatrix zu einem Quaternion

Da aber die meisten APIs heutzutage mit Rotationsmatrizen anstelle von Euler-Winkeln oder Quaternions arbeiten, benötigen wir auch eine Methode, um aus einem Quaternion eine gleichwertige Rotationsmatrix zu erzeugen, die dieselbe Rotation repräsentiert. Unter der Voraussetzung, dass es sich um ein Unit-Quaternion handelt, gilt die folgende Formel zur Erstellung der entsprechenden Rotationsmatrix:

$$\begin{vmatrix}
 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy & 0 \\
 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wz & 0 \\
 2xz-2wy & 2yz+2wz & 1-2x^2-2y^2 & 0 \\
 0 & 0 & 0 & 1
 \end{vmatrix}$$

Die folgende Funktion implementiert diese Formel, mit deren Hilfe wir dann eine ganz normale Rotationsmatrix aus einem Quaternion erstellen können.



```
void ZFXQuat::GetMatrix(ZFXMatrix *pMat) {
    float wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;

    // Einheitsmatrix
    memset(pMat, 0, sizeof(ZFXMatrix));
    pMat->_44 = 1.0f;

    x2 = x + x;
    y2 = y + y;
    z2 = z + z;

    xx = x * x2;
    xy = x * y2;
    xz = x * z2;

    yy = y * y2;
    yz = y * z2;
    zz = z * z2;

    wx = w * x2;
    wy = w * y2;
    wz = w * z2;

    pMat->_11 = 1.0f - (yy + zz);
    pMat->_12 = xy - wz;
    pMat->_13 = xz + wy;

    pMat->_21 = xy + wz;
    pMat->_22 = 1.0f - (xx + zz);
    pMat->_23 = yz - wx;

    pMat->_31 = xz - wy;
    pMat->_32 = yz + wx;
    pMat->_33 = 1.0f - (xx + yy);
} // GetMatrix
```

Quaternions und Kameras

So viel zu Quaternions. Diese werden uns erst sehr viel später in diesem Buch in einer realen Anwendung begegnen, nämlich dann, wenn wir unsere Kamera-Klasse implementieren. Unsere Quaternion-Klasse ist bisher noch sehr rudimentär, aber durch eine Internet-Recherche kann man dieser Klasse sehr viel Funktionalität hinzufügen, sofern man sie benötigen sollte.

4.9 Ein Blick zurück, zwei Schritt nach vorn

Ein langer Weg – in beide Richtungen

Der obligatorische Blick zurück zeigt diesmal einen großen Wust an Klein-kram, den wir allerdings erfolgreich zu einem in sich schlüssigen Ganzen zusammengeführt haben. Zu Beginn dieses Kapitels haben wir viel darüber gelernt, mit welchen Mitteln wir unsere 3D-Berechnungen beschleunigen

können. Daran anschließend haben wir uns dann die notwendigen Werkzeuge geschaffen, um diese Beschleunigung ausnutzen zu können. Gerade der Teil der Assembler- und SSE-Programmierung war sicherlich für einige eine neue Herausforderung, aber ein schlauer Mann hat mal gesagt, dass das Leben ja langweilig wäre, wenn es nur aus Pfannkuchen bestünde. :-)

Eine vernünftige Mathematik-Basis kann ich für die Entwicklung einer eigenen Engine oder eines eigenen Spiels nur dringend anraten, da man idealerweise in seinem Code irgendwo eine Blackbox für all die mathematischen notwendigen Übel dieser Welt hat. Es gibt nichts Schlimmeres, als innerhalb der Implementierung eines recht komplexen Algorithmus auch noch etliche Zeilen Code zu haben, die sich mit der Umsetzung mathematischer Operationen beschäftigen. Wir haben hier zwar schon einen guten Satz Material zur Hand, aber davon kann man ja nie genug haben. Daher möchte ich hier ein echtes Kleinod zur Vertiefung empfehlen. Das Buch *Geometric Tools for Computer Graphics* von David Eberly und Philip J. Schneider gehört einfach in die Bibliothek eines jeden 3D-Programmierers.

Weitere Literatur

Nun sind wir gewappnet, für all das, was die 3D-Engine an grundlegenden 3D-Mathematik-Aufgaben an uns stellen wird. Durch unsere eigenen Klassen können wir diese Aufgaben dann nicht nur schnell und effektiv, sondern auch noch schön objektorientiert und damit auch optisch elegant lösen. Dies ist nicht nur ein Punkt, der für eitle Programmierer wichtig ist. Vielmehr ist es eine Frage der Übersichtlichkeit und der guten Wartbarkeit des Codes. Geraten wir beispielsweise in die Verlegenheit, unseren Code erweitern zu müssen (beispielsweise für neue SIMD-ähnliche Techniken), dann kann die gute Wartbarkeit des Codes, unabhängig von dem Programmierer, der ihn geschrieben hat, zu einem zeitkritischen Faktor werden.

*Doch wir sind
gerüstet*

Im nun folgenden Kapitel verlassen wir jedoch zunächst die Gefilde der 3D-Mathematik (wenn auch nicht hundertprozentig) und kümmern uns um unseren Renderer. Bisher können wir kaum mehr, als den Bildschirm von *Windows* zu übernehmen und ihn einfarbig zu löschen. Jetzt ist es an der Zeit, auch etwas Sinnvolleres auf den Bildschirm zu bringen. Nun werden wir lernen, Projektionsmatrizen und Kameramatrizen einzustellen, Texturen zu laden und 3D-Grafik am Bildschirm anzuzeigen.

*Und so sieht's
aus.*

Für dieses Kapitel gibt es auf der CD zwar den kompletten Quelltext, den ihr euch zu einer statischen Bibliothek kompilieren könnt sowie auch die Bibliothek selbst. Aber es gibt kein fertiges Programm, mit dem man die Bibliothek testen könnte. Wir könnten ja nicht mehr tun, als ein paar Vektoren zu addieren oder zu transformieren und uns das Ergebnis ausgeben lassen. Und wo nichts ist, da soll man nichts erzwingen, also lassen wir das.

Wermutstropfen

Zudem bitte ich zu beachten, dass sich der vorliegende Quellcode nur kompilieren lässt, wenn man auf dem Entwicklungsrechner ein Betriebssystem, eine Entwicklungsumgebung und eine CPU (ab *Pentium III* oder den neues-

Remembrall

ten *AMD Athlon*) hat, die alle SSE unterstützen, beispielsweise *Visual C++ Professional* (mit Service- und Processor Pack) oder *Visual C++ .NET Standard* und *Windows XP*. Die fertig kompilierte Bibliothek kann jedoch von jedem Rechner aus in jedem anderen Projekt verwendet werden. Sie wird jedoch nur die schnellen SSE-Funktionen verwenden, wenn der Rechner, auf dem das Programm dann läuft, eine entsprechende CPU und ein entsprechendes Betriebssystem hat. Wer ein nicht-SIMD-fähiges System zum Kompilieren verwenden will, der muss alle SIMD-spezifischen Codes aus den Quelltexten entfernen, weil der Compiler diese nicht erkennt und als Fehler meldet.

Teil 2 Rendern von Grafik

Kapitel 5:	Materialien, Texturen und Transparenz	215
Kapitel 6:	Das Render-Interface der ZFXEngine	259
Kapitel 7:	3D-Pipeline und Shader	361
Kapitel 8:	Skeletale Animation von Charakteren	411

5 Materialien, Texturen und Transparenz

»Wenn zwei Menschen immer dasselbe denken, ist einer von ihnen überflüssig.« (Winston Churchill)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ sinnvolles Management von Ressourcen
- ➔ Grundlagen der Definition von Licht und Materialien
- ➔ erweiterter Realismus durch Verwendung von Texturen
- ➔ das Laden von Grafikdateien als Texturen für 3D-Objekte
- ➔ Erzeugung von Transparenzeffekten mit Texturen

5.1 Mittleres Management

Das einleitende Zitat dieses Kapitels drückt den Sachverhalt zwar ein wenig unromantisch aus, es birgt jedoch in seinem Kern eine große Wahrheit. Diese lässt sich auch ein wenig modifiziert auf den Bereich übertragen, mit dem wir uns hier beschäftigen, nämlich mit der Entwicklung von Software. Ganz herzlos technisch betrachtet, sagt das Zitat doch nur aus, dass Redundanz überflüssig ist. Dies ist eine der wichtigsten Grundsätze, die wir uns bei der Entwicklung von Software für Computergrafik auf die Fahnen schreiben sollten. Die Geschwindigkeit unseres Codes ist von elementarer Bedeutung. Auf der offensichtlichsten Ebene bedeutet dies, dass derjenige Code am schnellsten ist, den wir gar nicht ausführen – mal abgesehen davon, wie sinnvoll solcher Code ist. Am langsamsten ist natürlich der Code, der unnötigerweise redundant ausgeführt wird, insbesondere wenn einmaliges Ausführen reichen würde, beispielsweise dann, wenn man während des Programmablaufs oder gar innerhalb einer Funktion Werte mehrfach berechnet, die man einmal hätte initialisieren und als Konstante verwenden können. Muss ich mehr sagen als $2 * \pi$?

*Churchills
Wahrheit*

Doch wenn man etwas tiefer gräbt, sorgt nicht nur schneller Code mit möglichst wenig komplizierten mathematischen Berechnungen für schnellen Code. Auch die Verwendung des Speichers kann für Geschwindigkeitsverlust sorgen. Wenn man es »richtig« macht, dann kann man durch schlechtes

*Speicher ist
Speed*

Speichermanagement sogar mehr Performance killen als durch das Aufrufen einer Wurzelberechnung in einer Endlosschleife. Man stelle sich nur einmal vor, dass man ein 3D-Modell geladen hat, das auf allen seinen dreitausend Polygonen dieselbe Grafik als Textur verwendet. Laden wir diese Grafik nun dreitausendmal, dann haben wir den schnellen Speicher der Grafikkarte bereits nach ein paar hundert Instanzen dieser Grafik hoffnungslos ausgeschöpft.¹ Selbst der Arbeitsspeicher des Computers kommt hier an seine Grenzen und darüber hinaus. Gehen wir aber mal davon aus, dass der Arbeitsspeicher reichen würde, um die Datenmenge zu fassen. Der Performance-Killer ist in diesem Fall der Transfer der Daten von dem Arbeitsspeicher über den Bus des Mainboards hin zum VRAM der Grafikkarte. Dass die Grafik im VRAM liegt, ist aber immer dann nötig, wenn sie (z.B. als Textur) zur Anzeige am Monitor verwendet werden soll. Nach jedem einzelnen Polygon des Modells wird eine andere Instanz derselben Grafik für die Grafikkarte eingestellt. Fast bei jedem Switch muss diese Grafik in Form mehrerer Tausend Bytes über den Bus geschoben werden. Dagegen geht selbst ein mit Handbremse anfahrrender Käfer ab wie eine Rakete.

*Und die Moral von
der Geschicht' ...*

Damit schließt sich der Kreis zu unserem leicht modifizierten Zitat. Wenn zwei Datensätze exakt identische Daten enthalten, dann ist einer von ihnen überflüssig. Im Falle unseres 3D-Modells ist es vollkommen ausreichend, die Grafik ein einziges Mal zu laden und zur Grafikkarte zu schicken. Die knapp 250 Kbyte an Daten bleiben dann dort im VRAM bequem liegen und werden von dort aus für alle dreitausend Polygone des Modells verwendet. Das bedeutet, dass in diesem Fall keinerlei Verkehr über den Bus notwendig ist – und schon gar nicht so viel wie im vorherigen Fall. Optisch macht das am Bildschirm keinen Unterschied. Es spart uns (bzw. dem Computer) aber die Arbeit, dreitausend Textur-Switches auszuführen und bis zu dreitausendmal identische Daten über den Bus zu quälen. Im Endeffekt wirkt sich das auf die Programmgeschwindigkeit aus, wie der Wechsel von einem 1-ES-Motor² zu dem Warp-Antrieb der Enterprise D.

*Management
versus Bürokratie*

Das sehen wir also alles ein. Doch was hat das mit Management zu tun? Nun, streng theoretisch muss man Management nach Staehle sowohl institutional als auch funktional betrachten.³ Das bedeutet, dass es immer eine Institution (z.B. eine Person) gibt, die etwas *managt*, und dass es bestimmte Funktionen gibt, die das Managen überhaupt erfüllen soll. Seit den 90er Jahren gibt es bei personalstarken Unternehmen den Trend, das mittlere Management auszudünnen. Das hat unter anderem den Sinn, dass die Kommunikationsketten zwischen den oberen und unteren Führungsebenen kürzer und damit effektiver werden. Doch das macht nicht immer und überall

1 Der Speicherverbrauch hängt natürlich von der Größe und Auflösung (Bittiefe) der Grafik ab. Wir gehen hier von einer durchschnittlichen Textur von ca. einem Viertel Megabyte aus, was in etwa einer 256x256x24-Textur entspricht.

2 ES = Esel-Stärken

3 Management; München, 1994

Sinn. Übertragen wir dieses Problem wieder auf unseren Bereich, die Software-Entwicklung. Die unterste Ebene ist bei uns quasi das Programm, also der Quellcode des Computerspiels an sich. Von dort aus wird beispielsweise eine bestimmte Menge an Polygonen an das Top-Management geschickt. Das ist in unserem Fall die Grafikkarte. Diese arbeitet mit den von der unteren Ebene zur Verfügung gestellten Informationen und entscheidet nach Analyse und Bearbeitung der Daten in der 3D-Pipeline endgültig darüber, welche Polygone als Pixel auf dem Bildschirm landen und welche nicht. Damit haben wir eine schöne flache Hierarchie ohne lange Kommunikationswege. Doch genau auf diese Weise entstehen Probleme wie die mit den dreitausend Texturen.

Im Verlauf dieses Buches werden wir es vor allem mit einem Problem zu tun haben. Und das ist das Management von Daten. Wie bereits weiter oben dargestellt, ist es keine gute Idee, dem Top-Management mehr Arbeit als nötig aufzudrücken. Was interessiert es denn den Top-Manager, also die Grafikkarte, wie viele Polygone ein Level insgesamt hat und welche Texturen diese verwenden? Müsste er sich damit befassen, dann käme er vor lauter Analysieren der Daten gar nicht mehr dazu, sinnvolle Entscheidungen innerhalb eines vernünftigen Zeitrahmens zu treffen. Aus diesem Grunde hat man die *Executive Summary* erfunden, also sozusagen die Kurzfassung für Führungskräfte, die lediglich die wichtigsten, entscheidungsrelevanten Daten enthält. Zwischen den beiden Schichten, d.h. dem Quelltext mit dem spielrelevanten Code am unteren Ende und der Grafikkarte am oberen Ende der Hierarchie, sollte also immer noch eine weitere Schicht sitzen. Und eben diese Schicht ist das mittlere Management in unserer Software. Damit haben wir auch schon die institutionale Seite der Dinge geklärt. Unsere Middle-Manager sind Instanzen einer entsprechenden Manager-Klasse, die die funktionale Seite des Managements für einen spezifischen Aufgabenbereich implementiert.

*Unser mittleres
Management*

Die Aufgaben eines Managers können dabei mindestens ebenso unterschiedlich sein wie die verschiedenen Management-Stile, die man in der Betriebswirtschaftslehre kennt. Das möchte ich uns an dieser Stelle aber ersparen, und ich beziehe mich daher gleich auf unsere konkreten Manager aus Fleisch und – ich meine: aus Silizium und Strom. Im Gegensatz zu einem modernen mittelständischen Unternehmen mit einer flachen Hierarchie haben wir in unseren Software-Projekten eine große Menge an Stellen auf der mittleren Führungsebene, die wir mit Managern besetzen müssen. In späteren Kapiteln werden wir noch einige Manager-Klassen entwerfen und implementieren. Immer wenn wir in unserer Engine mit großen Datenmengen arbeiten, ist es sinnvoll, einen Manager einzustellen, der Entscheidungen darüber trifft, wie diese Daten effektiv verwaltet und verwendet werden können. So bekommt das Top-Management, also die Grafikkarte und auch die CPU, nur die Daten zu Gesicht, die es wirklich für seine Arbeit benötigt – eben unsere Version einer *Executive Summary*. Wer das erste Kapitel auf-

*Funktionales
Management*

merksam gelesen hat, der wird hier deutliche Parallelen zu der Salopp-Definition einer Engine gefunden haben. Und tatsächlich ist es doch die Aufgabe einer Engine, das Management der Daten von der Festplatte zu übernehmen, so dass diese effektiv über die Hardware-Ebene (Bildschirm, Tastatur, Maus, ...) dem Spieler als interaktive, multimediale Umgebung angeboten werden können.

Voll konkret

In diesem Kapitel werden wir gleich unseren ersten Manager kennen lernen. Im Folgenden geht es um Materialien und Texturen. Was genau das ist und wozu man das braucht, dazu erkläre ich nachher noch mehr. Vereinfacht gesagt, gibt man durch Materialien an, wie die Oberfläche eines Objekts einfallendes Licht reflektiert oder gar emittiert. Texturen hingegen sind zweidimensionale Grafiken, die auf die Oberflächen eines 3D-Modells aufgeklebt werden, damit es realistischer aussieht. Wenn wir also 3D-Modelle, beispielsweise einen Level oder ein Monster, von der Festplatte in unser Spiel laden, dann haben wir zum einen eine Menge von Polygonen, zum anderen aber auch eine Menge von Texturen und Materialien, die von diesen Polygonen verwendet werden. In diesem Kapitel geht es darum, Materialien und Texturen durch einen Manager verwalten zu lassen. Unser Ziel ist es beispielsweise, Materialien und Texturen jeweils nur ein einziges Mal im Speicher zu haben und nicht mehrfach von der Festplatte laden zu müssen und den Speicher unnötig vollzumüllen. Diese Aufgabe scheint zunächst nicht sonderlich schwer zu sein. Aber spätestens dann, wenn zwei vollkommen verschiedene 3D-Modelle beispielsweise dieselbe Grafik als Textur verwenden, ist es schon ein wenig knifflig herauszufinden, ob wir diese Grafik bereits geladen haben oder nicht. Daher konkretisiert man diese Anforderung am besten in einem eigenständigen Stück Code und beispielsweise nicht durch Hardcoding in den Laderoutinen für 3D-Modelle in der Art von: *Wenn Modell A schon geladen ist, dann lade Textur X und Y nicht, aber wenn Modell B geladen ist und A nicht, dann lade nur Textur X und Textur Z, aber nicht Textur Y.*

Als Erstes stellen wir daher einen Manager ein, der sich mit der Verwaltung und auch der Beschaffung von Material- und Texturdaten befasst. Das kann man sich ein wenig wie einen Versorgungsoffizier (VO) vorstellen. Das untere Management (Laderoutine für 3D-Modelle) schreibt eine Anforderung an den VO auf der mittleren Management-Ebene, dass es für das 3D-Modell eines Panzers eine Textur mit Fleckentarn-Bemalung benötigt. Der VO schaut dann in seinem Lager (RAM) nach, ob dort eine entsprechende Grafik vorhanden ist. Falls nicht, so ist er dafür verantwortlich, diese Grafik zu beschaffen (von der Festplatte laden) und eine Inventarnummer zu vergeben. Ist diese Grafik aber schon vorhanden, so muss er nur die Inventarnummer herausgeben. Wird diese Grafik aus dem Bestand dann konkret benötigt (beim Rendern des Modells), dann geht der Renderer mit der Inventarnummer wieder zum VO und lässt sich die Grafik für den Rendervorgang zur Verfügung stellen.

Das klingt doch logisch, oder? Dann wollen wir die Arbeit mal aufnehmen.

5.2 Eine Klasse für Skins

Der folgende Abschnitt befasst sich zunächst damit, die Begriffe Textur, Licht und Material zu klären. Danach machen wir uns an die Arbeit, all diese Begriffe in einer Struktur zu vereinen, damit wir sie einfacher handhaben und managen können. Eine solche Kombination aus Textur und Material nenne ich hier in Ermangelung eines besseren Namens *Skin*.

Am Ende dieses Kapitels wird dann ein Manager stehen, der Skins laden und verwalten kann. Da ich die Grundlagen von Texturen, Material und Licht als bekannt voraussetze, werde ich die Grundlagen entsprechend schnell abhandeln. Der Schwerpunkt soll im Folgenden auf dem Management von Skins liegen.

Texturen

Kommen wir nun zu den 3D-Modellen, die wir am Bildschirm anzeigen lassen wollen. Grundsätzlich gilt: Je weniger Polygone ein 3D-Modell verwendet, desto schneller kann es berechnet und am Bildschirm angezeigt werden. So ist es beispielsweise keine gute Idee, bei einem Mosaiktisch jedes einzelne Teilchen als Polygon zu erzeugen. Vielmehr kann man hier die Platte des Tisches als einzelnes Polygon modellieren. Doch damit sieht das Ding nicht grade wie ein Mosaiktisch aus. Um dies dennoch vorzutäuschen, kann man Texturen verwenden. Eine Textur ist eine einfache zweidimensionale Grafik, die auf das 3D-Modell gelegt wird. Diese Grafik zeigt dann ein Abbild des 3D-Modells, wie es in der realen Welt aussähe. Abbildung 5.1 zeigt uns ein 3D-Modell einmal mit und einmal ohne eine Textur.



Abbildung 5.1:
Ein 3D-Modell; links
mit und rechts ohne
Textur

Wie man sehen kann, erspart man sich durch die Illusion, die die Textur erzeugt, das Modellieren vieler Details des Modells durch Polygone. Dies beschleunigt die Darstellung des Modells am Bildschirm ganz enorm. Ein Problem dabei ist aber, dass eine 3D-Engine in der Regel die Beleuchtung eines Modells basierend auf den Vertices der Polygone vornimmt. Zeigt eine Textur also scharfe Kanten des Objekts, die aber auf einem einzigen Polygon liegen, so werden diese in der Regel falsch und unrealistisch beleuchtet.

Detailgrad

Durch verschiedene andere Effekte kann man die Illusion durch Texturen aber weiter im Realitätsgrad erhöhen, beispielsweise durch Bump-Mapping.

Bump-Mapping

Die Beleuchtung eines gerenderten 3D-Objekts wird für gewöhnlich anhand des Winkels der Vertex-Normalenvektoren des Polygons zu den vorhandenen Lichtquellen berechnet. Legt man eine Textur auf ein Polygon, so täuscht diese Textur dem Betrachter einen wesentlich höheren Polygon-Detailgrad des Modells vor. Dies geht natürlich auf Kosten der realistischen Beleuchtung, da weiterhin nur die Vertex-Normalenvektoren des einzelnen Polygons verwendet werden.

Das so genannte Bump-Mapping verschafft hier Abhilfe. Neben der Textur lädt man zusätzlich eine Bump-Map. Diese speichert jedoch in ihren einzelnen Pixeln keine RGB-Farbwerte, sondern Tiefeninformationen für die Textur, also eine Art Heightmap. Aus dieser Bump-Map kann man dann für jeden korrespondierenden Pixel der Textur einen eigenen, echten Normalenvektor berechnen. Die Beleuchtung berechnet man nun nicht mehr anhand der Vertex-Normalenvektoren, sondern pixelweise anhand der Normalenvektoren der Pixel. So kann man 3D-Modelle trotz niedrigen Polygrads sehr detailliert und damit realistisch beleuchten.

Licht und Material

In einer 3D-Szene, die man darstellt, verwendet man zur Steigerung des Realismus bestimmte Arten von Lichtquellen. Die realistische Darstellung von Licht, Schatten und Schattierungen trägt einen Großteil zum Betrug des Auges bei, dem man eine virtuelle Umgebung als möglichst real verkaufen möchte. In der Computergrafik unterscheidet man grob gesagt zwischen zwei großen Gruppen von Lichtarten. Zum einen ist dies **ambientes Licht** (Ambient Light) und zum anderen **direktes Licht** (Direct Light). Das ambi-ente Licht ist das Licht, das unterschwellig fast überall vorhanden ist. Es handelt sich dabei um Licht, das vielfach reflektiert wurde und daher einen Raum scheinbar von überall her mit gleicher Intensität ausleuchtet und keiner Lichtquelle zuzuordnen ist. Das direkte Licht lässt sich im Gegensatz dazu immer einer Lichtquelle zuordnen. Bekanntermaßen kennt man dafür in der Computergrafik die folgenden drei Typen von Lichtquellen, die weiter unten auch in Abbildung 5.2 dargestellt sind:

- ➔ Directional Light
- ➔ Point Light
- ➔ Spot Light

Als Directional Light (gerichtetes Licht) bezeichnet man das Licht extrem weit entfernter Lichtquellen. Zur Vereinfachung der Berechnung kann man annehmen, dass die Lichtstrahlen solcher Lichtquellen am Punkt unseres Interesses parallel eintreffen. Das vereinfacht die Berechnung deshalb, weil man den Lichtwert eines beleuchteten Objekts einfach als Winkel zwischen dem Normalenvektor des Objekts und den einfallenden Lichtstrahlen berechnen kann. Man muss nicht berücksichtigen, dass die einfallenden Lichtstrahlen auch in unterschiedlichen Winkeln auftreffen, wie das bei anderen Lichtarten der Fall wäre. Beispielsweise kann man das Licht der Sonne oder eines entfernten Sterns als Directional Light betrachten. Die einzigen Daten, die man zur Berechnung der Beleuchtung solcher Lichtquellen benötigt, sind die Richtung, in die das Licht verläuft, und seine Farbe.⁴

Directional Light

Ein Point Light ist eine Lichtquelle, die – wie der Name schon sagt – punktförmig ist. Dabei strahlt sie ihr Licht gleichmäßig in alle Richtungen ab. Bei der Berechnung einer solchen Lichtquelle benötigt man vor allem die Position der Lichtquelle und die Lichtfarbe sowie die Reichweite des Lichts. Die 3D-Pipeline errechnet daraus für jedes zu beleuchtende Objekt die Intensität des Lichtes aufgrund der Entfernung und des Auftreffwinkels des Lichtes.

Point Light

Ein Spot Light ist sozusagen eine der normalsten Lichtquellen in unserer Welt. Es handelt sich dabei um eine Lichtquelle, die von einer bestimmten Position aus Licht in eine bestimmte Richtung abstrahlt – eben ein Lichtspot. Dieser ist analog zu einem Point Light, nur dass er sein Licht nur in einem bestimmten Winkel und nicht gleichmäßig in jede Richtung abstrahlt. Das simuliert beispielsweise den Lampenschirm oder die Fassung einer Glühbirne, die das gleichmäßige Abstrahlen des Lichtes in alle Richtungen verhindern. Für so ein Spot Light muss man sowohl die Position als auch die Richtung und Reichweite des Lichts angeben. Daneben sind noch andere Daten notwendig, wie beispielsweise der Winkel, in dem das Licht abgestrahlt wird. Ein Spot Light hat sogar zwei solcher Winkel, nämlich einen inneren für den Hauptstrahl und einen äußeren, mit dem man einen Ring schwächeren Lichts um den Hauptkegel herum simulieren kann.

Spot Light

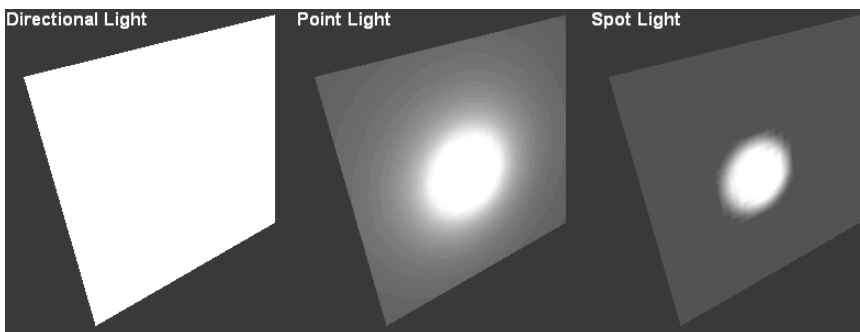


Abbildung 5.2:
Beleuchtung eines dunkelgrauen Rechtecks durch die drei verschiedenen Typen direkten Lichts

⁴ Durch die Farbe gibt man automatisch auch die Intensität mit an.

*Erläuterung zur
Abbildung*

In Abbildung 5.2 sehen wir die drei Typen direkten Lichts, die auf ein dunkel-graues Rechteck scheinen. Das Directional Light beleuchtet die gesamte Fläche gleichmäßig hell, so dass sie fast weiß erscheint. Das Point Light bescheint die Fläche sehr intensiv und verläuft zu den Rändern des Rechtecks hin deutlich. Das Spot Light hingegen hat einen extrem scharf abgegrenzten Lichtkegel auf der Fläche, wobei der Großteil der Fläche des Rechtecks nicht beleuchtet wird und seine ursprüngliche Farbe behält.

*: -)
TIPP*

Wie immer in der Computergrafik ist es so, dass das, was mehr Realismus erzeugt und mehr Aufwand erfordert, wesentlich langsamer zu berechnen ist als ein vereinfachtes Modell. Daher ist ambientes Licht am schnellsten zu berechnen; Spot Lights hingegen kosten verhältnismäßig viel Rechenzeit. Man sollte also bei der Ausleuchtung seiner 3D-Szene darauf achten, nicht übermäßig viele Lichtquellen zu verwenden.

*Biologie und
Physik des
Sehens*

Doch die Lichtquellen an sich sind nicht das einzig Entscheidende. Anhand einer Lichtquelle kann man zwar entscheiden, wie viel Licht von entsprechenden Emittlern in eine 3D-Szene eingeflutet wird. Das hat jedoch letzten Endes noch erst mal gar keine Bedeutung auf das von der Szene reflektierte Licht. Aktivieren wir kurz einen Flashback-Prozessor, und begeben uns zurück in den Biologie- und Physik-Unterricht an der Schule: Wie funktioniert das Sehen eigentlich? Ohne Licht ist alles dunkel, das ist nicht weiter überraschend.

*f
INFO*

Auch wo viel Licht ist, kann es dunkel sein. Betrachtet man den Wellencharakter des Lichtes, so heben sich zwei exakt gegenphasige Lichtwellen gegenseitig auf, und man hat trotz zweier starker Lichtquellen absolute und perfekte Dunkelheit.

Wenn wir nun eine Lichtquelle aktivieren, dann durchflutet Licht den Raum. Dem Wellen-Teilchen-Dualismus folgend, muss man das aus so genannten Photonen bestehende Licht in bestimmten Situationen mal als Teilchen (analog einer Billiard-Kugel) und mal als Welle (analog einer Funkwelle) betrachten. Nun sehen wir also etwas. Doch was sehen wir? Das, was wir sehen, sind nicht etwa die Objekte im Raum selbst. Nein, alles, was wir mit unseren Augen wahrnehmen können, sind Photonen. Unsere Augen sind also nur Sensoren, die die Umgebung nach einfallenden Photonen scannen. Doch warum sehen wir dann die Objekte, die wirklich da im Raum stehen? Nun, die Photonen treten aus der Lichtquelle aus und durchfluten den Raum. Dabei prallen sie in der Regel an den harten Objekten ab und werden in eine andere Richtung reflektiert. Dann prallen sie wieder von einem Objekt ab und sausen woanders hin und immer so weiter. Entweder verlieren sie im Laufe dieses stressigen Fluges ihre gesamte Energie, oder sie prallen irgendwann zufällig in unser Auge. Dort werden sie absorbiert und in ein Signal an das Gehirn umgesetzt. Folglich sehen wir also nicht die Objekte selbst, sondern nur die von ihnen reflektierten Photonen.

Warum sehen die Photonen, die alle von derselben Lichtquelle stammen, alle unterschiedlich aus und stechen unterschiedlich stark in unser Auge? Schauen wir beispielsweise direkt in einen Spiegel, so kann uns dieser genauso stark blenden wie die Lichtquelle selbst, während das von einer normalen Wand reflektierte Licht für das Auge wesentlich angenehmer ist. Und genau an dieser Stelle kommen Materialien ins Spiel.

Das Material eines Objekts entscheidet darüber, wie die Photonen reflektiert werden. Unter dem Begriff »Material« versteht man zum einen die chemische und zum anderen die physikalische Beschaffenheit der Oberfläche. Beispielsweise sagt man, dass eine Oberfläche stumpf aussieht, wenn sie nicht sehr grell im Licht erscheint. Das liegt daran, dass diese Oberfläche (unter dem Mikroskop betrachtet) sehr rau und uneben ist. Normalerweise werden Photonen wie Billardkugeln abhängig von ihrem Eintreffwinkel reflektiert. Ein Holzbrett ist aber eben nur auf den ersten Blick eine ebene Fläche. Bereits mit der Hand spürt man die Unebenheit, und mit dem Elektronenmikroskop erscheint die scheinbar ebene Fläche wie ein Gebirge. Die Photonen treffen also auf demselben Holzbrett auf unendlich viele Oberflächenwinkel, von denen sie reflektiert werden. Daher streut das Material Holz das gleichmäßig einfallende Licht kreuz und quer in den Raum zurück. Ein Betrachter in einem bestimmten Winkel empfängt daher nur sehr viel weniger der auf das Holz fallenden Photonen auf seinen Scannern (Augen).

Stumpfes Material

Das genaue Gegenteil von stumpfen Materialien sind spiegelnde Materialien. Hier ist die Oberfläche extrem gleichmäßig und eben, selbst auf der Ebene der Photonen. Daher wird das einfallende Licht fast hundertprozentig in einem neuen Winkel reflektiert. Leuchtet man mit einer Lampe direkt in einen Spiegel, so gibt es nur einen relativ kleinen Winkelbereich, in dem man in den Spiegel schauen kann und dabei von der Lampe extrem geblendet wird. In genau diesem Winkel wird der Großteil der Photonen von dem Material gleichmäßig reflektiert.

*Spiegelndes
Material*

Wir wissen bereits, dass der Realitätsgrad einer virtuellen 3D-Szene mit dem Licht und dessen Reflexion durch die Objekte zusammenhängt. Die Komplexität dieses Vorgangs lässt sich aber bei weitem nicht schnell genug berechnen, um in Echtzeit simuliert zu werden. Allein die detaillierten Informationen über die chemische und physikalische Beschaffenheit eines Materials und dessen Simulation würde viel zu lange dauern. Also geht man in der Computergrafik einen anderen Weg. Man definiert zwar neben den Texturen auch noch Materialien für die 3D-Objekte, aber in vereinfachter Form. Ein Material in der Welt der Computer ist nichts anderes als eine Struktur, die Aussagen darüber macht, wie einfallendes Licht reflektiert werden soll. Aber das sehen wir gleich im Detail.

*Simuliertes
Material*

Grundlegende Strukturen

Für unsere Engine benötigen wir diverse Datentypen, um ein einheitliches Arbeiten zu ermöglichen. Hier stelle ich gleich ein paar grundlegende Datenstrukturen vor, die wir in der Datei `ZFX.h` definieren. Damit sind sie nicht nur in der DLL verfügbar, sondern auch für den Benutzer unserer Engine. Schließlich bekommt er diesen Header auch an die Hand und muss ihn in seine Projekte einbinden – er enthält ja unter anderem unsere selbst definierten Rückgabewerte.

*Der kleinste
Nenner*

Die wohl kleinste Einheit bei unserer Arbeit mit Materialien und Licht ist sozusagen eine Farbe. Daher definieren wir den folgenden Datentyp für einen Farbwert in unserer Engine:



```
typedef struct ZFXCOLOR_TYPE {
    union {
        struct {
            float fR;
            float fG;
            float fB;
            float fA;
        };
        float c[4];
    };
} ZFXCOLOR;
```

Eine Farbe besteht bei uns immer aus vier `float`-Werten – jeweils einen für den Rot-, Grün-, Blau- und Alphaanteil einer Farbe. Dabei werden nur Werte im Bereich von 0.0f bis 1.0f als gültig angesehen. Sie entsprechen quasi den Prozentangaben, wie viel Prozent einer bestimmten Farbe in den Lichtwert mit eingeht. Der Alphawert ist wie gewohnt für die Transparenz einer Farbe zuständig.

*Material mal
wieder*

Mit einem solchen Farbwert bewaffnet, können wir uns schon an die nächste Struktur machen, nämlich an eine, mit der wir ein Material, und damit die Eigenschaften für die Lichtreflexion eines Objekts, definieren können. Nun habe ich bei der Darstellung des Lichts weiter oben noch etwas verschwiegen: Das direkte Licht wird in der Computergrafik noch ein weiteres Mal in zwei Kategorien unterteilt. Zum einen haben wir das Diffuse Light und zum anderen das Specular Light. Wenn wir in der Computergrafik eine Lichtquelle definieren, so müssen wir insbesondere drei Dinge angeben: zum einen, welchen Farbwert es zum ambienten Licht der Szene beiträgt, zweitens, welchen Farbwert es zum diffusen Licht der Szene beiträgt, und zu guter Letzt, welchen Farbwert es zum spekulären Licht der Szene beiträgt. Doch was bedeutet das konkret?

Nun, es gibt in der Computergrafik verschiedene Reflexionsmodelle für einfallendes Licht. Die beiden meistverwendeten sind das *Diffusreflexionsmodell* und das *Spekulärreflexionsmodell*. Das Modell der diffusen Reflexion gibt an, wie matte und stumpfe Oberflächen das einfallende Licht reflektieren. Das Modell der spekulären Reflexion gibt an, wie glatte, spiegelnde Oberflächen einfallendes Licht reflektieren. Ersteres ist wesentlich einfacher zu berechnen, denn die Reflexion hängt nur von der Intensität und Richtung des einfallenden Lichts ab. Die Beleuchtung der Szene, die sich daraus ergibt, ist also von der Position des Betrachters unabhängig, da man unterstellt, dass die diffuse Reflexion das Licht durch seine raue Oberfläche gleichmäßig in alle Richtungen streut. Bei der spekulären Reflexion ist das nicht so. Hier ist die Position des Betrachters entscheidend und verkompliziert die Berechnung. Je nachdem, von welcher Position aus der Betrachter die Szene beobachtet, sieht er die Reflexion der spiegelnden Oberfläche sehr intensiv, nur ein wenig oder eben gar nicht. Und da das mit dem ganzen Geleuchte auf den ersten Blick etwas verwirrend ist, spendiere ich euch die Abbildung 5.3. Hier sieht man in vereinfachter Darstellung die Unterteilung des virtuellen Lichtes in ambientes und direktes Licht. Das ambiente Licht benötigt, wie oben erwähnt, keine eigene Lichtquelle, es ist sozusagen einfach da. Wir geben einfach für unsere 3D-Szene einen entsprechenden Lichtfarbwert an. Dann haben wir in der Szene noch beliebige viele Quellen für direktes Licht.⁵ Abhängig von der Art des Materials wird objektweise dann ein Lichtwert diffus und/oder spekulär aus dem direkten Licht berechnet. Dazu wird der ambiente Lichtwert addiert. Das ist schon das ganze Geheimnis.

*Diffus und
spekulär*

Die Gesamtberechnung eines Farbwertes für einen Vertex oder gar einen projizierten Pixel gestaltet sich in der Realität noch ein wenig komplizierter. Nicht nur die direkten Lichtquellen in der Szene beeinflussen die endgültige Helligkeit und Farbe eines Pixels, sondern alle im Folgenden genannten Faktoren:

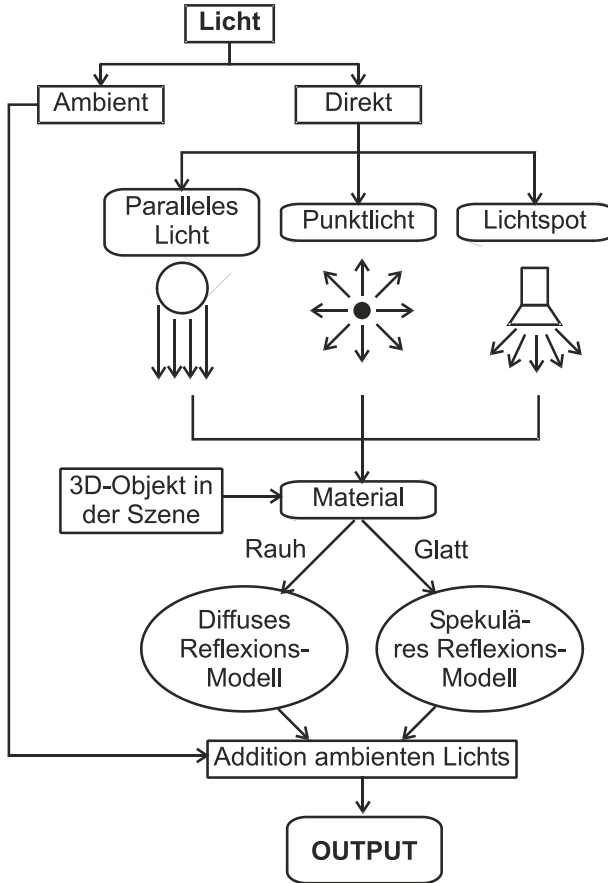
- *Farbe der Vertices*
- *Material der Polygone*
- *Direkte Lichtquellen*
- *Globale ambiente Beleuchtung (ambienter Lichtwert der Szene)*
- *(Textur)*

Während die Farbe der Vertices und die der ambienten Beleuchtung jeweils einen einzigen Farbwert darstellt, können das Material und die direkten Lichtquellen mehrere Farbwerte liefern und so zum ambienten, diffusen und spekulären Licht etwas beisteuern. Aber dazu kommen wir später bei der Definition einer Struktur für Materialien.

⁵ Grafikkarten unterstützen bis zu acht Lichtquellen, in Hardware berechnet. Über eigene Vertex- und Pixel-Shader kann man dieses Limit jedoch umgehen.



Abbildung 5.3:
Vereinfachte Darstellung der möglichen Arten von Lichtquellen in einer 3D-Szene und ihr Einfluss auf die Beleuchtung



Die Textur ist in der obigen Aufzählung auch genannt, da sie (solange sie vorhanden ist) sozusagen die Grundfarbe eines Pixels liefert. Die Beleuchtung eines Pixels mit farbigem Licht wird dann diesen Farbwert aufhellen bzw. abdunkeln und gemäß der Lichtfarbe einfärben.

Lichtquellen

Auch in unserer Engine möchten wir natürlich eine vernünftige Beleuchtung verwenden. Nichts kann dem Auge eine realistischere Szene vortäuschen als eine gute Beleuchtung mit Schatteneffekten. Selbst ohne Texturen zu verwenden, kann man durch eine entsprechend ausgeleuchtete Geometrie einen sehr realistischen Eindruck erzeugen. Um eine solche Beleuchtung durchführen zu können, benötigen wir aber auf alle Fälle Lichtquellen. Dafür definieren wir die folgenden Typen von Quellen direkten Lichts:

```
typedef enum ZFXLIGHTID_TYPE {
    LGT_DIRECTIONAL,
    LGT_POINT,
    LGT_SPOT;
} ZFXLIGHTID;
```

**Struktur**

Nun benötigen wir natürlich eine Struktur, in der man die verschiedenen Werte einer Lichtquelle definieren kann. Vieles davon haben wir bereits weiter oben angesprochen, insbesondere, welche Angaben man für welche Art von Lichtquelle benötigt. Für gerichtetes Licht benötigt man beispielsweise nur die Richtung und die Farben des Lichts. Abbildung 5.4 zeigt den Lichtwurf einer Spotlicht-Quelle. In Abbildung 5.2 kann man den weichen Übergang des äußeren Kegels, der recht klein gehalten ist, noch gut erkennen.

```
typedef struct ZFXLIGHT_TYPE {
    ZFXLIGHTID Type;           // Art der Lichtquelle
    ZFXCOLOR   cDiffuse;       // RGBA diffuse light value
    ZFXCOLOR   cSpecular;      // RGBA specular light value
    ZFXCOLOR   cAmbient;       // RGBA ambient light value
    ZFXVector  vcPosition;     // Position der Quelle
    ZFXVector  vcDirection;    // Richtung des Lichts
    float      fRange;         // Reichweite
    float      fTheta;         // Winkel des inneren Kegels
    float      fPhi;           // Winkel des äußeren Kegels
    float      fAttenuation0;   // Abschwächung
    float      fAttenuation1;  // Abschwächung
} ZFXLIGHT;
```

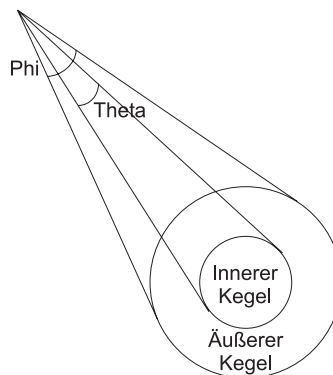


Abbildung 5.4:
Die beiden Winkel
und Lichtkegel
eines Spotlights

Damit sind wir nun endlich bei der Struktur angekommen, die wir für unser Material verwenden. Diese enthält vier Farbwerte, nämlich einen für ambientes, einen für diffuses, einen für spekuläres und einen für emittiertes Licht. Dabei sind diese Farbwerte als Prozentangaben zu verstehen, wie viel des in der 3D-Szene vorhandenen Lichtes dieses Material reflektiert. Enthält die Szene beispielsweise 50% ambientes Licht und hat ein Material einen Wert

Material Struktur

von 50% für ambientes Licht, so ist das Objekt nur so hell, als ob es von 25 % ambientem Licht beleuchtet würde, denn die Hälfte des einfallenden ambienten Lichtes verschluckt das Material. Dass ein Material auch Licht emittieren kann, hatte ich bisher noch nicht ausdrücklich erwähnt. Man denke hier nur an lustig leuchtende Uranstäbe oder Glühwürmchen. Selbst wenn in einer 3D-Szene gar kein Licht bzw. keine Lichtquelle vorhanden ist, so kann ein Objekt dennoch leuchten. Aber Vorsicht: Dieser Farbwert geht nur in die Berechnung für das Objekt selbst mit ein und trägt nichts zum Licht der 3D-Szene bei. In einem dunklen Raum ohne Lichtquellen würde ein solches Material zwar das Objekt selbst zum Leuchten bringen und erhellen, nicht jedoch andere Objekte in der Szene. Ist dies gewünscht, muss man noch eine eigene, echte Lichtquelle für dieses Objekt hinzufügen.



```
typedef struct ZFXMATERIAL_TYPE {
    ZFXCOLOR cDiffuse; // RGBA diffuse light
    ZFXCOLOR cAmbient; // RGBA ambient light
    ZFXCOLOR cSpecular; // RGBA specular light
    ZFXCOLOR cEmissive; // RGBA emissive light
    float fPower; // Specular power
} ZFXMATERIAL;
```

Der `float`-Wert in dieser Struktur gibt noch die Stärke für das Spekularreflexionsmodell an, also wie stark eine spiegelnde Oberfläche einfallendes Licht reflektiert.



Es ist zu beachten, dass man in der Computergrafik mit der Bezeichnung Lichtquelle oder Light lediglich einen unsichtbaren Photonenemitter meint. Dieser dient als Basis für die Beleuchtung von 3D-Objekten in der Szene. Damit ist aber explizit verbunden, dass zwei bestimmte Dinge nicht geleistet werden: Zum einen hat so eine Lichtquelle kein eigenes 3D-Modell. Platziert man in einem Raum beispielsweise eine Lichtquelle an der Decke, so werden zwar alle Objekte in dem Raum entsprechend beleuchtet, der Betrachter kann aber nicht erkennen, wo die Lichtquelle sitzt. Hier muss man zusätzlich das 3D-Modell einer Deckenlampe entsprechend platzieren.

Zum anderen dienen die virtuellen Photonen wirklich nur zur Beleuchtung von Oberflächen und sind selbst nicht sichtbar. Selbst wenn man direkt in eine Lichtquelle im virtuellen Universum blickt, so wird man nicht geblendet, und man sieht auch nicht die Lichtstrahlen, die von der Lichtquelle ausgehen. Diese Effekte muss man selbst zusätzlich programmieren.

Damit haben wir den Bereich Lichtfarbe und Material hinreichend abgedeckt. Jetzt benötigen wir noch eine Struktur, in der wir Texturen speichern können. Nun möchten wir ja mit verschiedenen APIs arbeiten, und OpenGL und Direct3D verwenden jeweils andere Objekte für Texturen. Unter Direct3D muss man sich ein eigenes `IDirect3DTexture`-Interface-Objekt erzeugen, während es bei OpenGL ausreicht, die Pixeldaten der Grafik anzugeben. Daher definieren wir die Struktur wie folgt:


```
typedef struct ZFXTEXTURE_TYPE {  
    float fAlpha;  
    char *chName;  
    void *pData;  
} ZFXTEXTURE;
```



Diese Struktur enthält einmal einen Pointer, unter dem wir den Namen der Textur abspeichern werden. Das ist der Name der zu ladenden *.bmp-Grafikdatei. In dem zweiten Pointer vom Typ `void` speichern wir dann die notwendigen Daten. In unserem Fall wird das ein Pointer auf ein `LPDIRECT3DTEXTURE9`-Objekt sein. Ein OpenGL-Renderer würde hier einen Pointer auf die Pixeldaten ablegen.

Zu guter Letzt benötigen wir noch eine Struktur für ein Skin-Objekt. Immer wenn ein Polygon gerendert wird, müssen wir ein Material und keine, eine oder mehrere Texturen angeben. Beim Multitexturing kann man mehrere Texturen auf dasselbe Polygon legen und diese beliebig miteinander *blenden*, also deren Farbwerte zusammenrechnen. Theoretisch kann man bis zu acht Texturen gleichzeitig auf ein Objekt legen, wobei heutige Grafikkarten in der Regel nur die simultane Verwendung von bis zu vier Texturen gleichzeitig in einem Singlepass zulassen.

Als Singlepass-Rendering bezeichnet man die Möglichkeit einer Grafikkarte, in einem einzigen Rendervorgang ein Polygon mit mehreren Texturen gleichzeitig zu rendern. Ältere Grafikkarten können dies nicht. Um hier mehrere Texturen auf ein- und dasselbe Polygon zu legen, ist je Textur ein separater Rendervorgang mit verschiedenen Parametereinstellungen notwendig. Dies bezeichnet man als Multipass-Rendering. Verständlicherweise ist es langsamer als das Singlepass-Rendering.



In unserer `ZFXSKIN`-Struktur müssen wir also drei Dinge speichern: zum einen das verwendete Material, dann die verwendeten Texturen und natürlich noch einen `bool`-Wert, der angibt, ob Alphablending von dieser Skin verwendet wird. Aus Letzterem können wir dann beim Aktivieren der Skin sehen, ob wir die Renderstates für das Alphablending aktivieren müssen.

```
typedef struct ZFXSKIN_TYPE {  
    bool bAlpha;  
    UINT nMaterial;  
    UINT nTexture[8];  
} ZFXSKIN;
```



Wie ihr seht, speichern wir hier nur IDs auf das Material und die verwendete Textur. Das hat etwas mit dem Management zu tun. Wir werden später ein Array haben, in dem alle geladenen Texturen und Materialien stehen. Es kann ja durchaus sein, dass Materialien und Texturen in verschiedenen Kombinationen von verschiedenen Skins genutzt werden. Auch in diesem

Fall werden wir die entsprechenden Daten nur einmal laden und in der ZFX-SKIN-Struktur die entsprechenden Indices auf die wirklichen Daten in den entsprechenden Arrays angeben. Diese Arrays verpacken wir am besten in der Interface-Definition für unseren Skin-Manager.



Man sollte mit seiner Engine nie den Weg des geringsten Aufwandes gehen. Getreu dem Motto »Irgendwann muss ich sowieso bremsen, also fahr' ich gleich mit angezogener Handbremse ... « kommt man weder mit dem Auto noch mit einer 3D-Engine sehr weit. Beinahe jede zusätzliche Operation beim Rendern von 3D-Grafik kostet Zeit, so auch das Alphablending. Hier muss jeder zu rendernde Pixel zusätzlich daraufhin geprüft werden, ob er einen Alpha-Wert ungleich 1.0f hat. Wenn man von vornherein weiß, dass das nicht der Fall ist (dass eine Textur also kein Alphablending verwendet), sollte man das Alphablending gar nicht erst aktivieren, um sich diese zusätzlichen Tests zu sparen.

Das kostet uns im Programmcode bei jedem Texturswitch genau eine if-Abfrage, spart der Grafikkarte aber eine ähnliche Abfrage für jeden berechneten Pixel. Man sollte es also vermeiden, Device-States wie beispielsweise das Alphablending pauschal beim Programmstart zu aktivieren, nur weil man auch Texturen mit Transparenzeffekten nutzt.

Interface-Definition für einen Skin-Manager

*Interfaces als
Management-by-
Objective*

Eigentlich fahren wir mit unserem bisherigen Engine-Ansatz eine recht moderate Strategie, indem wir *Management-by-Objective* betreiben. Wir haben bisher nur ein Interface für einen Renderer. Wie dieser aber seine Funktionsweise implementiert, dazu geben wir keinerlei Rahmenrichtlinien vor, sondern nur das Objective (dt. Ziel): Er soll 3D-Grafik rendern können. Wie die untergeordneten Stellen ein solches vorgegebenes Ziel erreichen, ist sekundär und interessiert uns dabei nicht wirklich. Das Interface sieht bisher ja nur vor, dass man den Renderer initialisieren kann. Später wird der Renderer auch nur vorsehen, dass man Dreieckslisten rendern kann. Wie eine Implementierung des Interfaces das umsetzt, also beispielsweise mit OpenGL oder Direct3D, mit Primitiven-Listen oder Vertex-Buffern, das ist in der Interface-Definition nicht enthalten.

*Objective: Skin-
Management*

Diese Strategie ist aber nicht an jeder Stelle wirklich sinnvoll. Daher werden wir den Skin-Manager auch als Interface anlegen, dessen eigentliche Implementierung dann in der DLL zusammen mit dem Renderer steht. So zwingen wir denjenigen, der eine eigene Render-DLL für unsere Engine schreiben will, wenigstens ein gewisses Maß an Optimierung einzuhalten. Insbesondere bei Texturen hat man es mit vielen Daten zu tun, die nicht redundant geladen werden sollten. Durch unser Interface können wir dies, anders als bei einer teilweise implementierten Basisklasse, zwar nicht ausschließen, aber durch die Interface-Definition können wir immerhin aufzeigen, was

eine Implementierung des Interfaces leisten können sollte. Wir geben also auch für den Skin-Manager ein Interface an und formulieren dadurch für den Programmierer das Ziel, ein vernünftiges Management für Materialien und Texturen in seiner DLL einzuführen.

```
class ZFXSkinManager {
protected:
    UINT          m_nNumSkins;           // Anz. Skins
    UINT          m_nNumMaterials;       // Anz. Materialien
    UINT          m_nNumTextures;        // Anz. Texturen
    ZFXSKIN       *m_pSkins;             // mem für skins
    ZFXMATERIAL   *m_pMaterials;         // mem für mats
    ZFXTEXTURE    *m_pTextures;         // mem für textrn

public:
    ZFXSkinManager(void) {};
    virtual ~ZFXSkinManager(void) {};

    virtual HRESULT AddSkin(const ZFXCOLOR *pcAmbient,
                           const ZFXCOLOR *pcDiffuse,
                           const ZFXCOLOR *pcEmissive,
                           const ZFXCOLOR *pcSpecular,
                           float fSpecPower,
                           UINT *nSkinID)=0;

    virtual HRESULT AddTexture(UINT nSkinID,
                               const char *chName,
                               bool bAlpha,
                               float fAlpha,
                               ZFXCOLOR *cColorKeys,
                               DWORD dwNumColorKeys)
        =0;

    virtual bool    MaterialEqual(
        const ZFXMATERIAL *pMat0,
        const ZFXMATERIAL *pMat1)=0;

    virtual ZFXSKIN    GetSkin(UINT nSkinID)=0;

    virtual ZFXMATERIAL GetMaterial(UINT nMatID)=0;

    virtual const char* GetTextureName(UINT nTexID,
                                         float *pfAlpha,
                                         ZFXCOLOR *pAK,
                                         UCHAR *pNum)=0;

    virtual void    LogCurrentStatus(char *chLog,
                                     bool bDetail)=0;
};
```



Skins hinzufügen

Wie man es von einem Interface erwartet, ist auch dieses recht einfach gehaltener. Es gibt hier lediglich drei Pointer für drei Arrays und entsprechende Zähler für den Füllstand der jeweiligen Arrays. Hier sollen geladene Materialien, Texturen und sich daraus zusammensetzende Skins gespeichert werden. Neben dem Konstruktor und dem Destruktor verfügt das Interface eigentlich nur über zwei wichtige Funktionen. Über `ZFXSkinManager::AddSkin` lässt sich ein neues Skin-Objekt anlegen. Dazu muss man zwangsläufig alle Daten angeben, die für ein Material benötigt werden, denn eine Skin benötigt immer ein Material. Im Gegensatz dazu ist es aber nicht nötig, eine Textur anzugeben. Schließlich kann es durchaus sein, dass wir auch einmal etwas ohne Textur rendern wollen. Belohnt wird man von der Methode dann durch eine ID, die man im letzten Parameter erhält. Diese ID identifiziert das neue Material eindeutig, und man sollte sie sich gut merken – wenn man später etwas mit dieser Skin rendern möchte, dann muss man die entsprechende ID angeben.

Texturen hinzufügen

Die Methode `ZFXSkinManager::AddTexture` besitzt bereits eine kompliziertere Parameterliste. Sie dient dazu, einer bereits vorhandenen Skin eine Textur hinzuzufügen. Dazu muss man natürlich die ID der Skin parat haben. Als zweiten Parameter gibt man den Namen der Grafikdatei an, die als Textur verwendet werden soll.⁶ Die folgenden vier Parameter dienen nun ausschließlich dem Alphablending. Möchte man Transparenzeffekte auf dieser Textur verwenden, so muss man für den `bool`-Parameter `true` angeben. Nun gibt es aber zwei verschiedene Arten von Transparenzeffekten: zum einen den so genannten Color Key und zum anderen eine allgemeine Transparenz für die gesamte Textur. Das erläutern wir bei der eigentlichen Implementierung noch genauer. Ein Color Key bezeichnet aber eine bestimmte Farbe der Textur, beispielsweise volles Rot, die wir mit einer bestimmten Transparenz versehen wollen. Sollen alle roten Pixel der Grafik beispielsweise 50% Transparenz erhalten, würden wir als Color Key `{ 1.0f, 0.0f, 0.0f, 0.5f }` als `ZFXCOLOR` angeben. Die Funktion ist aber flexibel genug, dass man beliebig viele solcher Color Keys definieren kann. Die #Anzahl der definierten Keys steht dann im letzten Parameter. Der `float`-Wert der Parameterliste hingegen bezeichnet den Transparenzgrad, den die gesamte Textur (also jeder Pixel, egal welche Farbe er hat) erhalten soll. Dies kann auch 1.0f sein, falls diese Option der Transparenz nicht gewünscht wird.

MISC-Funktionen

Unsere Interface-Definition enthält aber noch zwei andere Funktionen, die nicht von zentraler Bedeutung sind, sondern eher ein nützliches Feature. Zum einen kann man über eine öffentliche Funktion zwei Material-Objekte auf Gleichheit prüfen. Der Manager benötigt dies auf alle Fälle, um zu sehen, ob er ein Material bereits geladen hat. Aber falls der Anwender unserer DLL auch einmal Materialien vergleichen möchte, so kann er diese Funktion gleich mit nutzen. Die zweite Funktion soll ein Log-File erzeugen, in das der Manager seine Daten pumpt. Dies ist zum Debuggen für uns ganz sinnvoll, denn hier können wir uns alle geladenen Materialien und Texturen im Manager anzeigen lassen.

⁶ Wir arbeiten hier nur mit 24-Bit-BMP-Dateien.

Jetzt haben wir also eine Interface-Definition für einen Skin-Manager. Diese Interface-Definition schreiben wir sogleich in die passende Datei `ZFXRenderDevice.h` und fügen dann dem Render-Device-Interface `ZFXRenderDevice` ein entsprechendes Attribut `ZFXSkinManger *m_pSkinMan` hinzu. Wer jetzt einen Renderer für die *ZFXEngine* implementieren möchte, der ist damit gezwungen, auch einen Skin-Manager vorzusehen.

*Manager als
Member*

In diesem Kapitel implementieren wir einen Skin-Manager in den beiden Dateien `ZFXD3D_skinman.h` und `ZFXD3D_skinman.cpp`, die wir im nächsten Kapitel zu unserem DLL-Projekt `ZFXD3D` hinzufügen werden. Am Ende dieses Kapitels steht zwar die fertige Implementierung dieser Klasse, aber noch kein lauffähiges Testprojekt, da unser Render-Device ja bisher noch gar nichts rendern kann.



Skin-Manager des Direct3D-Renders

Nun kommen wir schon zur tatsächlichen Implementierung eines Skin-Managers, und zwar für unser `ZFXD3D-Render-Device`. Entsprechend nennen wir unsere Klasse `ZFXD3DSkinManager`. Sehen wir uns erst mal die Definition dieser Klasse an, und stören wir uns dabei nicht an `ZFXD3DVCache`. Dies ist eine Klasse, die auch auf die Attribute des Skin-Mangers zugreifen darf. Diese Klasse lernen wir jedoch erst im nächsten Kapitel kennen.

```
class ZFXD3DSkinManager : public ZFXSkinManager {

    friend class ZFXD3DVCache;

public:
    ZFXD3DSkinManager(LPDIRECT3DDEVICE9 pDevice,
                      FILE *pLog);
    ~ZFXD3DSkinManager(void);

    HRESULT AddSkin(const ZFXCOLOR *pcAmbient,
                   const ZFXCOLOR *pcDiffuse,
                   const ZFXCOLOR *pcEmissive,
                   const ZFXCOLOR *pcSpecular,
                   float fSpecPower, UINT *nSkinID);

    HRESULT AddTexture(UINT nSkinID,
                      const char *chName,
                      bool bAlpha, float fAlpha,
                      ZFXCOLOR *cColorKeys,
                      DWORD dwNumColorKeys);

    bool MaterialEqual(const ZFXMATERIAL *pMat0,
                      const ZFXMATERIAL *pMat1);

    ZFXSKIN    GetSkin(UINT nSkinID);
```



```

ZFXMATERIAL GetMaterial(UINT nMatID);
const char* GetTextureName(UINT nTexID);

void LogCurrentStatus(char *chLog,
                     bool bDetailed);

protected:
    LPDIRECT3DDEVICE9 m_pDevice;
    FILE *m_pLog;

    inline bool ColorEqual(const ZFXCOLOR *pCo10,
                          const ZFXCOLOR *pCo11);

    HRESULT CreateTexture(ZFXTEXTURE *pTexture,
                        bool bAlpha);

    HRESULT SetAlphaKey(LPDIRECT3DTEXTURE9 *ppTexture,
                      UCHAR R, UCHAR G,
                      UCHAR B, UCHAR A);

    HRESULT SetTransparency(LPDIRECT3DTEXTURE9
                          *ppTexture,
                          UCHAR Alpha);

    DWORD MakeD3DColor(UCHAR R, UCHAR G,
                      UCHAR B, UCHAR A);

    void Log(char *, ...);
};

```

Zunächst fällt auf, dass der Konstruktor unserer Klasse eine Parameterliste hat. Zum einen benötigen wir in dieser Klasse einen Pointer auf das `IDirect3DDevice9`-Objekt, und zum anderen können wir den Pointer auf ein File-Objekt angeben, das als Log-File von der `Log()`-Funktion dieser Klasse verwendet werden soll (siehe Exkurs).

Log-File für Ablaufprotokolle

Nichts ist für das Debugging eines Programms hilfreicher als ein gut organisiertes Log-File. Dafür gibt es mindestens zwei sinnvolle Verfahren. Zum einen kann man eine eigene Log-File-Klasse entwerfen. Diese Klasse wird dann in alle eigenen Projekte eingebunden. Der Nachteil dabei ist, dass man die Log-File Klasse immer und überall zugänglich haben muss. Ich habe mich hier für einen anderen Weg entschieden. Das Loggen von Daten in ein File ist keine sonderlich komplizierte Aufgabe, daher habe ich eine kleine Log-File-Funktion geschrieben, die in jeder wichtigen Klasse implementiert ist.

*Dazu müssen zwei Voraussetzungen erfüllt sein: Jede Klasse, beispielsweise das ZFXRenderDevice oder der ZFXSkinManager, die das Log-File nutzen will, muss neben der gleich folgenden Member-Funktion auch ein Attribut FILE *m_pLog bereitstellen. Dies ist entweder eine eigene Textdatei der Klasse oder ein dem Konstruktor übergebener Pointer auf eine bereits geöffnete Datei.*

Dazu benötigt die jeweilige Klasse Zugriff auf eine einheitliche globale Variable bool g_bLF. Dies steht für Log-Flush, und wenn die Variable den Wert true hat, dann soll jede protokollierte Ausgabe sofort aus dem Output-Stream in die Datei geflusht werden. So bleiben alle Log-File-Ausgaben auch bei einem Absturz des Programms erhalten. Die eigentliche Log-Funktion sieht dann wie folgt aus (die Bezeichnung ZFXKlasseXYZ steht für eine beliebige unserer Klassen):

```
void ZFXKlasseXYZ::Log(char *chString, ...) {
    char ch[256];
    char *pArgs;

    pArgs = (char*) &chString + sizeof(chString);
    vsprintf(ch, chString, pArgs);
    fprintf(m_pLog, "[ ZFXKlasseXYZ]: ");
    fprintf(m_pLog, ch);
    fprintf(m_pLog, "\n");

    if (g_bLF)
        fflush(m_pLog);
} // Log
```

Diese Log()-Funktion kann man entsprechend der Standard-Funktion fprintf() verwenden, ihr also einen Formatierungsstring und beliebig viele Parameter übergeben, die in dem String ausgegeben werden sollen. Den Pointer pArgs setzen wir auf den Anfang dieser Liste von Parametern für den Formatierungsstring. Die Funktion vsprintf() dient schließlich dazu, die Daten aus einer solchen Parameterliste anhand eines Formatierungsstrings als echten String zu konvertieren. Diesen kompletten String speichern wir dann in der Variablen ch.

Für die bessere Lesbarkeit des Protokolls geben wir dann zuerst aus, aus welcher Klasse die folgende Meldung kommt. Dann geben wir die Meldung aus und schließen mit einem Zeilenumbruchzeichen die Ausgabe ab. Soll direkt geflusht werden, dann tun wir das auch noch.

Diese Log-Funktion wird sich also des Öfteren in unserem Code finden. Insbesondere unsere Render-Device-DLL enthält eine eigene Textdatei als Log-File, die durch die Klasse ZFXD3D geöffnet und geschlossen wird. Alle anderen Klassen in der DLL erhalten im Konstruktor einen Pointer auf diese Log-Datei.

Neben der Log()-Funktion hat unser Skin-Manager noch fünf andere, nicht-öffentliche Methoden. Diese schauen wir uns im Verlauf dieses Kapitels alle einzeln an. Hier beginnen wir aber zunächst mit der Implementierung des Konstruktors der Klasse.



```
ZFXD3DSkinManager::ZFXD3DSkinManager(LPDIRECT3DDEVICE9
                                     pDevice,
                                     FILE *pLog) {
    m_nNumMaterials = 0;
    m_nNumTextures  = 0;
    m_nNumSkins     = 0;
    m_pMaterials    = NULL;
    m_pTextures     = NULL;
    m_pSkins        = NULL;
    m_pLog           = pLog;
    m_pDevice        = pDevice;
    Log("online");
}
```

Initialisierung

Wie man sieht, passiert hier nichts Aufregendes. Die Zähler der Klasse werden auf 0 gesetzt und die entsprechenden Pointer ausgenullt. Den Pointer auf das Log-File speichern wir ebenso in den Klassen-Attributen wie den Zeiger auf das Direct3D-Device. Damit ist unser Skin-Manager auch schon betriebsbereit. Quasi als Ausgleich ist aber der Destruktor um einiges umfangreicher.



```
ZFXD3DSkinManager::~ZFXD3DSkinManager(void) {
    // Direct3D-Textur-Objekte freigeben
    for (UINT i=0; i<m_nNumTextures; i++) {
        if (m_pTextures[i].pData) {
            ((LPDIRECT3DTEXTURE9)(m_pTextures[i].pData))
                ->Release();
            m_pTextures[i].pData = NULL;
        }
        if (m_pTextures[i].pClrKeys) {
            delete [] m_pTextures[i].pClrKeys;
            m_pTextures[i].pClrKeys = NULL;
        }
        if (m_pTextures[i].chName) {
            delete [] m_pTextures[i].chName;
            m_pTextures[i].chName = NULL;
        }
    }
}
```



```

    }
}
// Speicher freigeben
if (m_pMaterials) {
    free(m_pMaterials);
    m_pMaterials = NULL;
}
if (m_pTextures) {
    free(m_pTextures);
    m_pTextures = NULL;
}
if (m_pSkins) {
    free(m_pSkins);
    m_pSkins = NULL;
}
Log("offline (ok)");
}

```

Im Destruktor müssen wir natürlich sämtliche Direct3D-Objekte wieder freigeben, die wir zuvor durch die Instanz unserer Klasse angefordert haben. Hier sind dies die LPDIRECT3DTEXTURE9-Texturobjekte im entsprechend gecasteten Pointer der ZFXTEXTURE-Struktur. Ebenfalls in dieser Struktur müssen wir den Speicher freigeben, den wir beim Laden für den Namen der Textur allokiert haben. Dann haben wir noch die drei Attribute für die Material-, Textur- und Skin-Arrays, deren Speicher wir auch freigeben müssen.

Freigabe

Damit können wir uns im Folgenden die Implementierung der Funktionalität der Klasse anschauen.

Farben und Materialien vergleichen

Eine der wichtigsten Aufgaben unseres Managers ist es ja, bereits geladene oder erzeugte Daten mit denen zu vergleichen, die als Nächstes geladen oder erzeugt werden sollen. So kann man Redundanz in der Belegung von RAM und VRAM verhindern. Dazu benötigen wir in unserer Skin-Manager-Klasse zwei Funktionen, mit denen wir Farben und Materialien auf Gleichheit prüfen können. Dazu vergleichen wir ganz einfach einzeln alle Elemente der jeweiligen Strukturen.

Ohne Worte

```

inline bool ZFXD3DSkinManager::ColorEqual(
    const ZFXCOLOR *pCo10,
    const ZFXCOLOR *pCo11)
{
    if ( (pCo10->fA != pCo11->fA) ||
        (pCo10->fR != pCo11->fR) ||
        (pCo10->fG != pCo11->fG) ||

```



```

        (pCol0->fB != pCol1->fB) )
    return false;
    return true;
} // ColorEqual
/*-----*/

bool ZFXD3DSkinManager::MaterialEqual(
    const ZFXMATERIAL *pMat0,
    const ZFXMATERIAL *pMat1)
{
    if (!ColorEqual(&pMat0->cAmbient, &pMat1->cAmbient)
        || !ColorEqual(&pMat0->cDiffuse, &pMat1->cDiffuse)
        || !ColorEqual(&pMat0->cEmissive, &pMat1->cEmissive)
        || !ColorEqual(&pMat0->cSpecular, &pMat1->cSpecular)
        || (pMat0->fPower != pMat1->fPower) )
        return false;
    return true;
} // MaterialEqual
/*-----*/

```

Mit diesen Hilfsfunktionen können wir uns daran machen, neue Materialien und Skins zu dem Manager hinzuzufügen. Zuvor benötigen wir aber noch eine klitzekleine Funktionalität.

Skins austeilen

*Skins aus dem
Inventar holen*

Der Manager muss auch dazu in der Lage sein, auf Wunsch ein Skin-Objekt zu einer ID auszuteilen. Vor dem Rendern kommt sozusagen jedes Objekt zu dem Skin-Manager und holt sich dort seine Skin für den Rendervorgang ab.



```

ZFXSKIN ZFXD3DSkinManager::GetSkin(UINT nSkinID) {
    ZFXSKIN skin;

    if (nSkinID < m_nNumSkins)
        memcpy(&skin, &m_pSkins[nSkinID],
            sizeof(ZFXSKIN));
    return skin;
} // GetSkin

```

Mit dem so erhaltenen Skin-Objekt kann der Anwender dann alles tun, was er möchte, da er ja nur eine Kopie erhalten hat. Diese kann er dazu verwenden, bestimmte Materialwerte zu prüfen, zu verrechnen oder das Material und seine Texturen beim Rendern für das Device zu aktivieren. Aber dazu kommen wir erst im nächsten Kapitel. Viel später benötigen wir auch die Funktionalität, dass ein Anwender sich das tatsächliche Material und den Namen und weitere Daten der Textur ausgeben lassen kann, beispielsweise dann, wenn wir in einem 3D-Editor Daten speichern wollen. Daher zeige ich euch hier noch schnell die folgenden analogen Methoden:



```

ZFXMATERIAL ZFXD3DSkinManager::GetMaterial(UINT nMatID)
{
    ZFXMATERIAL mat;
    if (nMatID < m_nNumMaterials)
        memcpy(&mat, &m_pMaterials[nMatID],
               sizeof(ZFXMATERIAL));
    return mat;
} // GetMaterial
/*-----*/

const char* ZFXD3DSkinManager::GetTextureName(UINT nID,
                                                float *pfAlpha,
                                                ZFXCOLOR *pAK,
                                                UCHAR *pNum)
{
    if (nID >= m_nNumTextures) return NULL;
    if (pfAlpha) *pfAlpha = m_pTextures[nID].fAlpha;
    if (pNum) *pNum = m_pTextures[nID].dwNum;

    if (m_pTextures[nID].pClrKeys && pAK) {
        memcpy(pAK, m_pTextures[nID].pClrKeys,
               sizeof(ZFXCOLOR) * m_pTextures[nID].dwNum);
    }

    return m_pTextures[nID].chName;
} // GetTextureName

```

5.3 Skins und Materialien aufnehmen

Wenn wir nun ein neues Skin-Objekt anlegen wollen, so geben wir auf alle Fälle an, welches Material dieses Skin-Objekt verwenden soll. Dann haben wir zwei Dinge zu tun: Zuerst müssen wir einen freien Platz im Array `m_pSkins` der Klasse finden. Am Anfang enthält dieser Pointer ja noch gar nichts, daher müssen wir Speicher dynamisch allokalieren. Das machen wir immer in Fünfziger-Schritten. Sobald also der Zählerstand dieses Arrays glatt durch fünfzig zu teilen ist, allokalieren wir fünfzig zusätzliche Einheiten Speicher für das Array über den `realloc()`-Befehl. Als Zweites müssen wir dann prüfen, ob das Material, das das neue Skin-Objekt verwenden soll, bereits existiert. Dazu durchlaufen wir das Array mit den Materialien, also `m_pMaterials`.

Es gibt viel zu tun ...

Jetzt verzweigen wir in zwei Fälle. Entweder finden wir in dem Array bereits ein entsprechendes Material. Dann speichern wir dessen ID in dem neuen Skin-Objekt. Finden wir dort aber kein gleiches Material, dann müssen wir das Material-Array um ein neues Material-Objekt mit diesen Eigenschaften erweitern. Auch hier prüfen wir zunächst die verfügbaren freien Plätze im Array und allokalieren falls nötig fünfzig Speicherplätze für neue Materialien.

... packen wir es an!

Dann kopieren wir die Eigenschaften des Materials in dieses Objekt und speichern in dem Skin-Objekt die ID des neuen Materials. Damit haben wir ein neues Skin-Objekt angelegt.



```
#define MAX_ID 65535

HRESULT ZFXD3DSkinManager::AddSkin(
    const ZFXCOLOR *pcAmbient,
    const ZFXCOLOR *pcDiffuse,
    const ZFXCOLOR *pcEmissive,
    const ZFXCOLOR *pcSpecular,
    float fSpecPower,
    UINT *nSkinID) {

    UINT    nMat, n;
    bool    bMat=false;

    // Speicher für 50 Elemente allokalieren, wenn nötig
    if ( (m_nNumSkins%50) == 0 ) {
        n = (m_nNumSkins+50)*sizeof(ZFXSKIN);
        m_pSkins = (ZFXSKIN*)realloc(m_pSkins, n);
        if (!m_pSkins) return ZFX_OUTOFMEMORY;
    }

    ZFXMATERIAL mat;
    mat.cAmbient  = *pcAmbient;
    mat.cDiffuse  = *pcDiffuse;
    mat.cEmissive = *pcEmissive;
    mat.cSpecular = *pcSpecular;
    mat.fPower    = fSpecPower;

    // Haben wir bereits so ein Material?
    for (nMat=0; nMat<m_nNumMaterials; nMat++) {
        if ( MaterialEqual(&mat, &m_pMaterials[nMat]) ) {
            bMat = true;
            break;
        }
    } // for [MATERIALS]

    // Falls existiert, ID speichern, sonst neu anlegen
    if (bMat) m_pSkins[m_nNumSkins].nMaterial = nMat;
    else {
        m_pSkins[m_nNumSkins].nMaterial = m_nNumMaterials;

        // Speicher für 50 Elemente allokalieren, wenn nötig
        if ( (m_nNumMaterials%50) == 0 ) {
            n = (m_nNumMaterials+50)*sizeof(ZFXMATERIAL);
            m_pMaterials = (ZFXMATERIAL*)realloc(
                m_pMaterials, n);
            if (!m_pMaterials) return ZFX_OUTOFMEMORY;
        }
    }
}
```

```

memcpy(&m_pMaterials[m_nNumMaterials], &mat,
       sizeof(ZFXMATERIAL));
m_nNumMaterials++;
}

m_pSkins[m_nNumSkins].bAlpha = false;
for (int i=0; i<8; i++)
    m_pSkins[m_nNumSkins].nTexture[i] = MAX_ID;

// speichere ID, und inkrementiere
(*nSkinID) = m_nNumSkins;
m_nNumSkins++;

return ZFX_OK;
} // AddSkin

```

Die IDs der jeweiligen Objekte sind natürlich einfach nur die Indexpositionen der Objekte im Array. Das nullte Objekt im Array hat die ID 0, das erste Objekt im Array hat die ID 1 und so weiter. Nun sehen wir auch ein, warum wir die Materialien und Texturen nicht direkt in der `ZFXSKIN`-Struktur speichern. Wir können nun beliebig viele Skin-Objekte anlegen, ohne Materialien und Texturen mehr als nur ein einziges Mal im Speicher haben zu müssen.

ID Vergabe

Weiterhin bemerkenswert ist zum einen die Einstellung des `bool`-Elements `bAlpha` auf `false` als Default-Wert. Zum anderen setzen wir die ID aller acht möglichen Texturen einer Skin auf den Wert `MAX_ID`. Dies ist der maximale Wert, den eine 16-Bit-Variable aufnehmen kann, und sagt in unserem Kontext aus, dass bisher keine gültige ID vergeben ist.

5.4 Texturen aufnehmen

Jetzt haben wir bereits die Möglichkeit, Skins und Materialien von unserem Manager verwalten zu lassen. Jede Skin muss dabei genau ein Material haben. Jetzt können wir jeder Skin aber optional noch bis zu acht Texturen-IDs zuweisen. Wenn wir beispielsweise ein 3D-Modell eines Aliens oder Monsters laden, so wird ein solches Modell in der Regel eine Textur verwenden, die die Arme, Beine und den restlichen Körper des Viehs in grün-schleimiger Haut zeigt. Dazu laden wir dann eine Bump-Map um das Modell realistisch zu beleuchten. In diesem Fall bräuchten wir also eine Skin für das Modell (vorausgesetzt, alle Polygone des Modells verwenden das selbe Material) mit zwei Texturen. Also werden wir jetzt ...

*Skins und
Texturen*

Texturen zu den Skins hinzufügen

Texture Stages

Das funktioniert fast genauso wie das Hinzufügen eines neuen Materials zu einer Skin. Die ellenlange Parameterliste der entsprechenden Funktion `ZFX-SkinManager::AddTexture` haben wir ja weiter oben schon besprochen. Zuerst müssen wir die ID der Skin wissen, zu der die neue Textur gehören soll. Man kann einer Skin ja bis zu acht verschiedene Texturen zuweisen. Dazu würde man die Funktion `AddTexture()` entsprechend achtmal aufrufen. Die Reihenfolge der Aufrufe entscheidet dann darüber, in welcher Texture Stage die Textur beim Rendern stehen wird. Die zuerst zugewiesene Textur ist für die 0. Stage, die nächste für die 1. Stage und immer so weiter.



Beim Multipass- oder Singlepass-Rendering stellt man alle verwendeten Texturen für jeweils eine andere so genannte Texture Stage ein. Aktuelle Grafikkarten bieten die Möglichkeit, bis zu acht Texture Stages mit den IDs 0, ..., 7 zu verwenden.

Security-Check

Wenn wir nun eine Textur zu einem Skin-Objekt hinzufügen wollen, so müssen wir zunächst zwei Security-Checks durchführen: Zum einen muss es die angegebene SkinID wirklich geben, und zum anderen darf das entsprechende Skin-Objekt bisher nur maximal Referenzen auf bis zu sieben Textur-IDs beinhalten – sonst sind bereits alle acht Texturen für das Objekt gesetzt. Einen freien Slot für eine neue Textur-ID markieren wir ja durch den Wert `MAX_ID`.

Weiterer Ablauf

Sind diese beiden Checks überstanden, dann kann der Manager seine eigentliche Arbeit aufnehmen. Er muss nun in sein Inventar schauen und nachprüfen, ob die entsprechende Textur bereits in den Speicher geladen wurde. Das prüfen wir, indem wir alle Texturen im Array `m_pTextures` durchlaufen und deren Bitmap-Namen mit dem Namen der gerade hinzuzufügenden Textur vergleichen. Jetzt verzweigen wir in zwei Fälle. Wenn der Manager die entsprechende Textur bereits im Inventar hat, dann speichern wir die Inventarnummer (ID bzw. Indexwert) für das entsprechende Skin-Objekt. Dazu müssen wir in einer weiteren Schleife den ersten freien der acht möglichen Slots suchen.

Neue Grafik laden

Im anderen Fall der Verzweigung, wenn wir noch keine Textur mit identischem Namen im Inventar haben, müssen wir die Grafik zuerst noch in das Inventar laden und ihr eine neue ID zuweisen. Auch hier kontrollieren wir den verfügbaren Speicher des Arrays `m_pTextures`. Ist dieser nicht mehr ausreichend, allokatieren wir Platz für fünfzig neue Objekte des Typs `ZFXTEXTURE`. Dann speichern wir den Namen der zu ladenden Grafik für das neue Textur-Objekt ab und laden die entsprechende Grafik von der Festplatte. Zum Laden der Grafik verwenden wir die Funktion `ZFXD3DSkinManager::CreateTexture`. Nun ist nur noch eine Position offen: Falls diese Textur Alphablending verwenden soll, müssen wir noch die entsprechenden beiden Funktionen

SetAlphaKey() und SetTransparency() aufrufen. SetAlphaKey() steht dabei in einer Schleife, da wir beliebig viele Key-Farbwerte zulassen. Die komplette Funktion sieht wie folgt aus:

```
HRESULT ZFXD3DSkinManager::AddTexture(
    UINT        nSkinID,
    const char *chName,
    bool        bAlpha,
    float       fAlpha,
    ZFXCOLOR    *cColorKeys,
    DWORD       dwNumColorKeys)
{
    ZFXTEXTURE *pZFXTex=NULL; // Hilfszeiger
    HRESULT hr;
    UINT    nTex, n;
    bool    bTex=false;

    // Ist die SkinID überhaupt gültig?
    if (nSkinID >= m_nNumSkins) return ZFX_INVALIDID;

    // Sind bereits alle 8 Texturen der Skin belegt?
    if (m_pSkins[nSkinID].nTexture[7] != MAX_ID)
        return ZFX_BUFFER_SIZE;

    // Ist diese Textur schon geladen?
    for (nTex=0; nTex<m_nNumTextures; nTex++) {
        if (strcmp(chName, m_pTextures[nTex].chName)==0) {
            bTex = true;
            break;
        }
    } // for [TEXTURES]

    // lade neue Textur, falls erforderlich
    if (!bTex) {
        // Speicher für 50 Elemente allokalieren, wenn nötig
        if ( ( m_nNumTextures%50 ) == 0 ) {
            n = (m_nNumTextures+50)*sizeof(ZFXTEXTURE);
            m_pTextures = (ZFXTEXTURE*)realloc(
                m_pTextures, n);
            if (!m_pTextures) return ZFX_OUTOFMEMORY;
        }

        // Alphablending nötig?
        if (bAlpha) m_pSkins[nSkinID].bAlpha = true;
        else m_pTextures[m_nNumTextures].fAlpha = 1.0f;

        m_pTextures[m_nNumTextures].pClrKeys = NULL;

        // Speichere Texturnamen ab
        m_pTextures[m_nNumTextures].chName =
```



```

        new char[strlen(chName)+1];
memcpy(m_pTextures[m_nNumTextures].chName, chName,
       strlen(chName)+1);

// erzeuge neues Direct3D-Textur-Objekt
hr = CreateTexture(&m_pTextures[m_nNumTextures],
                  bAlpha);
if (FAILED(hr)) return hr;

// AlphaChannel zufügen, falls nötig
if (bAlpha) {
    pZFXTex = &m_pTextures[m_nNumTextures];

    // Infos merken
    pZFXTex->dwNum = dwNumColorKeys;
    pZFXTex->pClrKeys = new
        ZFXCOLOR[dwNumColorKeys];
    memcpy(pZFXTex->pClrKeys, cColorKeys,
           sizeof(ZFXCOLOR)*pZFXTex->dwNum);

    LPDIRECT3DTEXTURE9 pTex = (LPDIRECT3DTEXTURE9)
        pZFXTex->pData;

    // zuerst die AlphaKeys abarbeiten
    for (DWORD dw=0; dw<dwNumColorKeys; dw++) {
        hr = SetAlphaKey(&pTex,
                        UCHAR(cColorKeys[dw].fR*255),
                        UCHAR(cColorKeys[dw].fG*255),
                        UCHAR(cColorKeys[dw].fB*255),
                        UCHAR(cColorKeys[dw].fA*255));
        if (FAILED(hr)) return hr;
    }

    if (fAlpha < 1.0f) {
        // dann allgemeine Transparenz
        m_pTextures[m_nNumTextures].fAlpha=fAlpha;
        hr = SetTransparency(&pTex,
                            UCHAR(fAlpha*255));
        if (FAILED(hr)) return hr;
    }
}

// speichere ID, und inkrementiere
nTex = m_nNumTextures;
m_nNumTextures++;
}

// speichere ID im ersten freien Textur-Slot der Skin
for (int i=0; i<8; i++) {
    if (m_pSkins[nSkinID].nTexture[i] == MAX_ID) {

```



```

        m_pSkins[nSkinID].nTexture[i] = nTex;
        break;
    }
}
return ZFX_OK;
} // AddTexture

```

Man beachte hier vor allem, wie wir den `void`-Pointer der `ZFXTEXTURE`-Struktur auf den Typ `LPDIRECT3DTEXTURE9` casten. Des Weiteren ist es wichtig, dass wir im Falle von Transparenz zuerst die Alpha-Key-Werte für die Textur setzen und dann erst die allgemeine Transparenz. Die erstgenannte Funktion erwartet nämlich, dass ein entsprechender Farbwert, für den ein Alpha-wert gesetzt werden soll, keinen anderen Alphawert als 1.0f enthält.

Grafikdateien als Texturen laden

Unter Windows gibt es Dutzende verschiedene Möglichkeiten, Bitmap-Dateien zu laden und an die Pixeldaten heranzukommen. Früher habe ich immer komplett eigene Loader geschrieben, doch hier möchte ich die Arbeit ein wenig vereinfachen. Wir verwenden die Funktion `LoadImage()` der WinAPI, also des Programmierungs-Interfaces von Windows. Durch diese Funktion erhalten wir ein Handle auf die geöffnete Datei. Jetzt rufen wir die Funktion `GetObject()` der WinAPI zu Hilfe, die uns für dieses Handle eine so genannte *DIB-Sektion* erzeugen kann. Dabei steht DIB für **D**evice **I**ndependent **B**itmap (geräteunabhängige Bitmap) und bezeichnet ein Bildformat, das unabhängig von Grafikkarten und der Betriebssystem-Version ist.

*Bitmaps und
Windows*

Da wir uns hier aber schwerpunktmäßig über fortgeschrittene Themen der Spieleprogrammierung unterhalten wollen, setze ich voraus, dass jeder weiß, wie man eine Bitmap-Datei unter Windows laden und an die Pixeldaten kommen kann. Wer hier noch ein wenig Nachholbedarf hat, dem empfehle ich anhand der hier verwendeten WinAPI-Funktion und Strukturen in der MSDN (Hilfe in VC++ aufrufen!) die jeweiligen Prototypen und deren Parameter bzw. die Strukturen und deren Elemente zu studieren und so die Funktionsweise des Codes nachzuvollziehen.

Eigeninitiative

In der DIB-Sektion haben wir damit alle Daten der Bitmap-Datei griffbereit. Dazu gehört zum einen ein so genannter Header, also eine Struktur, die Informationen über das Bild speichert, insbesondere dessen Breite, Höhe und Bittiefe. Nun verwenden wir die Direct3D-Funktion `IDirect3DDevice9::CreateTexture`. Diese Funktion ist dazu da, uns ein Textur-Objekt von Direct3D zu erzeugen, das genug Platz bietet, um die Pixeldaten aufzunehmen. Ein solches Textur-Objekt vom Interface-Typ `IDirect3DTexture9` ist dabei eigentlich auch nicht viel mehr, als ein lineares Array für die Pixeldaten; aber wenn wir mit Direct3D eine Textur einstellen wollen, so geht das nur über ein Objekt dieses Interface-Typs. Wichtig ist dabei vor allem das Farbformat, das wir für die Textur verwenden.

*Texturen bei
Direct3D*

Farbformate

In unserer Engine lassen wir nur zwei Farbformate zu: zum einen `D3DFMT_R5G6B5` und zum anderen `D3DFMT_A8R8G8B8`. Dies sind beides Formatdefinitionen von Direct3D. `D3DFMT_R5G6B5` ist ein 16-Bit-Farbformat. Der Rot- und Blauanteil der Farbe belegt dabei je fünf Bits ($0-2^5 = 32$ Werte), der Grünanteil sechs Bits ($2^6 = 64$ Werte). Für einen Alphaanteil ist dabei kein Platz vorgesehen. Wir verwenden dieses Farbformat daher nur für solche Grafiken, die kein Alphablending benötigen. Das zweite Farbformat mit 32 Bit bietet für alle vier Farbanteile mit acht Bit je 256 verschiedene Werte ($2^8 = 256$). Oftmals wird heute in jeder Lebenslage bereits nach 32-Bit-Texturen geschrien. Man sollte dabei aber nie vergessen, dass eine höhere Bittiefe bei Texturen auch einen höheren Speicherverbrauch durch Texturen bedeutet, da jeder Pixel dann bei einem Vergleich von 16 und 32 Bit doppelt so viel Speicher belegt. In der Regel sind 16-Bit-Texturen schön genug, aber es sei jedem freigestellt, auch für Texturen ohne Alphablending ein entsprechendes 32-Bit-Format wie `D3DFMT_X8R8G8B8` zu verwenden.

Farbkonvertierung

Das geschulte Auge eines professionellen Mathematikers entdeckt natürlich sofort, dass wir keinerlei Probleme haben werden, die Daten aus einer 24-Bit-Bitmap-Datei in eine 32-Bit-Textur zu laden. Denn auch in diesem 32-Bit-Format werden ja genau 24 Bits für die RGB-Farbwerte verwendet. Wir können die Pixeldaten also eins zu eins aus der Datei übertragen. Die restlichen acht Bits jedes Pixels der Textur sind für den Alphawert reserviert. Wenn wir jedoch die 24-Bit-Pixeldaten aus der Bitmap in eine 16-Bit-Textur schreiben wollen, so geht dies nicht ohne Konvertierung der Farbwerte. Im 24-Bit-Fall haben die RGB-Anteile der Farbe ja jeweils acht Bit Platz, also Werte von 0-255. Im 16-Bit-Fall sind nur Werte von 0-31 bzw. 0-63 möglich. Das folgende Makro konvertiert `R8G8B8`-Farbwerte in das Format `R5G6B6`:

*Shift me, baby!*

```
#define RGB16BIT(r,g,b) ((b%32) + ((g%64) << 6) +  
                        ((r%32) << 11))
```

Kompliziert? Nicht wirklich. Durch die Moduloberechnung begrenzen wir die RGB-Werte auf den ihnen zur Verfügung stehenden Wertebereich. Dann müssen wir die Werte nur noch entsprechend an die richtige Position schieben. Für den Blauwert ist das nicht nötig, da er ganz rechts steht. Entsprechend schieben wir den Grünwert um sechs Stellen nach links, weil er sechs Stellen vor dem Blauwert belegt. Der Rotwert wird dann einmal um die sechs Stellen nach links geschoben, die der Grünwert braucht, und dann noch einmal um fünf Stellen, die er selbst belegt, also insgesamt um elf Stellen. Durch diese Rechnung haben wir den 24-Bit-RGB-Wert nun zu einem analogen 16-Bit-Wert umgerechnet.

Nach all der Verwirrung schauen wir uns lieber die gesamte Funktion an:

```
HRESULT ZFXD3DSkinManager::CreateTexture(
    ZFXTTEXTURE *pTexture,
    bool bAlpha) {

    D3DLOCKED_RECT    d3dRect;
    D3DFORMAT          fmt;
    DIBSECTION         dibS;
    HRESULT             hr;
    int                LineWidth;
    void               *pMemory=NULL;

    HBITMAP hBMP = (HBITMAP)LoadImage(NULL,
                                        pTexture->chName,
                                        IMAGE_BITMAP,0,0,
                                        LR_LOADFROMFILE |
                                        LR_CREATEDIBSECTION);

    if (!hBMP) return ZFX_FILENOTFOUND;

    GetObject(hBMP, sizeof(DIBSECTION), &dibS);

    // wir unterstützen nur 24-Bit-Bitmaps
    if (dibS.dsBmih.biBitCount != 24) {
        DeleteObject(hBMP);
        return ZFX_INVALIDFILE;
    }

    if (bAlpha) fmt = D3DFMT_A8R8G8B8;
    else fmt = D3DFMT_R5G6B5;

    long lWidth  = dibS.dsBmih.biWidth;
    long lHeight = dibS.dsBmih.biHeight;
    BYTE *pBMPBits = (BYTE*)dibS.dsBm.bmBits;

    // erzeuge IDirect3DTexture-Objekt
    hr = m_pDevice->CreateTexture(lWidth, lHeight, 1, 0,
                                fmt, D3DPPOOL_MANAGED,
                                (LPDIRECT3DTEXTURE9*)
                                (&(pTexture->pData)),
                                NULL);

    if (FAILED(hr)) return ZFX_FAIL;

    // Dummy-Pointer setzen
    LPDIRECT3DTEXTURE9 pTex = ((LPDIRECT3DTEXTURE9)
                               pTexture->pData);

    if (FAILED(pTex->LockRect(0, &d3dRect, NULL, 0)))
        return ZFX_BUFFERLOCK;

    if (bAlpha) {
```



```

        LineWidth = d3dRect.Pitch >> 2; // 32 Bit = 4 Byte
        pMemory = (DWORD*)d3dRect.pBits;
    }
    else {
        LineWidth = d3dRect.Pitch >> 1; // 16 Bit = 2 Byte
        pMemory = (USHORT*)d3dRect.pBits;
    }

    // kopiere jeden Pixel
    for (int cy = 0; cy < lHeight; cy++) {
        for (int cx = 0; cx < lWidth; cx++) {

            if (bAlpha) {
                DWORD Color = 0xff000000;
                int i = (cy*lWidth + cx)*3;
                memcpy(&Color, &pBMPBits[i],
                    sizeof(BYTE)*3);
                ((DWORD*)pMemory)[cx+(cy*LineWidth)]=Color;
            } // 32 Bit
            else {
                // konvertiere 24-Bit-Pixel in 16-Bit-Pixel
                UCHAR B=(pBMPBits[(cy*lWidth+cx)*3 +0])>>3,
                    G=(pBMPBits[(cy*lWidth+cx)*3 +1])>>3,
                    R=(pBMPBits[(cy*lWidth+cx)*3 +2])>>3;

                USHORT Color = RGB16BIT(R,G,B);

                // Pixel in 16-Bit-Farbe schreiben
                ((USHORT*)pMemory)[cx+(cy*LineWidth)]=Color;
            } // 16 Bit

        } // for
    } // for

    pTex->UnlockRect(0);
    DeleteObject(hBMP);
    return ZFX_OK;
} // CreateTexture

```

*Ein bisschen
Wiederholung
schadet nie ...*

Für den Fall, dass es hier jemanden gibt, der seit längerem nicht mehr mit Direct3D gearbeitet hat, erläutere ich hier noch einmal die wichtigsten Dinge bei der Befüllung von Textur-Objekten: Um an den Speicherbereich zu kommen, in dem ein IDirect3DTextureX-Objekt seine Pixeldaten speichert, ruft man die Funktion LockRect() auf. Damit sperrt man die Textur, so dass Direct3D sie nicht mehr verwenden kann. In der Direct3D-Struktur D3DLOCKED_RECT findet man dann mit pBits einmal einen Pointer auf den Speicherbereich und mit Pitch zum anderen einen Wert, der angibt, wie breit eine Daten-Zeile im Speicher ist.

Die Daten einer Bitmap liegen im Speicher zwar in Form eines linearen Arrays vor, aber für den Zugriff auf die Daten kann man trotzdem die Breite und Höhe des Bildes verwenden. Mit der Formel

!!
STOP

$$\text{Index} = (\text{ZeilenNo} * \text{Zeilenbreite}) + \text{SpaltenNo}$$

kann man den Index eines Pixels in diesem linearen Array ausrechnen, wobei die x-Koordinate des Pixels in dem Bitmap-Bild dem Wert von SpaltenNo und die y-Koordinate dem Wert von ZeilenNo entspricht. Die Zeilenbreite wird bestimmt durch die Breite des Bildes in Pixeln.

Aber genau das ist bei Direct3D-Texturen (und auch den fast analogen IDirect3DSurfaceX-Objekten) nicht der Fall. Die Zeilenbreite in einem solchen Direct3D-Objekt entspricht nicht mehr unbedingt der ursprünglichen Breite des Bildes. Am Ende jeder Zeile können noch Füllbits hinzukommen. Durch die Variable Pitch erhalten wir bei einem Lock den genauen Wert der Zeilenbreite in Bytes. Und mit genau diesem Wert sollte man immer arbeiten. Wenn wir 32-Bit-Surfaces verwenden, teilen wir diesen Wert noch durch 4 (was dem Shiften um 2 nach rechts entspricht), weil ein Pixel vier Bytes lang ist. Bei 16-Bit-Surfaces teilen wir entsprechend durch 2, weil ein Pixel zwei Bytes lang ist.

Nun lesen wir die Pixeldaten aus der DIB-Sektion Pixel für Pixel aus und konvertieren die Pixeldaten im Falle der 16-Bit-Texturen entsprechend. Bei dem Zugriff auf das Array mit den 24-Bit-Daten der DIB-Sektion multiplizieren wir übrigens immer mit 3, weil ein Pixel dort drei Bytes lang ist.

Sind alle Pixeldaten gut in der Textur verstaut, lösen wir den Lock durch Aufruf der `UnlockRect()`-Funktion und geben die Textur damit wieder zur Verwendung frei. Das Bitmap-Bild brauchen wir nun auch nicht mehr, daher geben wir die von der WinAPI allokierten Ressourcen durch die Funktion `DeleteObject()` wieder frei. Damit haben wir die Grafik aus der Bitmap-Datei in ein Direct3D-Textur-Objekt geladen.

Fertig!

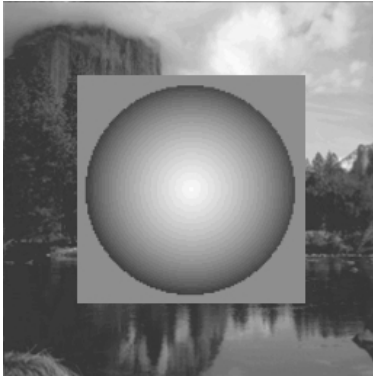
5.5 Transparenz der Texturen einstellen

Was wäre die virtuelle Welt ohne Transparenzeffekte? Sicherlich bei weitem nicht so schön, wie sie es mit solchen Effekten ist. Halb durchsichtige Materialien wie beispielsweise dreckiges Glas oder Wasserflächen wären nicht möglich. Daher gehe ich hier kurz auf die Grundlagen der Darstellung von Transparenz ein.

Abbildung 5.5 zeigt zwei Rechtecke mit Texturen, wobei das vordere, kleinere Rechteck das hintere überdeckt. Wir werden zu dieser Abbildung später noch einmal zurückkommen, wenn wir die Textur des vorderen Rechtecks mit Transparenzeffekten belegt haben.

Abbildung 5.5:

Zwei Rechtecke mit
Texturen, die
übereinander liegen



Transparenz in der Computergrafik

Die Anwendung transparenter Teile in einer virtuellen Welt führt immer wieder zu Schwierigkeiten mit der Darstellung – allerdings nur dann, wenn man selbst keine Vorstellung davon hat, wie der Computer Transparenz simuliert. Daher möchte ich in diesem Exkurs noch einmal kurz darstellen, wie diese Simulation technisch abläuft.

Dazu verdeutlichen wir uns zunächst, wie Transparenz realistisch funktioniert. Das wohl strapazierteste Beispiel dafür ist eine Glasscheibe. Durch diese sehen wir auch Objekte, die auf der von uns abgewandten Seite der Glasscheibe liegen. Warum ist das so? Nun, die Photonen, die diese Objekte emittieren oder reflektieren, durchdringen das Glas fast ungehindert und erreichen so dennoch unser Auge. Folglich muss ein Computer dies ebenso umsetzen.

Die Grafikkarte schaut also nach, welche Objekte hinter der virtuellen Glasscheibe liegen, und zeigt diese weiterhin an, ohne die Glasscheibe davor zu rendern. Richtig?

Nein, eben so ist es nicht. Das Problem ist, dass die Grafikkarte keine Vorstellung von Objekten hat und auch kein Gedächtnis, was wann wo gerendert wurde. Immer dann, wenn man ein Objekt einer Szene rendert, malt die Grafikkarte eine Menge von Pixeln in den Pixel-Buffer und zugehörige Tiefenwerte der Pixel in den Depth-Buffer. Wir rendern also erst die Objekte hinter dem Fenster. Dann rendern wir die Glasscheibe mit Alpha-Blending für die Transparenz. Nun transformiert die Grafikkarte die virtuelle Glasscheibe zu einer Menge von Pixeln mit Tiefenwerten. Dann schaut sie an den korrespondierenden Stellen im Pixel-Buffer nach, ob dort bereits Pixel vorhanden sind. Ist dies der Fall, dann nimmt sie den Farbwert dieser Pixel und verrechnet ihn mit dem Farbwert des entsprechenden Pixels

der Glasscheibe, basierend auf dem Grad der Transparenz. Der so entsprechend kumulierte Farbwert wird dann mit dem Tiefenwert der Glasscheibe gerendert.

Eben das ist eine Sache, die es zu bedenken gilt. Transparenz lässt sich eben nur dann korrekt darstellen, wenn **zuerst** alle Objekte hinter der Glasscheibe fertig gerendert sind und **dann erst** die Glasscheibe gerendert wird. Würden wir zuerst die Glasscheibe rendern, so wären im Pixel-Buffer noch keine anderen Pixel, und es könnte kein kumulierter Farbwert errechnet werden. Es wird also einfach der Pixel der Glasscheibe gerendert. Rendern wir dann die Objekte hinter der Scheibe, so hat die Grafikkarte längst vergessen, dass da mal was mit Transparenz war. Die Werte im Depth-Buffer (von der Glasscheibe) verbieten den Pixeln der Objekte nun, sich in den Pixel-Buffer zu rendern. Folglich schimpft der Programmierer auf die blöde Grafikkarte, weil seine Objekte nicht am Bildschirm gerendert werden. Dabei ist er selbst schuld an der Misere.

Wenn man mit Transparenz arbeitet, dann gelten immer die folgenden beiden Grundsätze:

- Rendere zuerst alle Polygone ohne Transparenz!
- Rendere alle Polygone mit Transparenz, sortiert nach ihrer Entfernung zur Kamera, und beginne mit den entferntesten!

Um Transparenzeffekte zu erreichen, gibt es verschiedene Möglichkeiten. Die folgende Liste gibt eine kurze Übersicht über die wichtigsten Methoden:

- ➔ Alphawert im Material
- ➔ Alphawert in der Farbe eines Vertex
- ➔ Alphawert in der Texturfarbe

Wir werden hier mit der letztgenannten Methode arbeiten. In jedem Pixel einer Textur lässt sich neben den Rot-, Grün- und Blauwerten der Farbe ja auch noch der so genannte Alphawert speichern. Dieser gibt das Ausmaß der Transparenz dieses Pixels an, wobei ein Wert von 1.0f für 100% sichtbar und ein Wert von 0.0f für 0% sichtbar, d.h. total transparent steht.

In der Welt von Direct3D stellen wir Farben aber nicht mehr als einzelne Werte mit ARGB-Anteilen dar, sondern als Kombination der einzelnen Komponenten in einem 32-Bit-Wert, einem `DWORD`. Die folgende Funktion nimmt die einzelnen Farbwerte in einem Wertebereich von 0-255, also 8 Bit, auf und wandelt diese in einen für Direct3D gültigen Farbwert um.

Direct3D-Farbwerte



```

DWORD ZFXD3DSkinManager::MakeD3DColor(UCHAR R, UCHAR G,
                                         UCHAR B, UCHAR A)
{
    DWORD Color;
    DWORD RBitMask = 0x00ff0000;
    DWORD GBitMask = 0x0000ff00;
    DWORD BBitMask = 0x000000ff;
    DWORD ABitMask = 0xff000000;

    // skaliere auf Prozentwerte
    float fA = A/255.0f;
    float fR = R/255.0f;
    float fG = G/255.0f;
    float fB = B/255.0f;

    // erstelle absolute Werte, und beschneide auf Bereich
    Color = ( ((DWORD)(ABitMask * fA)) & ABitMask ) +
            ( ((DWORD)(RBitMask * fR)) & RBitMask ) +
            ( ((DWORD)(GBitMask * fG)) & GBitMask ) +
            ( ((DWORD)(BBitMask * fB)) & BBitMask );

    return Color;
} // MakeD3DColor

```

Farbarithmetik

Die Bitmasken geben dabei an, welchen Wert ein entsprechender Rot-, Grün-, Blau- oder Alpha-Anteil einer Farbe maximal hat. Die Werte im Bereich von 0-255 werden dann auf Werte im Bereich von 0.0f bis 1.0f konvertiert. Aus diesen setzen wir dann den endgültigen Farbwert zusammen. Die konvertierten Einzelanteile besagen dabei quasi, wie viel Prozent des jeweils maximalen Anteils von Rot, Grün, Blau und Alpha genommen werden soll. Daher multiplizieren wir diese Werte mit den entsprechenden Bitmasken. Bei dieser Multiplikation können aber auch Werte entstehen, die außerhalb des Bitbereichs liegen, den ein Wert im gesamten Farbwert belegen darf. Daher verwenden wir das logische UND und verknüpfen das Ergebnis der Multiplikation mit der Bitmaske, die auch gleichzeitig den gültigen Bereich angibt, in dem sich ein Farbwert bewegen darf. Dadurch bescheiden wir die eventuell zu lang gewordenen Werte auf den für sie vorgesehenen Bereich in dem DWORD-Wert.

Andere Farbformate

Das Ergebnis dieser Funktion ist ein 32 Bit großer Wert, der eine Codierung einer Farbe im Format A8R8G8B8 darstellt. Man kann hier auch beliebige andere Farbformate als Zielformat verwenden. Direct3D kennt nämlich grundsätzlich als Farbwert nur einen 32-Bit-Wert. Die geringere Bittiefe anderer Farbformate wird hier nicht über die Größe des endgültigen Farbwertes gesteuert, sondern über die maximal zulässigen Werte für die einzelnen Komponenten. Langer Rede kurzer Sinn: Für andere Farbformate muss man einfach entsprechende Bitmasken mit entsprechend geringeren Maximalwerten setzen.

Color-Keys über Alpha Channels

Mit dieser `MakeD3DColor()`-Funktion können wir nun daran gehen, Alphawerte für eine Textur einzustellen. Als Erstes kümmern wir uns um die so genannten Alpha-Keys. Dies sind Werte, denen man gezielt eine Transparenz zuweist. So kann man sagen, alle Pixel der Farbe Blau ($RGB=0.0f, 0.0f, 1.0f$) sollen zu 80 Prozent transparent sein ($A=0.2f$). Dann lassen wir die Funktion `SetAlphaKey()` auf die Textur los, und diese sucht alle blauen Pixel in der Textur und verpasst ihnen einen entsprechenden Alphawert. Abbildung 5.6 zeigt nochmals die Szene aus Abbildung 5.5, doch diesmal haben wir der Textur des vorderen Rechtecks einen Alpha-Key verpasst. Die Randfarbe (rosa bzw. hellgrau in der Schwarzweiß-Abbildung) ist nun vollkommen transparent.

*Gezielte
Transparenz*

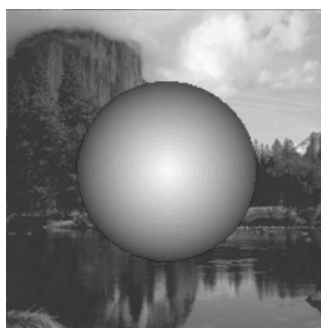


Abbildung 5.6:
Die vordere Textur
hat nun einen
Alpha-Key.

Das klingt nicht nur einfach zu realisieren, sondern ist es auch. Wir durchlaufen einfach alle Pixel der Textur und ersetzen die entsprechenden Farbwerte durch analoge Farbwerte, die einen entsprechenden Alphawert enthalten.

Umsetzung

```
HRESULT ZFXD3DSkinManager::SetAlphaKey(
    LPDIRECT3DTEXTURE9
    *ppTexture,
    UCHAR R, UCHAR G,
    UCHAR B, UCHAR A) {

    D3DSURFACE_DESC d3dDesc;
    D3DLOCKED_RECT d3dRect;
    DWORD dwKey, Color;

    // Sicherheits-Check: Muss 32-Bit-ARGB-Format sein
    (*ppTexture)->GetLevelDesc(0, &d3dDesc);
    if (d3dDesc.Format != D3DFMT_A8R8G8B8)
        return ZFX_INVALIDPARAM;

    // Auszutauschender Farbwert
    dwKey = MakeD3DColor(R, G, B, 255);

    // Neu zu setzender Farbwert
    if (A > 0) Color = MakeD3DColor(R, G, B, A);
```



```

else Color = MakeD3DColor(0, 0, 0, A);

if (FAILED((*ppTexture)->LockRect(0, &d3dRect, NULL,
                                0)))

    return ZFX_BUFFERLOCK;

// überschreibe alle Key-Pixel
for (DWORD y=0; y<d3dDesc.Height; y++) {
    for (DWORD x=0; x<d3dDesc.Width; x++) {
        if ( ((DWORD*)d3dRect.pBits)[d3dDesc.Width*y+x]
            == dwKey )
            ((DWORD*)d3dRect.pBits)[d3dDesc.Width*y+x]
                = Color;
    }
}
(*ppTexture)->UnlockRect(0);

return ZFX_OK;
} // SetAlphaKey

```

**!!
STOP**

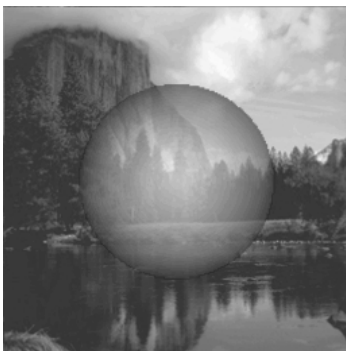
Man beachte allerdings, dass diese Funktion erwartet, dass alle Pixel des gesuchten RGB-Wertes einen Alphawert von 255 bzw. 1.0f haben. Man muss die Alpha-Keys also für eine Textur immer im jungfräulichen Zustand setzen, bevor man beispielsweise eine allgemeine Transparenz für die gesamte Textur einstellt.

Allgemeine Transparenz über Alpha Channels

Was der Alpha-Key für den einzelnen Pixel ist, das ist die allgemeine Transparenz für die gesamte Textur. Egal ob wir spezielle Alpha-Keys für die Textur gesetzt haben oder nicht, wir können ja auch sagen, dass alle Pixel einer Textur, egal welche Farbe sie haben, eine bestimmte Transparenz haben sollen. Dann loopen wir einfach durch alle Pixel der Textur und setzen einen entsprechenden Alphawert für jeden von ihnen.

Abbildung 5.7:

Neben dem Alpha-Key hat die vordere Textur nun auch allgemeine Transparenz, so dass die hintere Textur durchscheint.



Genau das macht die folgende Funktion:

```
HRESULT ZFXD3DSkinManager::SetTransparency(LPDIRECT3DTEXTURE9
                                           *ppTexture,
                                           UCHAR Alpha) {

    D3DSURFACE_DESC d3dDesc;
    D3DLOCKED_RECT d3dRect;
    DWORD          Color;
    UCHAR          A, R, G, B;

    // Sicherheits-Check: Muss 32-Bit-ARGB-Format sein
    (*ppTexture)->GetLevelDesc(0, &d3dDesc);
    if (d3dDesc.Format != D3DFMT_A8R8G8B8)
        return ZFX_INVALIDPARAM;

    if (FAILED((*ppTexture)->LockRect(0, &d3dRect, NULL,
                                      0)))
        return ZFX_BUFFERLOCK;

    // laufe durch alle Pixel
    for (DWORD y=0; y<d3dDesc.Height; y++) {
        for (DWORD x=0; x<d3dDesc.Width; x++) {
            // Hole den Farbwert des Pixels
            Color = ((DWORD*)d3dRect.pBits)[d3dDesc.Width*y+x];

            // berechne neuen ARGB-Wert vom Pixel-Farbwert
            A = (UCHAR)( (Color & 0xff000000) >> 24);
            R = (UCHAR)( (Color & 0x00ff0000) >> 16);
            G = (UCHAR)( (Color & 0x0000ff00) >> 8);
            B = (UCHAR)( (Color & 0x000000ff) >> 0);

            // setze nur, wenn neuer Alphawert höher
            if (A >= Alpha)
                A = Alpha;

            ((DWORD*)d3dRect.pBits)[d3dDesc.Width*y+x] =
                MakeD3DColor(R, G, B, A);
        }
    }
    (*ppTexture)->UnlockRect(0);

    return ZFX_OK;
} // SetTransparency
```





Man beachte hier, dass der Alphawert für einen Pixel nur dann gesetzt wird, wenn sein bereits vorhandener Alphawert größer ist, er also bisher ein geringeres Maß an Transparenz hat. Setzt man beispielsweise alle blauen Pixel durch einen Alpha-Key auf 100% Transparenz, so soll dies ja auch dann so bleiben, wenn wir dem Rest der Textur eine allgemeine Transparenz von 80% zuweisen.

5.6 Ein Blick zurück, zwei Schritt nach vorn

Vor und hinter den Kulissen

Hoppla, schon wieder ein Kapitel geschafft. Doch sollte man sich nicht täuschen lassen, was es hier alles zu lernen gab. Auf der praktischen Seite stand sicherlich die Arbeit mit Materialien und Texturen im Vordergrund. Aber wir haben auch die praktische Anwendung des Managements von Komponenten in einer Engine kennen gelernt. Und genau diesem Punkt sollte man nicht zu wenig Beachtung schenken: In diesem Kapitel haben wir gelernt, mit den Ressourcen der Grafikkarte nicht zu verschwenderisch umzugehen. In späteren Kapiteln werden wir auch noch sehen, wie man durch sinnvolles Management auch mit der Rechenzeit der CPU und GPU schonend umgehen kann.

Background

Auf der theoretischen Seite haben wir vor allem etwas darüber gelernt, wie die Beleuchtung virtueller Welten in der Computergrafik funktioniert. Leider gibt es für dieses Kapitel keine lauffähige Demo, denn wir haben hier lediglich eine Komponente entwickelt. Was unseren bisherigen Code betrifft, so haben wir ja bisher noch nicht die Möglichkeit, überhaupt etwas rendern zu können. Doch dazu werden wir bald kommen, keine Panik.

Mögliche Erweiterungen

Betrachten wir unseren Skin-Manager in seiner Gänze, so fehlt ihm objektiv betrachtet eventuell noch eine Kleinigkeit. Als Skin haben wir bisher nur Materialien optional in Verbindung mit Texturen betrachtet. Solche Skins brauchen wir später immer dann, wenn wir Polygone rendern wollen. Allerdings kann man mit jeder 3D-API neben Polygonen auch 2D-Grafik rendern. Dazu bietet beispielsweise Direct3D das Interface `IDirect3DSurface` an. Ein solches Surface-Objekt ist eigentlich nichts anderes als eine Bitmap-Grafik, die man auf den Bildschirm rendern kann. Nun könnte man auch das Management von solchen Surface-Objekten mit in den Skin-Manager stecken, da es ja auch um Bitmap-Grafiken geht. Den Manager entsprechend zu erweitern, dass er auch ein Array `m_pSurfaces` verwaltet und IDs auf 2D-Grafik direkt an den Aufrufer ausgibt, das überlasse ich euch als Hausaufgabe. Ein Surface-Objekt kann man ganz analog einem Textur-Objekt von Direct3D locken und dann mit Pixeldaten befüllen. So kann der Skin-Manager beispielsweise auch ein Inventar von 2D-Grafiken führen, das per Aufruf einer Renderfunktion und Übergabe der Inventar-Nummer (ID) am Bildschirm angezeigt wird. Sinnvoll ist dies vor allem für Zwischengrafiken, Startbildschirme usw.

Im nächsten Kapitel werden wir uns endlich darum kümmern, etwas auf den Bildschirm zu bringen . Dort werden wir lernen, wie wir Polygone rendern können. Bevor es so weit ist, müssen wir jedoch noch einiges an Arbeit in unsere Render-Device-DLL investieren. Es wird noch ein langer Weg werden, denn wir wollen die Arbeit ja auch gleich richtig machen. Dabei wird uns auch das Thema Management wieder endlos lange Seiten quälen, denn nichts sorgt für mehr Effektivität als das vernünftige Management der zu rendernden Polygone.

Ausblick

6 Das Render-Interface der ZFXEngine

» Wenn man einen Riesen sieht, so untersuche man erst den Stand der Sonne und gebe Acht, ob es nicht der Schatten eines Pygmäen ist.« (Novalis)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ perspektivische und orthogonale Projektion von 3D auf 2D
- ➔ Koordinatenumrechnung zwischen dem 2D- und 3D-Raum
- ➔ statische und dynamische Vertex-Buffer
- ➔ effizientes Rendern von grafischen Primitiven durch Caching

6.1 Simplizität versus Flexibilität

Ein jedes Computerprogramm ist ein komplexes Gebilde, für das wir hier die Analogie einer Zwiebel bemühen wollen. Die äußeren Schichten des Programms werden durch den Code der Applikation gebildet, der das User-Interface repräsentiert. Ob dies nun eine Eingabemaske oder die Ego-Perspektive eines 3D-Shooters ist, sei dahingestellt. Die inneren Schichten befassen sich auf verschiedenen Ebenen mit dem Management von Daten und dem Wrappen gewisser APIs, beispielsweise der WinAPI, WinSock, OpenGL oder DirectX. Den Kern der Zwiebel bildet dann die Kommunikation mit der Hardware, beispielsweise das Senden von Triangles zum Rendern an die Grafikkarte.

Zwiebeln

Hier kommen die konkurrierenden Ziele Simplizität und Flexibilität ins Spiel. Eine ideale 3D-Engine ist beispielsweise sowohl hochgradig flexibel als auch super simpel zu bedienen. Leider ist aber beides gleichzeitig nicht zu erreichen. Die einfachste 3D-Engine kann beispielsweise durch einen einzigen Funktionsaufruf gestartet werden, ein 3D-Modell rendern und sich dann wieder beenden. Einfacher geht es wohl kaum noch, allerdings geht es auch kaum noch unflexibler. Je mehr Flexibilität wir jedoch in die Engine einbauen, desto mehr kommen wir in Gefilde, in denen wir lediglich für jede Funktion der API (z.B. Direct3D oder OpenGL) eine eigene Wrapper-Funktion mit identischer

Riesen

Parameterliste schreiben. Dann ist unsere Engine so flexibel, wie es nur geht, aber eben auch genauso kompliziert in der Verwendung wie eine 3D-API.

Pygmäen Unser Job als Designer einer 3D-Engine ist es daher, einen sinnvollen Kompromiss zwischen Simplität und Flexibilität zu finden. Im Verlaufe dieses Kapitels werden wir entsprechend die Funktionalität unserer `ZFXD3DRender Device`-Klasse sehr umfassend erweitern. Unser Ziel ist es dabei, möglichst viele für die API notwendige Abläufe vor dem Benutzer der *ZFXEngine* zu verstecken. Das bedeutet jedoch, dass wir uns an vielen Stellen überlegen müssen, was in welcher Reihenfolge ausgeführt werden soll, kann oder muss.

Novalis Beim Durcharbeiten der folgenden Seiten wird sich der eine oder andere daher vielleicht gelegentlich mit einem Riesen konfrontiert sehen. Man tut aber gut daran, wenn man sich den Code mal von einer abstrakteren Position aus anschaut. Hier wird man dann hoffentlich den Schatten des Pygmäen erkennen und die Einfachheit im Code entdecken. Nun denn, meine jungen 3D-Jedis, stellt einfach die Augen und Ohren auf, und folgt mir auf einer kurze Reise durch den Wrapper einer 3D-API.

6.2 Projekteinstellungen

Neue Dateien In diesem Kapitel haben wir viel vor, daher müssen wir auch an unserem Projekt viel ergänzen. Wir kopieren also das bisherige Projekt aus dem dritten Kapitel in ein neues Verzeichnis für dieses Kapitel und fügen dem Projekt noch einiges hinzu. Als Erstes benötigen wir ein paar *.cpp- und *.h-Dateien mehr, als wir bisher haben. Die folgenden Dateien fügen wir dem Projekt ZFXD3D als neue, leere Dateien hinzu:

- ➔ ZFXD3D_misc.cpp
- ➔ ZFXD3D_vcache.cpp
- ➔ ZFXD3D_vcache.h

Und die Arbeit der letzten Kapitel Die erste der drei Dateien dient dazu, einen Teil der Funktionen für die Klasse ZFXD3D aufzunehmen. Die anderen beiden Dateien sind für die Definition und Implementierung einer neuen Klasse innerhalb der DLL gedacht. Neben diesen leeren Dateien müssen wir aber auch die Früchte unserer Arbeit aus den letzten Kapiteln einbinden. Daher kopieren wir die Dateien `zfx3d.h` und `zfx3d.lib` mit unseren Mathefunktionen auch in das neue Verzeichnis und binden sie in unser Projekt ein. Gleiches gilt natürlich für unsere Skin-Manager-Dateien `ZFXD3D_skinman.cpp` und `ZFXD3D_skinman.h`, die zum Projekt ZFXD3D hinzugefügt werden.

Erweiterung des Interfaces In diesem Kapitel werden wir die Funktionalität unseres Render-Devices in der DLL ganz beträchtlich erweitern. Dazu müssen wir die Änderungen aber oftmals an zwei Stellen durchführen. Zum einen ergänzen wir die Defi-

nition des Interfaces `ZFXRenderDevice` und zum anderen die Definition der davon abgeleiteten Klasse `ZFXD3D`. Alle neuen öffentlichen Funktionen und die wichtigsten Attribute stehen dabei natürlich im Interface, die privaten Hilfsmethoden und Attribute in der abgeleiteten Klasse. Einen eisernen Grundsatz der strengen Objektorientierung werden wir aber auch hier wieder mit Vergnügen brechen: Unsere Interface-Definition erhält nun ein Attribut, das öffentlich zugänglich ist, und zwar ist dies eine Instanz unseres Skin-Managers:

```
// Im Interface class ZFXRenderDevice:
private:
    ZFXSkinManager *m_pSkinMan;

public:
    virtual ZFXSkinManager* GetSkinManager(void)=0;
```



Man beachte allerdings, dass das Attribut selbst natürlich vom Interface-Typ `ZFXSkinManager` und nicht vom Typ der von dieser Klasse abgeleiteten Klasse `ZFXD3DSkinManager` ist. Schließlich ist dem Anwender der *ZFXEngine* nach außen hin lediglich das Interface und nicht unsere Implementierung dieses Interfaces bekannt. Dass wir dieses Attribut hier öffentlich zugänglich machen, hat den Sinn, dass die Alternative ein noch schlechterer Weg wäre. Entweder müssten wir den Skin-Manager losgelöst von dem Render-Device als eigenes Objekt anbieten, das sich der Anwender selbst instanziiert muss. Oder wir müssten den Manager als privates Attribut in die Klassen-Definition mit aufnehmen und sämtliche öffentliche Methoden des Managers auch als Methoden des Render-Device implementieren, die ihrerseits lediglich die Methoden des Skin-Managers aufrufen müssten. Diese Variante halte ich in diesem Fall für sehr unschön, daher lassen wir das Attribut öffentlich über eine Accessor-Methode. Beispielsweise kann ein Anwender nun wie folgt ein Skin-Objekt erzeugen, wenn er einen Pointer auf das Render-Device hat:

Igitt, ein public-Attribut!

```
UINT nMySkin;
ZFXCOLOR cWhite = { 1.0f, 1.0f, 1.0f, 1.0f };
ZFXCOLOR cBlack = { 0.0f, 0.0f, 0.0f, 1.0f };

pDevice->GetSkinManager()->AddSkin(&cWhite, &cWhite,
                                    &cBlack, &cBlack, 0.0f,
                                    &nMySkin);
```



Die kompletten Ergänzungen des Interfaces `ZFXRenderDevice` und der Klasse `ZFXD3D` möchte ich hier jedoch noch nicht abdrucken. Es kommen eine ganze Menge an Funktionen und Attributen hinzu, mit denen wir jetzt eigentlich noch nichts anfangen können. Wer seine Neugier nicht mehr zügeln kann, den möchte ich daher bitten, auf der CD-ROM zu diesem Buch den entsprechenden Arbeitsbereich zu öffnen. Im Folgenden werde ich an entsprechen-

Die neuen Interface- und Klassen-Definitionen

der Stelle die neu hinzukommenden Funktionen und Attribute erläutern. Dabei sind alle öffentlichen Funktionen grundsätzlich im Interface als virtuelle Prototypen definiert. Alle für die Engine wichtigen Attribute stehen ebenfalls im Interface. Lediglich die privaten Funktionen stehen ausschließlich in der Klassen-Definition `ZFXD3D`. Dasselbe gilt für diejenigen Attribute, die ausschließlich intern für die Arbeit mit Direct3D von Bedeutung sind, von einem beliebigen `ZFXRenderDevice` (beispielsweise basierend auf OpenGL) aber nicht explizit verwendet werden.

6.3 Sicht und Projektion

Projektion Zwei grundlegende Aspekte der zweidimensionalen Darstellung einer dreidimensionalen Szene sind die Sicht und die Projektion. Logischerweise beinhaltet eine solche Darstellung eine Projektion der Daten, und zwar von 3D auf 2D. Doch was für eine Projektion man dabei verwendet, das bleibt einem selbst überlassen. In diesem Kapitel werden wir mit den beiden gängigsten Arten der Projektion arbeiten. Zum einen ist dies die übliche *perspektivische Projektion*, wie man sie aus jedem 3D-Spiel kennt, zum anderen aber auch die *orthogonale Projektion*, wie sie beispielsweise in den 2D-Ansichten von CAD- oder 3D-Modellierungssoftware verwendet wird.

Sicht Der andere Aspekt einer 3D-Engine ist die Sicht. Diese schließt mehrere Dinge mit ein, vor allem aber die so genannte *Viewmatrix*. In dieser Matrix ist die Position des Betrachters der 3D-Szene in der virtuellen Welt festgelegt. Daneben gehört zu der Sicht aber auch das Sichtfenster, in dem die projizierte Szene angezeigt werden soll. Dies ist entweder der gesamte Bildschirm oder nur ein Teil davon. Und diesen Bereich bezeichnet man in der Computergrafik als *Viewport*.

Wir Möchte man nun eine 3D-Engine implementieren, die möglichst viele optische Effekte unterstützt, so ist es unabdingbar, dass man den Bereich der Sicht und Projektion möglichst flexibel gestaltet. Beispielsweise sollte man schnell zwischen orthogonaler und perspektivischer Projektion wechseln können. Ebenso sollte es möglich sein, die Parameter der perspektivischen Projektion zu verändern. Dadurch kann man beispielsweise einen Zoomeffekt für ein Zielfernrohr oder Fernglas erzielen. Eine entsprechende Matrix lässt sich zwar recht schnell berechnen, aber bekanntermaßen ist es schneller, vorher bekannte Werte zwischenspeichern und später nur abzurufen, als sie neu zu berechnen. Zudem beinhaltet die Berechnung der Projektionsmatrix einige hässliche trigonometrische Funktionen, die ja nicht gerade schnell in der Berechnung sind. Daher bieten wir mit unserer Engine die Möglichkeit an, eine gewisse Anzahl an Matrizen vorher zu berechnen und zwischenspeichern, und zwar durch so genannte *multiple Stages*, zu denen wir gleich kommen werden.

In dem nun folgenden Abschnitt dieses Kapitels werden wir die MISC-Funktion für die Klasse `ZFXD3D` entwickeln, die natürlich dann in der Datei `ZFXD3D_misc.cpp` steht. Dafür definieren wir im Interface `ZFXRenderDevice` die nun folgenden öffentlichen Funktionen und Attribute, in denen die wichtigsten Werte gespeichert werden sollen:

MISC

```
// unter private:
float      m_fNear,           // Near-Plane
           m_fFar;           // Far-Plane
ZFXENGINEMODE m_Mode;        // 2D, 3D,...
int        m_nStage;         // Stage (0-3)
ZFXVIEWPORT m_VP[4];         // Viewports

// unter public:
// Viewmatrix aus vRight, vUp, vDir, vPos
virtual HRESULT SetView3D(const ZFXVector&,
                        const ZFXVector&,
                        const ZFXVector&,
                        const ZFXVector&)=0;

// Viewmatrix aus Position, Fixpunkt, WorldUp
virtual HRESULT SetViewLookAt(const ZFXVector&,
                             const ZFXVector&,
                             const ZFXVector&)=0;

// Near und Far Clipping-Plane
virtual void SetClippingPlanes(float, float)=0;

// Modus der Stage, 0:=perspective, 1:=ortho
virtual HRESULT SetMode(int, int n)=0;

// Field of View und Viewport für Stage n
virtual HRESULT InitStage(float, RECT*, int n)=0;

// Ebenen des View-Frustums
virtual HRESULT GetFrustum(ZFXPlane*)=0;

// Bildschirmpunkt zu Weltstrahl
virtual void Transform2Dto3D(const POINT &pt,
                           ZFXVector *vc0,
                           ZFXVector *vcD)=0;

// Weltkoordinaten zu Bildschirmkoordinaten
virtual POINT Transform3Dto2D(const ZFXVector
                             &vcP)=0;
```



Neben den Werten für die Near und Far Clipping-Plane und einem `int`-Wert, zu dem wir gleich kommen, haben wir noch zwei neue Datentypen bei den Attributen. Diese sind wie gewohnt in der Datei `ZFX.h` definiert,

*Die neuen
Attribute*

damit sie überall zugänglich sind. Bei `ZFXVIEWPORT` handelt es sich um eine einfache Viewport-Struktur, durch die man Teile des aktiven Fensters als Render-Bereich festlegen kann. Die Struktur ist wie folgt definiert:



```
// einfacher Viewport Typ
typedef struct ZFXVIEWPORT_TYPE {
    DWORD X;
    DWORD Y;
    DWORD Width;
    DWORD Height;
} ZFXVIEWPORT;
```

Dann haben wir noch einen enumerierten Typ namens `ZFXENGINEMODE`. Mit diesem können wir unsere Engine zwischen drei verschiedenen Betriebsmodi umschalten. Wie wir das machen und was diese bewirken, das sehen wir im Verlauf dieses Kapitels.



```
typedef enum ZFXENGINEMODE_TYPE {
    EMD_PERSPECTIVE, // perspektivische Projektion
    EMD_TWOD,        // Welt- = Screenkoordinate
    EMD_ORTHOGONAL   // Orthogonale Projektion
} ZFXENGINEMODE;
```

*Die neuen
Funktionen*

Was genau hinter den einzelnen Funktionen steht, das klären wir bei der Implementierung in den folgenden Unterabschnitten. Bei einigen Funktionen geht es ja bereits aus dem Namen hervor, bei anderen erkennt man nicht sofort, welchen Zweck sie erfüllen sollen. Aber Ungeduld ist ja bekanntlich die Mutter der Dummheit, wie Leonardo sagen würde.



Wer an dieser Stelle mit Begriffen wie Viewmatrix, Viewport, Field Of View (FOV), Viewfrustum, Near und Far Clipping-Plane usw. immer noch nichts anfangen kann, der sollte sich ein Grundlagenbuch zum Thema 3D-Grafik besorgen.

Multiple Stages

*Zooms und
Fischaugen*

Mit multiplen Stages ist nichts anderes gemeint, als dass wir es dem Anwender der ZFXEngine ermöglichen, bis zu vier verschiedene Paare aus Projektionsmatrix und Viewport zusammen zu speichern. Dabei handelt es sich ausschließlich um perspektivische Projektionsmatrizen, denn wir benötigen nur eine feste orthogonale Projektionsmatrix. Bei der perspektivischen Projektion verwendet man zunächst auch nur eine feste Projektionsmatrix. Sobald man allerdings mit Spezialeffekten arbeitet, möchte man eventuell noch andere Matrizen verwenden, beispielsweise eine für ein 12x-Zielfernrohr oder eine für einen Fischaugeneffekt. Hier bieten wir dem Anwender die Möglichkeit, bis zu vier solcher Matrizen zwischenspeichern und auf Abruf ohne Berechnung bereit zu haben.

Normalerweise sollte also die nullte Stage in unserer Engine immer die normale, perspektivische Projektionsmatrix beinhalten. Die erste Stage könnte dann den 8x-Zoom eines Fernglases im Speicher halten, während die zweite Stage den 12x-Zoom eines Zielfernrohres im Speicher hält. Da wir eine entsprechende Stage auch immer gleich mit einem Viewport verbinden, ist es auch komfortabel möglich, beispielsweise die Ansicht des Fernglases oder des Zielfernrohres nur in einem bestimmten Bereich des Bildschirms anzeigen zu lassen – etwa als zusätzliches Fenster in der normalen 3D-Ansicht der Szene.

*Fenster im
Fenster*

Um die entsprechend berechneten Daten aber auch im Speicher halten zu können, benötigen wir logischerweise Variablen, in diesem Fall Attribute der Klasse `ZFXD3D`. Letzten Endes soll der Anwender der `ZFXEngine` überhaupt nichts mit Matrizen für View und Projektion zu tun haben. Er soll lediglich die Winkel des Field Of View angeben, den Rest soll die Engine allein machen. Dies ist an dieser Stelle ein guter Kompromiss zwischen Simplität und Flexibilität. Die Matrizen interessieren den Anwender schließlich nicht, aber er möchte nicht auf eine konstante Einstellung der Engine beschränkt sein. Langer Rede kurzer Sinn: Hier sind die neuen privaten Attribute der Klasse `ZFXD3D`:

Neue Attribute

```
D3DMATRIX m mView2D,      // Viewmatrix 2D
          m mView3D,      // Viewmatrix 3D
          m mProj2D,      // Projektion orthog.
          m mProjP[4],    // Projektion persp.
          m mProjO[4],    // Projektion orthog.
          m mWorld,       // Welttransformation
          m mViewProj;    // Combo-Matrix für 3D
          m mWorldViewProj; // Combo-Matrix für 3D
```



Wie man bereits erkennen kann, gibt es jeweils eine eigene Viewmatrix für die perspektivische und die zweidimensionale Darstellung. Letztere wird dazu verwendet, die Weltkoordinaten von Vertices direkt in Bildschirmkoordinaten angeben zu können. Dazu kommen eine Matrix für orthogonale Projektion über den gesamten Bildschirm sowie die vier Matrizen für unterschiedliche perspektivische Projektion (vier Stages) und vier Matrizen für unterschiedliche orthogonale Projektion. Die Unterschiede der Stages bestehen darin, dass jede der vier Stages einen anderen Viewport (also nur einen Teil des Bildschirms) verwenden kann. Für jeden Viewport brauchen wir eine eigene Projektionsmatrix, da diese die Abmessungen des Viewports verwendet. Die perspektivischen Projektionsmatrizen sind dabei zusätzlich noch vom horizontalen Sichtfeld des Betrachters (FOV) abhängig.

Matrizen-Auflauf

Zu guter Letzt haben wir noch zwei Combo-Matrizen. In diesen beiden Mitgliedern werden wir immer das Produkt der aktuell aktiven perspektivischen Projektionsmatrix und der aktuell aktiven perspektivischen Viewmatrix vorrätig halten, bzw. noch mit der Weltmatrix multiplizieren, und

sie jedes Mal neu berechnen, wenn sich einer der Faktoren ändert. Diese Combo-Matrix benötigen wir später für verschiedene Berechnungen.

Viewports, Viewmatrizen und das Frustum

*Position des
Betrachters*

Beginnen wir unsere Arbeit mit der Erstellung der Viewmatrix. Möchte man eine 3D-Szene am Bildschirm anzeigen, so muss man natürlich die Position des Betrachters angeben, also die Position und die Blickrichtung, von der aus die Szene betrachtet wird. Im Jargon eines Computergrafikers spricht man hier auch von der *Position der Kamera*. Eine solche Kamera dient dazu, den Standpunkt der Betrachtung der Szene zu definieren. Schließlich kann es in einem Computerspiel mehr Betrachter geben als nur den Spieler selbst. Beispielsweise könnte der Spieler am Bildschirm das Geschehen auch durch die *Augen* der von ihm kommandierten Einheiten sehen wollen. Mit Hilfe der folgenden Funktion kann man die Viewmatrix für die *ZFXEngine* festlegen. Das ist immer dann nötig, wenn sich die Position oder Orientierung der Kamera geändert hat.

*Shader hier,
Shader da!*

In unserer Implementierung der Klasse ergänzen wir nun ein Attribut `m_bUseShaders` vom Typ `bool`. Wenn dieses den Wert `false` hat, dann beherrscht die Grafikhardware, auf der die Engine läuft, keine Vertex- oder Pixel-Shader. In diesem Fall müssen wir alle Einstellungen der Transformationsmatrizen für Projektion, View und World über die Direct3D-Funktion `IDirect3DDevice9::SetTransform` vornehmen. Anderenfalls verwenden wir die große Combomatrix unserer Klasse zusammen mit einem Vertex-Shader. Daher müssen wir bei den Device-Einstellungen im Folgenden immer unterscheiden, ob die Engine mit Shadern läuft oder noch konservativ *verbleit tankt*.



```
HRESULT ZFXD3D::SetView3D(const ZFXVector &vcRight,
                          const ZFXVector &vcUp,
                          const ZFXVector &vcDir,
                          const ZFXVector &vcPos) {
    if (!m_bRunning) return E_FAIL;

    m_mView3D._14 = m_mView3D._21 = m_mView3D._34 = 0.0f;
    m_mView3D._44 = 1.0f;

    m_mView3D._11 = vcRight.x;
    m_mView3D._21 = vcRight.y;
    m_mView3D._31 = vcRight.z;
    m_mView3D._41 = - (vcRight*vcPos);

    m_mView3D._12 = vcUp.x;
    m_mView3D._22 = vcUp.y;
    m_mView3D._32 = vcUp.z;
    m_mView3D._42 = - (vcUp*vcPos);
```

```

m_mView3D._13 = vcDir.x;
m_mView3D._23 = vcDir.y;
m_mView3D._33 = vcDir.z;
m_mView3D._43 = - (vcDir*vcPos);

if (!m_bUseShaders) {
    if (FAILED(m_pDevice->SetTransform(D3DTS_VIEW,
                                        &m_mView3D)))
        return ZFX_FAIL;
}

CalcViewProjMatrix();
CalcWorldViewProjMatrix();
return ZFX_OK;
}

```

Wie man erkennen kann, ist die Viewmatrix nicht sehr kompliziert aufgebaut. Die ersten drei Einträge der ersten drei Spalten der Matrix enthalten den Rechts-, Hoch- und Richtungsvektor der Kamera. Diese Vektoren sind die drei orthogonalen Einheitsvektoren, die das lokale Koordinatensystem der Kamera aufspannen. In den ersten drei Einträgen der letzten Zeile der Matrix findet sich das Punktprodukt des jeweiligen Vektors mit dem Positionsvektor der Kamera, während die letzte Spalte der Matrix der einer Einheitsmatrix entspricht. Diese Matrix setzen wir dann durch die Direct3D-Funktion `IDirect3DDevice9::SetTransform` als Viewmatrix fest.

Und so geht's

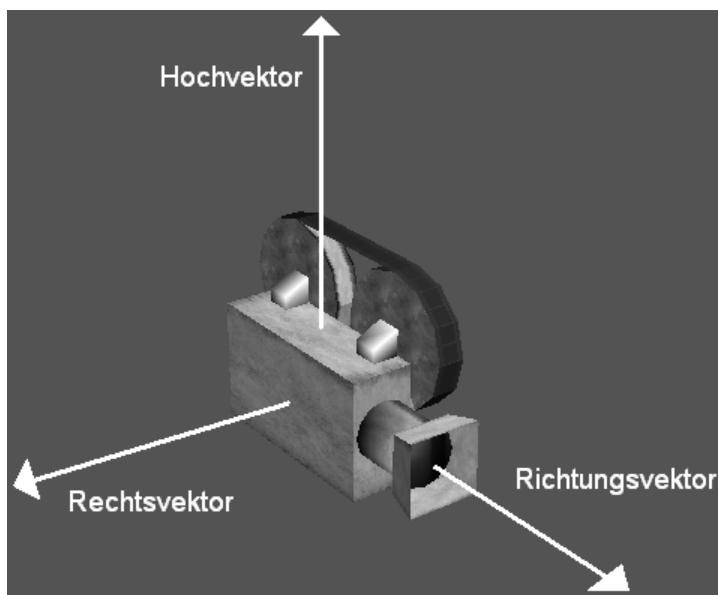


Abbildung 6.1:

Die drei lokalen Achsen der Kamera. Der Rechtsvektor ist der Einheitsvektor der X-Achse, der Hochvektor ist der Einheitsvektor der Y-Achse, und der Richtungsvektor ist der Einheitsvektor der Z-Achse.

Da sich nun unsere Viewmatrix geändert hat, müssen wir die Combomatrix aus Viewmatrix und Projektionsmatrix durch den Aufruf unserer Funktion `ZFXD3D::CalcViewProjMatrix` neu berechnen. Zu dieser Funktion kommen wir später noch, aber es handelt sich hier einfach nur um die Multiplikation zweier Matrizen. Selbiges gilt für den Aufruf von `ZFXD3D::CalcWorldViewProjMatrix`, nur dass wir hier eine Matrix aus dreien zusammen multiplizieren.

**Fixpunkt
anvisieren**

Die eben gezeigte Funktion ist immer dann zu verwenden, wenn man die Position und Ausrichtung der Kamera berechnet hat und nun die Szene durch diese Kameraperspektive betrachten möchte. Der umgekehrte Fall ist der, dass man die Position der Kamera kennt und die Kamera nun auf ein spezielles Objekt ausrichtet, dessen Position man ebenfalls kennt. Die Ausrichtung der Kamera hängt also von der Position des zu betrachtenden Objekts ab. Auf diese Art und Weise kann man beispielsweise eine Kamera ein bestimmtes Objekt der Szene fixieren lassen, um zum Beispiel eine 3rd-Person-Kamera zu realisieren.

**Zwischen-
geschaltete
Vorbereitung**

Natürlich sprechen wir hier von der berühmten *LookAt*-Funktion. Diese hat ihren Namen dadurch, dass sie dazu dient, die Kamera auf ein bestimmtes Objekt schauen zu lassen. Letzten Endes müssen wir aber die Viewmatrix genauso setzen wie bisher auch, nämlich aus dem Rechts-, Hoch-, Richtungs- und Positionsvektor der Kamera. Bevor wir dies aber tun können, müssen wir die drei erstgenannten Vektoren erst einmal in einer Zwischenrechnung herausfinden. Bei dem Richtungsvektor ist das nicht weiter problematisch, denn für eine *LookAt*-Funktion geben wir ja die Position der Kamera und des zu fixierenden Objekts an. Und den Richtungsvektor zwischen zwei Punkten berechnet man ja bekanntlich nach der guten, alten Formel »Endpunkt minus Startpunkt« und normalisiert das Ergebnis.

**Hoch- und
Rechtsvektor
errechnen**

Zusätzlich muss man der *LookAt*-Funktion noch den Hochvektor der Welt an der Position der Kamera angeben. Aus diesem können wir den Hochvektor für die Kamera mit Blick auf den Fixpunkt errechnen. Dazu bilden wir das Punktprodukt zwischen dem Richtungsvektor und dem Welt-Hochvektor. Dieser gibt ja den Winkel zwischen den beiden Vektoren an bzw. 0.0f, falls sie orthogonal zueinander sind. Mit diesem Wert des Punktproduktes multiplizieren wir den Richtungsvektor der Kamera und ziehen das Ergebnis vom Welt-Hochvektor ab. Im dem Fall, dass der Richtungsvektor nicht schon rechtwinklig zu dem Welthochvektor ist, skalieren wir dadurch sozusagen den Richtungsvektor entsprechend der Abweichung von der Orthogonalität und erreichen durch die Verrechnung mit dem Welt-Hochvektor genau einen Vektor, der orthogonal zu dem Richtungsvektor ist und damit den Hochvektor der Kamera bilden kann.

Führt dies zu keinem sinnvollen Ergebnis, weil die Länge des errechneten Vektors zu kurz ist, so verwenden wir anstelle der Welt-Hochachse entweder die Y-Achse oder auch die Z-Achse. Den Rechtsvektor zu dem Rich-

tungs- und Hochvektor der Kamera können wir dann natürlich ganz einfach durch das Kreuzprodukt über den beiden Vektoren bilden.

```
HRESULT ZFXD3D::SetViewLookAt(const ZFXVector &vcPos,
                               const ZFXVector &vcPoint,
                               const ZFXVector &vcWorldUp)
```



```
{
    ZFXVector vcDir, vcTemp, vcUp;

    vcDir = vcPoint - vcPos;
    vcDir.Normalize();

    // berechne Up-Vektor
    float fDot = vcWorldUp * vcDir;
    vcTemp = vcDir * fDot;
    vcUp = vcWorldUp - vcTemp;
    float fL = vcUp.GetLength();

    // falls zu kurz, nimm die y-Achse
    if (fL < 1e-6f) {
        ZFXVector vcY;
        vcY.Set(0.0f, 1.0f, 0.0f);

        vcTemp = vcDir * vcDir.y;
        vcUp = vcY - vcTemp;

        fL = vcUp.GetLength();

        // falls immer noch zu kurz, dann die z-Achse
        if (fL < 1e-6f) {
            vcY.Set(0.0f, 0.0f, 1.0f);

            vcTemp = vcDir * vcDir.z;
            vcUp = vcY - vcTemp;

            fL = vcUp.GetLength();

            // hier hilft nichts mehr
            if (fL < 1e-6f) return ZFX_FAIL;
        }
    }
    vcUp /= fL;

    // Rechtsvektor erstellen
    ZFXVector vcRight;
    vcRight.Cross(vcUp, vcDir);

    // erstelle und aktiviere die endgültige Matrix
    return SetView3D(vcRight, vcUp, vcDir, vcPos);
}
```

Abschließend rufen wir die Funktion `ZFXD3D::SetView3D` auf, um die Viewmatrix aus den eben hart erkämpften Vektoren zu erstellen.

Viewfrustum

Eines der wichtigsten Dinge in Bezug auf die Sicht der 3D-Szene ist das so genannte *Viewfrustum*. Dieses ist deshalb so wichtig, weil es das Sichtfeld des Betrachters exakt gegen den Monitor abgrenzt. Ebenso wie ein Mensch, der durch einen Fensterrahmen schaut, begrenzt ja auch der Bildschirm das Sichtfeld des Betrachters auf die 3D-Szene in Form eines liegenden Pyramidenstumpfes mit viereckigem Fuß (Frustum). Mit Hilfe des Viewfrustums können wir daher sehr einfach und schnell berechnen, welche Objekte überhaupt auf dem Bildschirm sichtbar sind und berechnet werden müssen. Alle anderen Objekte können aus der weiteren Berechnung und dem Rendervorgang ausgeschlossen werden. Das nennt man auch *Objekt-Culling*.

Konstruktion des Frustums

Die Größe und Form des Frustums ist natürlich von der aktiven Projektionsmatrix abhängig, die Koordinatenverschiebung des Frustums hängt von der aktiven Viewmatrix ab. Folglich können wir das Viewfrustum aus der Combomatrix aus der Projektions- und der Viewmatrix berechnen. Das Frustum speichern wir jedoch nicht in Form eines 3D-Modells einzelner Punkte, sondern lediglich durch die sechs Ebenen, die den Bereich des Frustums begrenzen. Die folgende Funktion nimmt einen Pointer auf ein Array aus sechs `ZFXPlane`-Objekten auf und speichert darin die sechs Ebenen des aktuellen Frustums.



```
HRESULT ZFXD3D::GetFrustum(ZFXPlane *p) {
    // Linke Plane
    p[0].m_vcN.x = -(m_mViewProj._14 + m_mViewProj._11);
    p[0].m_vcN.y = -(m_mViewProj._24 + m_mViewProj._21);
    p[0].m_vcN.z = -(m_mViewProj._34 + m_mViewProj._31);
    p[0].m_fd    = -(m_mViewProj._44 + m_mViewProj._41);

    // Rechte Plane
    p[1].m_vcN.x = -(m_mViewProj._14 - m_mViewProj._11);
    p[1].m_vcN.y = -(m_mViewProj._24 - m_mViewProj._21);
    p[1].m_vcN.z = -(m_mViewProj._34 - m_mViewProj._31);
    p[1].m_fd    = -(m_mViewProj._44 - m_mViewProj._41);

    // Top-Plane
    p[2].m_vcN.x = -(m_mViewProj._14 - m_mViewProj._12);
    p[2].m_vcN.y = -(m_mViewProj._24 - m_mViewProj._22);
    p[2].m_vcN.z = -(m_mViewProj._34 - m_mViewProj._32);
    p[2].m_fd    = -(m_mViewProj._44 - m_mViewProj._42);

    // Boden-Plane
    p[3].m_vcN.x = -(m_mViewProj._14 + m_mViewProj._12);
    p[3].m_vcN.y = -(m_mViewProj._24 + m_mViewProj._22);
```

```

p[3].m_vcN.z = -(m_mViewProj._34 + m_mViewProj._32);
p[3].m_fD    = -(m_mViewProj._44 + m_mViewProj._42);

// Near-Plane
p[4].m_vcN.x = -m_mViewProj._13;
p[4].m_vcN.y = -m_mViewProj._23;
p[4].m_vcN.z = -m_mViewProj._33;
p[4].m_fD    = -m_mViewProj._43;

// Far-Plane
p[5].m_vcN.x = -(m_mViewProj._14 - m_mViewProj._13);
p[5].m_vcN.y = -(m_mViewProj._24 - m_mViewProj._23);
p[5].m_vcN.z = -(m_mViewProj._34 - m_mViewProj._33);
p[5].m_fD    = -(m_mViewProj._44 - m_mViewProj._43);

// normalisiere Normalenvektoren
for (int i=0;i<6;i++) {
    float fL = p[i].m_vcN.GetLength();
    p[i].m_vcN /= fL;
    p[i].m_fD  /= fL;
}
return ZFX_OK;
}

```

Da das Viewfrustum ja von der Projektions- und der Viewmatrix abhängt, ist es nun klar, dass wir es jedes Mal neu berechnen müssen, wenn sich eine der beiden Matrizen ändert. Diese logische Erkenntnis bleibt dem Anwender der ZFXEngine überlassen. Wenn er sich einmal ein Frustum von der Engine hat geben lassen, so ist dieses nur so lange korrekt, wie sich die beiden Matrizen nicht ändern. Wenn sich beispielsweise die Kamera bewegt oder gedreht hat, dann ist das Frustum nicht mehr gültig. Aus diesem Grund berechnen wir die Combomatrix jedes Mal neu, damit diese Funktion hier von der Gültigkeit der Combomatrix zu jedem Zeitpunkt ausgehen kann.



Zwei entscheidende Werte für die Sicht einer 3D-Engine sind die Werte für die Near Plane und die Far Plane. Diese beiden Ausdrücke bezeichnen die Entfernung der beiden Ebenen des Frustums zum Betrachter, die parallel zu der Projektionsfläche (dem Bildschirm) verlaufen. Da das Frustum ja den Sichtbereich des Betrachters begrenzt, definieren die beiden Werte damit die Entfernungen, ab der Objekte im Sichtbereich liegen bzw. ab der sie wieder aus dem Sichtbereich verschwinden. Die Near Plane definiert dabei die Entfernung, die ein Objekt *mindestens* von der Kamera haben muss, um potenziell sichtbar zu sein. Ein negativer Entfernungswert sagt dabei aus, dass ein Objekt hinter der Kamera und damit unsichtbar ist. Jedoch gerät ein Objekt nicht automatisch ab einer Entfernung von 0 in den Sichtbereich. Aufgrund der Berechnungen für den Depth-Buffer würde dies Probleme verursachen,

*Near und Far
Clipping-Plane*

daher wird die Near Plane normalerweise auf einen Wert größer als 0 gesetzt, beispielsweise 0.1f. Die Entfernung der Far Clipping-Plane hingegen entspricht der maximalen Sichtweite der Kamera. Alle Objekte, die weiter entfernt sind, werden nicht mehr angezeigt.

Rattenschwanz

Die folgende Funktion sorgt dafür, dass die beiden Werte für die Near und die Far Plane gesetzt werden können. Dadurch lässt sich beispielsweise die effektive Sichtweite der Kamera verändern. Leider ist es aber so, dass diese beiden Werte an verschiedenen Stellen der 3D-Pipeline noch benötigt werden, insbesondere aber bei den Projektionsmatrizen. Das Ändern dieser beiden Werte zieht also einen Rattenschwanz an Änderungen in allen Projektionsmatrizen nach sich, die wir verwenden.



```
void ZFXD3D::SetClippingPlanes(float fNear, float fFar)
{
    m_fNear = fNear;
    m_fFar = fFar;

    if (m_fNear <= 0.0f) m_fNear = 0.01f;
    if (m_fFar <= 1.0f) m_fFar = 1.00f;

    if (m_fNear >= m_fFar) {
        m_fNear = m_fFar;
        m_fFar = m_fNear + 1.0f;
    }

    // 2D-Matrizen anpassen
    Prepare2D();

    // orthogonale Projektionsmatrix anpassen
    float Q = 1.0f / (m_fFar - m_fNear);
    float X = -Q * m_fNear;
    m_mProjO[0]._33 = m_mProjO[1]._33 = Q;
    m_mProjO[2]._33 = m_mProjO[3]._33 = Q;
    m_mProjO[0]._43 = m_mProjO[1]._43 = X;
    m_mProjO[2]._43 = m_mProjO[3]._43 = X;

    // perspektivische Projektionsmatrix anpassen
    Q *= m_fFar;
    X = -Q * m_fNear;
    m_mProjP[0]._33 = m_mProjP[1]._33 = Q;
    m_mProjP[2]._33 = m_mProjP[3]._33 = Q;
    m_mProjP[0]._43 = m_mProjP[1]._43 = X;
    m_mProjP[2]._43 = m_mProjP[3]._43 = X;
}
```

Abhängige Werte

Wie ihr seht, ist es nicht damit getan, einfach die beiden Werte auf sinnvolle Zahlen zu beschränken. Der jeweils dritte Eintrag der dritten und vierten Zeile der Projektionsmatrizen hängt von den beiden Werten der Near und Far Clipping-Plane ab, daher müssen sie für alle Matrizen aller vier Stages

neu berechnet werden. Gleiches gilt für die Projektionsmatrix und die View-matrix, die wir zum Rendern von 2D verwenden. Für die entsprechenden Änderungen schreiben wir uns hier die Funktion `ZFXD3D::Prepare2D`, zu der wir auch gleich kommen. Jetzt schauen wir uns erst mal die Projektion genauer an. In der Funktion haben wir ja schon ein paar Einträge der Projektionsmatrizen gesehen. Nun interessiert uns natürlich, wie die kompletten Projektionsmatrizen aussehen.

Orthogonale Projektion

Aus dem Kunstunterricht kennt wohl jeder ein paar Techniken, um dreidimensional wirkende Bilder zu zeichnen. Ich sage nur Fluchtpunkt oder Fünfundvierzig-Grad-Winkel und halbe Länge. Das Geheimnis hinter der Möglichkeit, auf einer zweidimensionalen Fläche dreidimensional wirkende Bilder entstehen zu lassen, heißt natürlich *Projektion*. Man projiziert dreidimensionale Informationen in ein zweidimensionales Bezugssystem, wobei eine Dimension auf der Strecke bleibt. Je nachdem, wie man diese wegfallende Information bei der Projektion interpretiert, entstehen verschiedene Typen der Projektion. Es gibt sicherlich viele verschiedene Verfahren, doch wir halten ja hier keine Vorlesung über lineare Algebra. Uns interessieren nur die beiden wichtigsten Möglichkeiten, nämlich die *orthogonale* Projektion und die *perspektivische* Projektion.

*Kunst und lineare
Algebra*

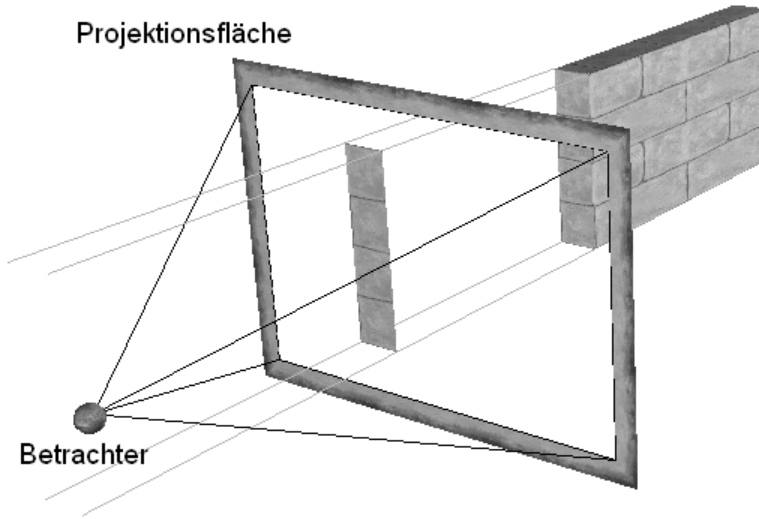
Damit sind wir schon bei der orthogonalen Projektion gelandet, die manchmal auch als *parallele* Projektion bezeichnet wird. Bei dieser Art der Projektion verzichtet man einfach großzügig darauf, die wegfallende Information, also die Tiefe eines Objekts, mit einzubeziehen. Daher sind im 3D-Raum parallel verlaufende Linien auch in der 2D-Darstellung nach der Projektion auf der Projektionsfläche (beispielsweise dem Bildschirm) weiterhin parallel. Sie erscheinen nicht schräg nach hinten aufeinander zulaufend wie beispielsweise bei der Methode des Fluchtpunktes. Abbildung 6.2 zeigt ein Beispiel dafür, wie ein Objekt parallel projiziert wird. Dabei verlaufen die Projektionslinien von den zu projizierenden Punkten aus genau orthogonal zur Projektionsebene. Daher kann man in der projizierten Darstellung auch nur die Frontseite der Mauer sehen (vergleiche mit Abbildung 6.3). Korrekterweise bezeichnet die parallele Projektion aber eine Situation, in der die Projektionsfläche nicht notwendigerweise orthogonal zu den Projektionslinien verlaufen muss. Die orthogonale Projektion ist also tatsächlich ein Spezialfall der parallelen Projektion.

Parallelität

Die Entfernung, die ein zu projizierendes Objekt zur Kamera hat, ist dabei vollkommen egal. Auch mit zunehmender Entfernung zur Position des Betrachters werden die Objekte nicht perspektivisch kleiner. Sie behalten ihre ursprüngliche Größe. Warum sollte so eine Art der Projektion sinnvoll sein? Schließlich bildet sie die Realität nicht besonders gut nach und ist damit augenscheinlich eine sehr schlechte Wahl für die Computergrafik.

CAD & Co.

Abbildung 6.2:
Orthogonale Projektion eines Mauerstücks aus dem 3D-Raum auf eine zweidimensionale Projektionsfläche



Doch halt! Wer sagt denn, dass wir in der Computergrafik immer eine realistische Darstellung der Szene anstreben? Nehmen wir einmal als Beispiel CAD¹-Programme oder 3D-Modellierungsprogramme. Diese haben in der Regel nur ein Fenster mit einer realistischen 3D-Ansicht der Szene. Dazu kommen aber drei Fenster, mit denen man die Szene als Drahtgittermodell von rechts, vorne und oben betrachten kann, so wie man sie auf einem Skizzenblock entwerfen würde. Und genau diese Ansichten wirken wie eine 2D-Skizze der Szene, denn auch hier fehlt die perspektivische Korrektheit und Tiefe in der Projektion. Natürlich ist es nun keine Überraschung mehr, dass man solche Ansichten auf eine Szene durch orthogonale Projektion erzeugt.

Die zweite Dimension

Wir möchten aber durch die orthogonale Projektion noch einen anderen Effekt erreichen, und zwar möchten wir neben der reinen orthogonalen Darstellung einer 3D-Szene noch eine andere Betriebsart für unsere Engine haben. Bei der normalen orthogonalen Projektion verwendet man natürlich die Viewmatrix so wie bisher. Wir haben also immer noch eine Viewmatrix, die die Position der Kamera im 3D-Raum für eine normale Projektion (perspektivisch oder orthogonal) beinhaltet. Durch eine anders aufgebaute, spezielle Viewmatrix kann man aber auch einen weiteren nützlichen Effekt in Kombination mit der orthogonalen Projektion erreichen. Durch diese Art der Viewmatrix kann man die Koordinaten der Vertices direkt in Bildschirmkoordinaten angeben. Die folgende Funktion erstellt die entsprechende Viewmatrix und eine orthogonale Projektionsmatrix:

¹ CAD steht für *Computer Aided Design* und ist die Bezeichnung für Programme, mit denen Ingenieure oder Architekten Maschinenteile oder Grundrisse entwerfen.

```

void ZFXD3D::Prepare2D(void) {
    // setze Matrizen auf Einheitsmatrix
    memset(&m_mProj2D, 0, sizeof(float)*16);
    memset(&m_mView2D, 0, sizeof(float)*16);
    m_mView2D._11 = m_mView2D._33 = m_mView2D._44 = 1.0f;

    // orthogonale Projektionsmatrix
    m_mProj2D._11 = 2.0f/(float)m_dwWidth;
    m_mProj2D._22 = 2.0f/(float)m_dwHeight;
    m_mProj2D._33 = 1.0f/(m_fFar-m_fNear);
    m_mProj2D._43 = -m_fNear*(1.0f/(m_fFar-m_fNear));
    m_mProj2D._44 = 1.0f;

    // 2D-Viewmatrix
    float tx, ty, tz;
    tx = -((int)m_dwWidth) + m_dwWidth * 0.5f;
    ty = m_dwHeight - m_dwHeight * 0.5f;
    tz = m_fNear + 0.1f;

    m_mView2D._22 = -1.0f;
    m_mView2D._41 = tx;
    m_mView2D._42 = ty;
    m_mView2D._43 = tz;
}

```



Wenn wir jetzt die beiden Matrizen `m_mView2D` und `m_mProj2D` für das Device aktivieren, dann können wir für die x- und y-Koordinaten von Vertices gleich Bildschirmkoordinaten angeben. Auf diese Art und Weise können wir durch echte 3D-Polygone auch problemlos zweidimensionale Grafik durch direktes Angeben von Bildschirmkoordinaten für die Polygone rendern, beispielsweise Instrumentenanzeigen im HUD (Heads-Up-Display) des Spielers. In dieser Funktion erkennen wir auch, wie eine orthogonale Projektionsmatrix aufgebaut ist. Die hier verwendete Matrix `m_mProj2D` bezieht sich dabei auf den gesamten Bildschirm, unabhängig von gesetzten Viewports. In der Funktion `ZFXD3D::InitStage` werden wir nachher noch die orthogonalen Projektionsmatrizen für die vier Stages einstellen. Der einzige Unterschied dieser Matrizen ist dabei die Verwendung der Höhe und Breite des jeweiligen Viewports der entsprechenden Stage.

Funktionsweise

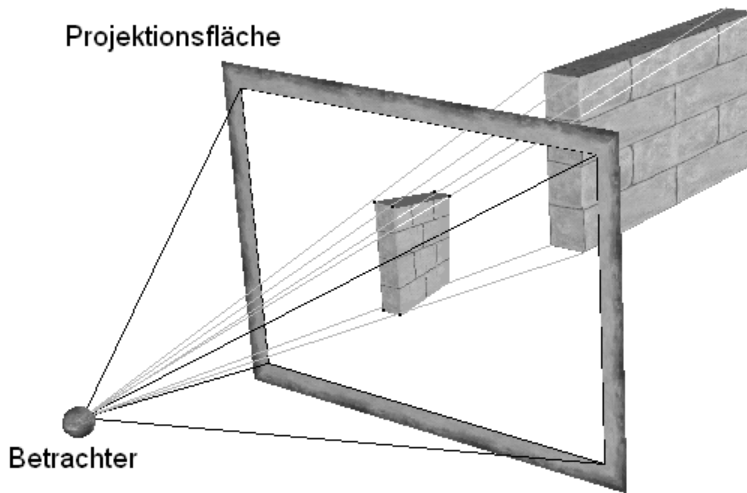
Perspektivische Projektion

Die wohl wichtigste Art der Projektion für die Anwendung in der Computergrafik ist aber natürlich die perspektivische Projektion. Dabei wird die dritte, bei der orthogonalen Projektion wegfallende Information, nämlich die Entfernung eines projizierten Punktes zur Projektionsebene, in die Projektion mit einbezogen. Weiter entfernt gelegene Objekte erscheinen deshalb perspektivisch korrekt kleiner als die näher gelegenen. Hinter der Projektionsfläche gelegene Objekte werden entsprechend auf Werte projiziert, mit

Unser Favorit

denen sie nicht auf der Projektionsfläche erscheinen. Damit sind auch im 3D-Raum parallel verlaufende Linien nach der perspektivischen Projektion nicht mehr parallel, sondern laufen auf den Fluchtpunkt des Bildes zu. Dies verdeutlicht Abbildung 6.3, die dieselbe Szene wie Abbildung 6.2 zeigt. Diesmal allerdings wird perspektivisch projiziert, den Fluchtpunkt bildet dabei die Position des Betrachters. Die Projektionslinien der Punkte des 3D-Objekts verlaufen nun nicht mehr orthogonal zur Projektionsfläche. Daher kann man auch in der 2D-Darstellung auf dem Bildschirm noch die Seitenfläche der Mauer erkennen und nicht nur die Front.

Abbildung 6.3:
Perspektivische
Projektion eines
Mauerstücks aus
dem 3D-Raum auf
eine zweidimen-
sionale Projektions-
fläche



Direct3D- Projektion

Da die Einstellung einer Projektionsmatrix zu den absoluten Grundlagen gehört, liste ich hier einfach nur die Funktion auf. Wer den mathematischen Background vertiefen möchte, der kann dies in der DirectX-Dokumentation nachlesen.



```
HRESULT ZFXD3D::CalcPerspProjMatrix(float fFOV,
                                     float fAspect,
                                     D3DMATRIX *m) {
    if(fabs(m_fFar - m_fNear) < 0.01f)
        return ZFX_FAIL;

    float sinFOV2 = sinf(fFOV/2);

    if(fabs(sinFOV2) < 0.01f)
        return ZFX_FAIL;

    float cosFOV2 = cosf(fFOV/2);

    float w = fAspect * (cosFOV2 / sinFOV2);
```



```

float h = 1.0f * (cosFOV2 / sinFOV2);
float Q = m_fFar / (m_fFar - m_fNear);

memset(m, 0, sizeof(D3DMATRIX));
(*m)._11 = w;
(*m)._22 = h;
(*m)._33 = Q;
(*m)._34 = 1.0f;
(*m)._43 = -Q*m_fNear;
return ZFX_OK;
} // CalcPerspProjMatrix

```

Für die Einstellung einer Projektionsmatrix ist nur der horizontale FOV- (Field Of View-)Winkel und die Aspekt-Ratio (Verhältnis von Höhe zu Breite des Viewports) entscheidend. Hier kann man auch noch einmal die Abhängigkeit der perspektivischen Projektion von den Werten der Near und Far Clipping-Plane erkennen.

Parameter

Ebenfalls trivial ist die Funktion zur Berechnung der Combomatrix aus der View- und der Projektionsmatrix. Wir nehmen einfach die beiden zurzeit aktiven Matrizen und multiplizieren sie zusammen. Dazu müssen wir lediglich unterscheiden, welcher Modus aktiv ist und gegebenenfalls welche Stage. Um unsere eigenen 3D-Mathe-Funktionen verwenden zu können, casten wir die Matrizen in unseren eigenen Matrix-Typ ZFXMatrix.

*Combomatrix,
die Erste*

```

void ZFXD3D::CalcViewProjMatrix(void) {
    ZFXMatrix *pA;
    ZFXMatrix *pB;

    // 2D, perspektivisch oder orthogonal
    if (m_Mode == EMD_TWOD) {
        pA = (ZFXMatrix*)&m_mProj2D;
        pB = (ZFXMatrix*)&m_mView2D;
    }
    else {
        pB = (ZFXMatrix*)&m_mView3D;

        if (m_Mode == EMD_PERSPECTIVE)
            pA = (ZFXMatrix*)&(m_mProjP[m_nStage]);
        else
            pA = (ZFXMatrix*)&(m_mProjO[m_nStage]);
    }
    ZFXMatrix *pM = (ZFXMatrix*)&m_mViewProj;
    (*pM) = (*pA) * (*pB);
} // CalcViewProjMatrix

```

}
LIST

Ganz analog berechnen wir noch eine zweite Combomatrix. Diesmal nehmen wir auch die aktuelle Weltmatrix noch in die Combo mit auf. Diese Megamatrix brauchen wir, wenn wir selbst 3D-Vektoren transformieren

*Combomatrix,
die Zweite*

wollen, beispielsweise über einen Vertex-Shader. Daher müssen wir auch diese Combomatrix immer aktuell halten, sobald sich eine der drei aktiven involvierten Matrizen ändert.



```
void ZFXD3D::CalcWorldViewProjMatrix(void) {
    ZFXMatrix *pProj;
    ZFXMatrix *pView;
    ZFXMatrix *pWorld;

    pWorld = (ZFXMatrix*)&m_mWorld;

    // 2D, perspektivisch oder orthogonal
    if (m_Mode == EMD_TWOD) {
        pProj = (ZFXMatrix*)&m_mProj2D;
        pView = (ZFXMatrix*)&m_mView2D;
    }
    else {
        pView = (ZFXMatrix*)&m_mView3D;

        if (m_Mode == EMD_PERSPECTIVE)
            pProj = (ZFXMatrix*)&(m_mProjP[m_nStage]);
        else
            pProj = (ZFXMatrix*)&(m_mProjO[m_nStage]);
    }

    ZFXMatrix *pCombo = (ZFXMatrix*)&m_mWorldViewProj;
    (*pCombo) = ((*pWorld) * (*pView)) * (*pProj);
} // CalcViewProjMatrix
```

Aktivieren von Sicht und Projektion

*Nach dem
Vergnügen die
Arbeit!*

Jetzt haben wir eine ganze Reihe von Funktionen entwickelt, mit deren Hilfe wir verschiedene Projektionsmatrizen berechnen und die Clipping-Planes dynamisch verändern können. Jetzt benötigen wir nur noch eine Funktion, mit der wir zwischen den verschiedenen Betriebsmodi der Engine umschalten können. Das ist eine Aufgabe, die keinerlei mathematische Anforderungen an uns stellt, sondern ein wenig logisches Denken erfordert. Alle Teile der Engine sind so miteinander verwoben, dass wir nicht vergessen dürfen, alle abhängigen Teile ebenfalls umzuschalten oder neu zu berechnen, wenn sich der Modus der Engine ändert.

Konsistenz

Ich zeige zuerst die Funktion, und dann reden wir darüber. Eines noch vorweg: In der Funktion findet sich auch ein Aufruf `ForcedFlushAll()`. Diese Funktion werden wir erst später entwickeln, aber ich kann ja ihren Grund schon mal verraten: Später werden wir Polygone cachen, also nicht direkt beim Senden an unsere DLL rendern, sondern zum Sortieren in der DLL zwischenspeichern. Die Render-Aufrufe sind aber natürlich immer im Kontext des aktuell aktiven Modus der Engine zu sehen. Wenn wir also bei-

spielsweise von perspektivischer Projektion zu orthogonaler Projektion umschalten, müssen wir zuerst alle Polygone im Zwischenspeicher unseres Devices tatsächlich rendern, weil sie sonst anders projiziert werden, als der Anwender dies beim Render-Aufruf beabsichtigt hat. Und nichts anderes macht diese Funktion.

```
HRESULT ZFXD3D::SetMode(ZFXENGINEMODE Mode, int nStage)
```

```
{
    D3DVIEWPORT9 d3dVP;

    if (!m_bRunning) return E_FAIL;
    if ((nStage > 3) || (nStage < 0)) nStage=0;

    if (m_Mode != Mode)
        m_Mode = Mode;

    // fundamentale Änderungen, also alles raus
    m_pVertexMan->ForcedFlushAll();

    // falls 2D-Modus, setze entsprechende Matrizen
    if (Mode==EMD_TWOD) {
        d3dVP.X      = 0;
        d3dVP.Y      = 0;
        d3dVP.Width  = m_dwWidth;
        d3dVP.Height = m_dwHeight;
        d3dVP.MinZ   = 0.0f;
        d3dVP.MaxZ   = 1.0f;

        if (FAILED(m_pDevice->SetViewport(&d3dVP)))
            return ZFX_FAIL;

        if (!m_bUseShaders) {
            if (FAILED(m_pDevice->SetTransform(
                D3DTS_PROJECTION,
                &m_Proj2D)))
                return ZFX_FAIL;

            if (FAILED(m_pDevice->SetTransform(
                D3DTS_VIEW,
                &m_View2D)))
                return ZFX_FAIL;
        }
    }

    // perspektivische oder orthogonale Projektion
    else {
        m_nStage = nStage;

        // set viewport
        d3dVP.X      = m_VP[nStage].X;
```



```

d3dVP.Y      = m_VP[nStage].Y;
d3dVP.Width  = m_VP[nStage].Width;
d3dVP.Height = m_VP[nStage].Height;
d3dVP.MinZ   = 0.0f;
d3dVP.MaxZ   = 1.0f;

if (FAILED(m_pDevice->SetViewport(&d3dVP)))
    return ZFX_FAIL;

if (!m_bUseShaders) {
    if (FAILED(m_pDevice->SetTransform(
                                                D3DTS_VIEW,
                                                &m_mView3D)))
        return ZFX_FAIL;

    if (m_Mode == EMD_PERSPECTIVE) {
        if (FAILED(m_pDevice->SetTransform(
                                                D3DTS_PROJECTION,
                                                &m_mProjP[nStage])))
            return ZFX_FAIL;
    }
    else { // EMD_ORTHOGONAL
        if (FAILED(m_pDevice->SetTransform(
                                                D3DTS_PROJECTION,
                                                &m_mProjO[nStage])))
            return ZFX_FAIL;
    }
}
CalcViewProjMatrix();
CalcWorldViewProjMatrix();
}
return ZFX_OK;
} // SetMode

```

Eigenlob Wie ihr seht, macht sich unsere gründliche Vorarbeit hier bezahlt. Wir müssen lediglich die verschiedenen Modi unterscheiden, die zur Verfügung stehen. Anhand dieser Modi wählen wir dann einen entsprechenden Viewport aus, aktivieren diesen für das Direct3D-Device und setzen die entsprechende Projektions- und Viewmatrix. Eigentlich ein Kinderspiel, man muss nur beachten, jeweils auch die richtige Stage zu verwenden. Für den reinen 2D-Modus ist die Stage aber irrelevant, und als Viewport wird der gesamte Pixel-Buffer verwendet.

Wir dürfen aber nicht vergessen, dass wir nun die aktiven Matrizen geändert haben. Folglich müssen wir nun auch die kombinierte View- und Projektionsmatrix neu berechnen lassen. Das ist schon das ganze Geheimnis, wie wir in unserer Engine zwischen den verschiedenen Modi umschalten können.

Eine Sache hätte ich fast vergessen: Natürlich müssen wir auch eine Möglichkeit haben, die verschiedenen Stages unserer Engine zu initialisieren. Dazu schreiben wir eine Funktion, mit der wir alle vier Stages der beiden 3D-Modi (perspektivisch und orthogonal) einstellen können. Zu jeder Stage gehört dabei ein Field of View (horizontaler Sichtwinkel) und ein Viewport, der angibt, welcher Teil des Pixel-Buffers berendert werden soll.

Initialisierung

```
HRESULT ZFXD3D::InitStage(float fFOV, ZFXVIEWPORT
                          *pView, int nStage) {
    float fAspect;
    bool bOwnRect=false;

    if (!pView) {
        ZFXVIEWPORT vpOwn = { 0, 0, m_dwWidth,
                               m_dwHeight};
        memcpy(&m_VP[nStage], &vpOwn, sizeof(RECT));
    }
    else
        memcpy(&m_VP[nStage], pView, sizeof(RECT));

    if ( (nStage>3) || (nStage<0) ) nStage=0;

    fAspect = ((float)(m_VP[nStage].Height)) /
              (m_VP[nStage].Width);

    // perspektivische Projektionsmatrix
    if (FAILED(this->CalcPerspProjMatrix(fFOV, fAspect,
                                         &m_mProjP[nStage])))
        return ZFX_FAIL;

    // orthogonale Projektionsmatrix
    memset(&m_mProjO[nStage], 0, sizeof(float)*16);
    m_mProjO[nStage]._11 = 2.0f/m_VP[nStage].Width;
    m_mProjO[nStage]._22 = 2.0f/m_VP[nStage].Height;
    m_mProjO[nStage]._33 = 1.0f/(m_fFar-m_fNear);
    m_mProjO[nStage]._43 = - m_fNear *
                          m_mProjO[nStage]._33;
    m_mProjO[nStage]._44 = 1.0f;

    return ZFX_OK;
} // InitStage
```

Diese Funktion kann übrigens jederzeit aufgerufen werden, nicht nur beim Start des Programms. Sie trägt den Änderungen in einer Stage entsprechend Rechnung und kalkuliert alle abhängigen Werte der Projektionsmatrizen automatisch mit. Wir können also die Einstellungen einer Stage zu jedem beliebigen Zeitpunkt der Programmausführung durch diesen einen Funktionsaufruf ändern.

Allzeit bereit!

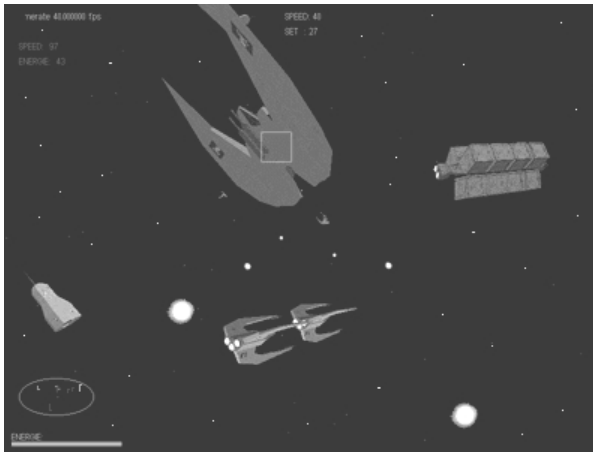
Koordinatenumrechnung 2D zu 3D und zurück

3D-Pipeline zu Fuß

Erstaunlicherweise ist es wesentlich einfacher, die 3D-Pipeline zu Fuß durchzurechnen, als den umgekehrten Weg zu beschreiten. Das bedeutet: Es ist einfacher, einen Punkt aus dem 3D-Raum in den 2D-Raum umzurechnen. Normalerweise führt die Grafikkarte diese Berechnung für uns durch. Allerdings kann es auch sein, dass wir einmal einen oder mehrere Punkte von Hand transformieren müssen, z. B. wenn wir auf dem HUD des Spielers eine Zielmarkierung anzeigen möchten, die immer über einem bestimmten Objekt der virtuellen Welt liegt. Das kennt man ja aus Flugsimulatoren oder aus Weltraumsimulatoren.

Abbildung 6.4:

Der Screenshot aus Wing Captain 2.0 zeigt eine Zielmarkierung auf der projizierten Position des gegnerischen Jägers, der gerade nach oben links aus dem Sichtfeld zu flüchten versucht.



Versteckte Transformation

Leider versteckt die Grafikkarte aber die Transformation, die sie durchführt, vor uns. Es gibt also keine direkte Möglichkeit, nach dem Rendern eines Objekts seine projizierten Koordinaten von der Grafikkarte zu erfahren. Daher müssen wir selbst eine Funktion schreiben. Diese nimmt einen Punkt im 3D-Raum auf, zum Beispiel die Position eines gegnerischen Raumjägers vor uns, und projiziert diese aus den Weltkoordinaten zu den Bildschirmkoordinaten. Als Rückgabewert erhalten wir von der Funktion die x- und y-Koordinate, an der der Punkt auf dem Bildschirm gerendert würde. Genau an diesen Koordinaten können wir nun unsere Zielmarkierung auf dem Bildschirm rendern.

Und hier ist die Funktion:



```
POINT ZFXD3D::Transform3Dto2D(const ZFXVector &vcPoint)
{
    POINT pt;
    float fClip_x, fClip_y;
    float fXp, fYp, fWp;
    DWORD dwWidth, dwHeight;
```

```

// Wenn 2D-Modus, nimm den ganzen Bildschirm
if (m_nMode == EMD_TWOD) {
    dwWidth = m_dwWidth;
    dwHeight = m_dwHeight;
}
// Sonst nimm die Viewport-Dimensionen
else {
    dwWidth = m_VP[m_nStage].Width;
    dwHeight = m_VP[m_nStage].Height;
}

fClip_x = (float)(dwWidth >> 1);
fClip_y = (float)(dwHeight >> 1);

// Transformation & Projektion
fXp = (m_mViewProj._11*vcPoint.x)
    + (m_mViewProj._21*vcPoint.y)
    + (m_mViewProj._31*vcPoint.z)
    + m_mViewProj._41;
fYp = (m_mViewProj._12*vcPoint.x)
    + (m_mViewProj._22*vcPoint.y)
    + (m_mViewProj._32*vcPoint.z)
    + m_mViewProj._42;
fWp = (m_mViewProj._14*vcPoint.x)
    + (m_mViewProj._24*vcPoint.y)
    + (m_mViewProj._34*vcPoint.z)
    + m_mViewProj._44;

float fWpInv = 1.0f / fWp;

// Umrechnung von [-1,1] zu Viewport-Größe
pt.x = (LONG)( (1.0f + (fXp * fWpInv)) * fClip_x );
pt.y = (LONG)( (1.0f + (fYp * fWpInv)) * fClip_y );

return pt;
}

```

Die Funktion multipliziert lediglich den Positionsvektor im 3D-Raum mit der Combomatrix aus Viewmatrix und Projektionsmatrix. Damit ist der Punkt bereits ordnungsgemäß transformiert. Allerdings müssen wir noch eines berücksichtigen: Die zweidimensionalen Koordinaten werden von Direct3D im Viewport-Koordinatenraum verwendet, und dieser kennt nur einen Wertebereich von $-1.0f$ bis $+1.0f$, egal wie groß der Viewport in Pixeln gemessen ist. Daher müssen wir die transformierten Werte noch auf diesen Wertebereich eingrenzen, und dann mit der jeweils halben Viewport-Höhe und -breite in Pixeln multiplizieren. Dann haben wir die Koordinaten des transformierten und projizierten Punktes tatsächlich in Pixelangaben bezüglich des aktiven Viewports. Ist dieser Viewport der gesamte Bildschirm, dann sind die Koordinaten des projizierten Punktes gleichbedeutend

Viewportspace

mit Bildschirmkoordinaten, ansonsten müssen wir noch die Position des kleineren Viewports relativ zur oberen linken Ecke des Bildschirms mit in Betracht ziehen.

*Und jetzt mal
anders herum*

Der umgekehrte Fall, nämlich die Umrechnung eines Punktes (x,y) von Bildschirmkoordinaten zu Weltkoordinaten des 3D-Raums, ist zwar rein rechenmäßig nicht wesentlich aufwändiger als der eben gezeigte Weg – von den dahinterstehenden Gedanken her ist er aber nicht ganz so trivial. Bei der Projektion aus dem 3D-Raum in den 2D-Raum geht eine der drei Informationen verloren (bzw. wird in die anderen beiden Werte mit einbezogen), nämlich die Tiefeninformation des Punktes. Möchten wir aber einen 2D-Punkt in den 3D-Raum zurückprojizieren, so können wir die entsprechende Tiefeninformation nicht wieder herstellen – möglicherweise gab es sie nie.

Picking

Nehmen wir einmal an, wir klicken mit der Maus auf den Bildschirm. Dann haben wir die 2D-Bildschirmkoordinaten des Mausklicks. Um beispielsweise ein Objekt der virtuellen Welt mit der Maus anklicken zu können (das nennt man auch *Picking*), müssen wir diesen Punkt in 3D-Koordinaten umrechnen. Das ist aber nicht möglich. Dem Mausklick fehlt, so wie jedem anderen 2D-Punkt auch, einfach die dritte Information. Ein Punkt auf dem 2D-Bildschirm entspricht also bei der Rückprojektion nicht einem Punkt des 3D-Raums, sondern eines Strahls durch den 3D-Raum. Von der Position der Kamera aus verläuft dieser Strahl in Blickrichtung der Kamera bis in alle Unendlichkeit – oder eben so weit, wie wir die Pick-Reichweite der Maus zulassen wollen. Wenn wir also einen 2D-Punkt der Bildschirmkoordinaten in den 3D-Raum rückprojizieren, dann erhalten wir einen Strahl. Das macht die folgende Funktion:



```
void ZFXD3D::Transform2Dto3D(const POINT &pt,
                             ZFXVector *vcOrig,
                             ZFXVector *vcDir) {
    D3DMATRIX *pView=NULL, *pProj=NULL;
    ZFXMatrix mInvView;
    ZFXVector vcS;
    DWORD      dwWidth,
               dwHeight;

    // 2D-Modus
    if (m_Mode == EMD_TWOD) {
        dwWidth = m_dwWidth;
        dwHeight = m_dwHeight;

        pView = &m_mView2D;
    }
    // sonst orthogonale oder perspektivische Projektion
    else {
        dwWidth = m_VP[m_nStage].Width,
```



```

dwHeight = m_VP[m_nStage].Height;

pView = &m_mView3D;

if (m_Mode == EMD_PERSPECTIVE)
    pProj = &m_mProjP[m_nStage];
else
    pProj = &m_mProjO[m_nStage];
}

// skaliere zu Viewport und inverser Projektion
vcS.x = ( ((pt.x*2.0f) / dwWidth) -1.0f)
        / m_mProjP[m_nStage]._11;
vcS.y = -( ((pt.y*2.0f) / dwHeight)-1.0f)
        / m_mProjP[m_nStage]._22;
vcS.z = 1.0f;

// invertiere Viewmatrix
mInvView.InverseOf(*(ZFXMatrix*)&m_mView3D._11));

// Strahl von Bildschirm zu Weltkoordinaten
(*vcDir).x = (vcS.x * mInvView._11)
             + (vcS.y * mInvView._21)
             + (vcS.z * mInvView._31);
(*vcDir).y = (vcS.x * mInvView._12)
             + (vcS.y * mInvView._22)
             + (vcS.z * mInvView._32);
(*vcDir).z = (vcS.x * mInvView._13)
             + (vcS.y * mInvView._23)
             + (vcS.z * mInvView._33);

// inverse Translation
(*vcOrig).x = mInvView._41;
(*vcOrig).y = mInvView._42;
(*vcOrig).z = mInvView._43;

// normalisieren
(*vcOrig).Normalize();
}

```

Um die Umrechnung korrekt durchführen zu können, müssen wir die Abmessungen des aktuellen Viewports kennen. Diese ermitteln wir daher zuerst. Wenn wir im reinen 2D-Modus sind (orthogonale Projektion und 2D-Viewmatrix), dann nehmen wir den gesamten Bildschirm. Ansonsten verwenden wir die Abmessungen des Viewports der aktiven Stage. Mit Hilfe dieser Werte rechnen wir die (x,y)-Koordinaten des Punktes dann in Viewport-Koordinaten im Wertebereich $-1.0f$ bis $+1.0f$ um und modifizieren sie mit den entsprechenden Einträgen der Projektionsmatrix. Dabei dividieren wir, um mit dem inversen Wert der Projektionsmatrix zu multiplizieren.

Es werde Licht

Schließlich wollen wir die Projektion ja umgekehrt anwenden. Nun invertieren wir die Viewmatrix und multiplizieren dann mit dem eben modifizierten Vektor. So erhalten wir den Richtungsvektor des entsprechenden Strahls. Als Ursprung des Strahls nehmen wir einfach die Kameraposition aus der invertierten Viewmatrix. Das ist schon alles.

Resümee: Sicht und Projektion

Verwirrt?

Jetzt haben wir einen ganzen Batzen Funktionen an uns vorbeiziehen sehen, die alle irgendwie untereinander zusammenhängen. Das mag vielleicht wieder den Eindruck des Riesen erwecken, obwohl wir es mit einem Pygmäen zu tun haben. Um unsere Engine durchzustarten, ist nämlich gar nicht so viel notwendig, wie es scheint. Um das zu verdeutlichen, zeige ich hier einen kurzen Code-Schnipsel, der demonstriert, wie man das ZFXD3D-Device einstellen kann. Für die orthogonale Projektion und das zweidimensionale Rendern muss man eigentlich nichts extra einstellen, lediglich die Werte der Near und Far Clipping-Plane. Für die perspektivische Projektion muss aber ein Viewport und der FOV gesetzt werden.

No Panic!

Und so schaut es aus, wenn man einen Viewport über den gesamten Bildschirm und einen FOV von 0.8 RAD (ca. 45 Grad) haben möchte:



```
RECT rcView;
rcView.left   = rcView.top = 0;
rcView.right  = m_dwWidth;
rcView.bottom = m_dwHeight;

m_nMode = -1;
m_nStage = 0;

IDENTITY(m_mView3D);

SetClippingPlanes(0.1f, 1000.0f);

if (FAILED(SetFOV(0.8f, &rcView, 0)))
    return ZFX_FAIL;
```

Modus

Die maximale Sichtweite beträgt dabei 1000 Einheiten, und die minimale Sichtweite 0.1 Einheiten. Ganz analog könnte man nun für die anderen drei Stages ebenfalls Projektionsmatrizen mit einem anderen FOV (Field of View) und Viewport einstellen. Das Attribut `m_nMode` bezeichnet übrigens den Betriebsmodus der Engine. Ein Wert von 0 bezeichnet hierbei die perspektivische Projektion, ein Wert von 1 steht für den 2D-Modus (orthogonale Projektionsmatrix und 2D-Viewmatrix für das Rendern in Bildschirmkoordinaten). Die letzte Option ist der Wert 2 für die normale Viewmatrix in Kombination mit der orthogonalen Projektionsmatrix für das Rendern von zum Beispiel den Seitenansichten eines CAD-Programms.

Festlegen der Welttransformation

In unserer Engine haben wir jetzt bereits alle notwendigen Methoden, um die zwei der drei gängigen Transformationen einzustellen und zu aktivieren. Namentlich sind dies die Projektion und die Sicht. Der Dritte im Bunde ist natürlich die Welttransformation, mit der die Objekte in der virtuellen Welt verschoben und rotiert werden können. Das Festlegen der Welttransformation durch eine Weltmatrix ist dabei nicht weiter schwierig. Wenn wir aber weiterhin so komfortabel wie möglich bleiben wollen, indem wir dem Anwender unserer Engine möglichst viel Arbeit abnehmen, die er auf alle Fälle erledigen müsste, dann müssen wir doch ein kleines Stück Arbeit in diese Funktion stecken.

*Die dritte
Transformation*

Zum einen müssen wir auch wieder bedenken, was uns bereits weiter oben über den Weg gelaufen ist. Falls wir noch Polygone in unseren Containern haben, die der Anwender zwar gedanklich abgehakt hat, die aber bei uns noch nicht wirklich gerendert sind, dann müssen wir diese auf alle Fälle rendern, wenn sich die Welttransformation ändert. Dazu rufen wir wie oben auch schon die Funktion `ForcedFlushAll()` auf. Nur so ist sichergestellt, dass alle Triangles auch wirklich mit der Transformation gerendert werden, die für sie eingestellt worden ist. Wenn sich die Transformationsmatrix dann ändert, müssen wir auch die Combomatrix neu berechnen. Nun folgt wieder eine Abfrage, die wir schon kennen. Falls unsere Engine Shader verwendet – zu denen wir gleich kommen –, dann müssen wir die Transponierte der Combomatrix berechnen und als Konstante für den Vertex-Shader einstellen. Anderenfalls stellen wir die Matrix für das Direct3D-Device als Welttransformation ein, um sie in der Fixed-Function-Pipeline zu verwenden.

*Shader oder nicht
Shader – das ist
hier die Frage*

Warum und wieso das mit den Shadern so ist, darauf geht erst das nächste Kapitel ein. Wir müssen an dieser Stelle einfach die Tatsache hinnehmen, dass wir die Combomatrix aus Welt-, View- und Projektionsmatrix transponieren müssen. Mit dieser transponierten Matrix werden dann alle Vertices in der GPU transformiert und werden dadurch korrekt in Bildschirmkoordinaten umgerechnet. Das folgende Kapitel beschäftigt sich ausführlich mit Shadern.

*Transformation
via Shader*

```
void ZFXD3D::SetWorldTransform(const ZFXMatrix *mWorld)
{
    // Last Chance Check
    m_pVertexMan->ForcedFlushAll();

    // Klassenattribut 'Welt Matrix'
    if (!mWorld) {
        ZFXMatrix m; m.Identity();
        memcpy(&m_World, &m, sizeof(D3DMATRIX));
    }
}
```



```

else
    memcpy(&mWorld, mWorld, sizeof(D3DMATRIX));

// abhängige Werte neu berechnen
CalcWorldViewProjMatrix();

// falls Shader, dann Konstante setzen
if (m_bUseShaders) {
    ZFXMatrix mTranspose;
    mTranspose.TransposeOf(
        *(ZFXMatrix*)&m_mWorldViewProj);
    m_pDevice->SetVertexShaderConstantF(0,
        (float*)&mTranspose, 4);
}
else
    m_pDevice->SetTransform(D3DTS_WORLD, &mWorld);
} // SetWorldTransform

```

Auch hier steht die einfache Nutzung der Funktion im Vordergrund. Gibt der Aufrufer statt einer Matrix einfach NULL an, dann wird automatisch die Einheitsmatrix als Welttransformation eingestellt. Das entspricht dann keiner Welttransformation.

6.4 Vertex-Strukturen

Flexibilität versus Statik

Leider ist es uns aber nicht immer möglich, unsere Engine so flexibel wie möglich zu halten. Ein solches Beispiel ist das Format der verschiedenen Vertex-Strukturen. Bekanntermaßen unterscheidet man drei Grundtypen von Vertices:

- ➡ Untransformierte und unbeleuchtete Vertices
- ➡ Untransformierte und beleuchtete Vertices
- ➡ Transformierte und beleuchtete Vertices

Den letzten der hier genannten Typen können wir dabei ausklammern, da die Verwendung dieses Typs wenig Sinn macht. Wir haben ja bereits eine 2D-Projektion, mit der wir in Bildschirmkoordinaten rendern können. Das bedeutet, wir geben den Vertices bereits transformierte Koordinaten. Nun haben wir an sich nur zwei verschiedene Vertex-Typen. Die unbeleuchteten sind dabei sicherlich die am häufigsten gebrauchten, da wir heutzutage schnelle Hardware haben, mit der wir dynamische Beleuchtung in Echtzeit berechnen können. Unter Umständen kann es aber gewünscht sein, auch bereits vorbeleuchtete Vertices zu verwenden.

Flexibles Vertex- Format

Obwohl es im Moment danach aussieht, dass wir nur zwei verschiedene Vertex-Typen kennen, ist das leider implementierungstechnisch nicht ganz

so einfach. Ein Vertex kann beispielsweise nicht nur ein Texturkoordinatenpaar enthalten, sondern mehrere. Das sind jeweils zwei zusätzliche `float`-Werte, die sich natürlich entsprechend auf den Speicherverbrauch von Vertex-Listen auswirken. Daher ist es sinnvoll, für jeden Vertex-Typ eine eigene Struktur zu definieren, die nur genau die Daten enthält, die benötigt werden. Ungenutzte Felder in solchen Listen sind Speicherverschwendung. Daher kennt man bei Direct3D das so genannte *Flexible-Vertex-Format* (FVF), bei dem man sich die Komponenten seiner Vertices frei zusammenstellen kann. Würden wir diese gesamte Flexibilität durch unser Render-Device nach außen hin spiegeln wollen, dann wäre das an dieser Stelle zu viel Aufwand. Daher definiere ich zunächst zwei ganz einfache Vertex-Strukturen vor, die wir ausschließlich verwenden werden.

```
// in ZFX.h
typedef enum ZFXVERTEXID_TYPE {
    VID_UU,        // Untransf. und unbeleuchtet
    VID_UL,        // Untransf. und beleuchtet
} ZFXVERTEXID;

typedef struct VERTEX_TYPE {
    float x, y, z;
    float vcN[3];
    float tu, tv;
} VERTEX;

typedef struct LVERTEX_TYPE {
    float x, y, z;
    DWORD Color;
    float tu, tv;
} LVERTEX;

// in ZFXD3D.h
#define FVF_VERTEX ( D3DFVF_XYZ
                    | D3DFVF_NORMAL
                    | D3DFVF_TEX1 )

#define FVF_LVERTEX ( D3DFVF_XYZ
                     | D3DFVF_DIFFUSE
                     | D3DFVF_TEX1 )
```



Möchte man seine Engine später um andere Vertex-Formate erweitern, dann muss man einfach eine entsprechende Struktur definieren und die entsprechenden vom Vertex-Format abhängigen Code-Teile um diesen Sonderfall erweitern. Tabelle 6.1 zeigt die wichtigsten Bezeichner, die zur Konstruktion eines FVF-Codes in Direct3D verwendet werden können. Dabei ist die Reihenfolge, in der die entsprechenden Felder in der Vertex-

Struktur stehen, zwingend so zu verwenden, wie sie in der Tabelle auftauchen.

Tabelle 6.1:
Flexible-Vertex-
Format-Bezeichner

FVF-Bezeichner	Bedeutung
D3DFVF_XYZ	Positionsdaten
D3DFVF_XYZRHW	Transformierte Position (nur bei transformierten Vertices)
D3DFVF_NORMAL	Normalenvektor (nur bei untransformierten Vertices)
D3DFVF_PSIZE	Point-Sprite-Größe
D3DFVF_DIFFUSE	Diffuse Vertex-Farbe
D3DFVF_SPECULAR	Spekuläre Vertex-Farbe
D3DFVF_TEX0	Texturkoordinaten 0
...	...
D3DFVF_TEX8	Textur Koordinaten 8

6.5 Shader-Support

Was sind Shader?

Wer hat nicht wenigstens schon einmal von ihnen gehört, den Vertex- oder Pixel-Shadern? Gepriesen als die neue Geheimwaffe im Kampf um die besten und schnellsten Effekte bei 3D-Engines? Was genau so ein Shader jedoch ist, darauf gehen wir erst in einem späteren Kapitel ein. Das hat auch einen handfesten Grund: Shader sind API-unabhängig, das bedeutet, man kann denselben Shader mit Direct3D und OpenGL zusammen verwenden. Aber es gibt immer Dinge, die sind nicht API-unabhängig. In diesem Fall ist es das Laden und Aktivieren der Shader. Ein Shader ist eine Art Assembler-Programm, allerdings nicht für die CPU, sondern für die GPU, also den Prozessor der Grafikkarte. Dieses Programmchen muss ganz analog zu einem C/C++-Programm zunächst kompiliert werden und kann dann in die GPU geladen werden.

Trennung des API-abhängigen Codes

Wie aber bereits erwähnt wurde, gibt es einige Unterschiede, wie man ein Shader-Programm in OpenGL bzw. Direct3D lädt und aktiviert. In diesem Kapitel werden wir daher alles besprechen und in unserer DLL implementieren, was API-abhängig ist. Auf die Programmierung eines Shaders gehen wir aber erst in einem späteren Kapitel ein, weil das dann wieder API-unabhängig ist. Wenn wir jetzt aber einen Shader verwenden wollen, dann gibt es drei verschiedene Möglichkeiten, das zu tun. Zum einen kann man das Shader-Programm mit einem externen Compiler kompilieren und das fertige Kompilat als solches laden und an die GPU schicken. Zum anderen kann

man aber auch das unkompierte Shader-Programm durch den in der API eingebauten Compiler jagen und so erst das Kompilat erhalten. Diese Option unterteilt sich noch in die beiden Möglichkeiten, dass man das Shader-Programm direkt in seinem Quellcode stehen hat oder es erst aus einer Textdatei zur Laufzeit lädt. Welchen Weg man dabei einschlägt, das bleibt einem selbst überlassen und hängt sicherlich auch damit zusammen, ob man einen unkompierten Shader offen zugänglich mit seinem Produkt ausliefern möchte.

Wir werden mit unserem ZFXRenderDevice-Interface natürlich alle drei Möglichkeiten unterstützen.

Notwendige Vorbereitungen

Unsere Engine soll grundsätzlich sowohl die Fixed-Function-Pipeline teilweise unterstützen als auch über Shader flexibel rendern können. Der Einfachheit halber konzentrieren wir uns hier auf Shader. Das bedeutet: So lange Shader von der Grafikkarte unterstützt werden, verwenden wir diese auch für jeden Render-Vorgang in unserer Engine. Unterstützt die Hardware, auf der die Engine zum Laufen gebracht wird, keine Shader, dann bieten wir in begrenztem Maße die normale 3D-Pipeline der API an. Dabei werden zwar noch alle Objekte korrekt transformiert und auf den Bildschirm gerendert, aber wir werden nicht alle Renderstates unterstützen und einige Techniken wie beispielsweise Beleuchtung und Animation der Objekte nur in der Shader-Version implementieren.

*Unterstützung für
Shader ermitteln*

Bei einem echten Engine-Projekt würde man natürlich die gesamte Pipeline sowohl für Shader als auch für eine Grafikkarte ohne Shader implementieren. Die Shader-Variante würde dann sicherlich mehr und bessere Visual Effects bieten, aber die Fixed-Function-Variante könnte zumindest einen analogen Ablauf gewährleisten. Ebenso würde man sicherlich auch die Option einbauen, zur Laufzeit zwischen Shadern und Fixed-Function-Pipeline umzuschalten. Da wir hier nur eine kleine Engine als Beispiel entwickeln, verzichten wir auf diesen Mehraufwand. Da die Zukunft der 3D-Grafik im Bereich der Shader zu suchen ist, konzentrieren wir uns natürlich darauf.

*Warum nur
Shader?*

Aber bevor wir auch nur irgendetwas mit Shadern machen, brauchen wir erst mal eine Funktion, die überprüft, ob die aktuelle Hardware auch wirklich Shader unterstützt. Und bevor wir das machen, brauchen wir auch noch ein paar Attribute und Methoden für unsere Klasse ZFXD3D:

Aller Anfang ...



```
// unter private:
LPDIRECT3DVDECL9      m_pDeclVertex;
LPDIRECT3DVDECL9      m_pDeclVertex;

LPDIRECT3DVSHADER9    m_pVShader[MAX_SHADER];
LPDIRECT3DPSHADER9    m_pPShader[MAX_SHADER];
UINT                  m_nNumVShaders;
UINT                  m_nNumPShaders;

void    PrepareShaderStuff(void);

// unter public:
HRESULT CreateVShader(void*, UINT, bool, bool,
                      UINT*);
HRESULT CreatePShader(void*, UINT, bool, bool,
                      UINT*);
HRESULT ActivateVShader(UINT, ZFXVERTEXID);
HRESULT ActivatePShader(UINT);
bool    UsesShaders(void)
        { return m_bUseShaders; }
```

**Das Interface
nicht vergessen!**

Ich weise hier noch einmal darauf hin, dass wir natürlich alle Funktionen, die wir in unserer Klasse als `public` deklarieren, auch in unserem Interface als rein virtuelle Methoden anlegen müssen. Das `bool`-Attribut `m_bUseShaders` ist dort übrigens auch deklariert.

Nun schreiben wir die Funktion, die zunächst prüft, ob die Hardware Shader unterstützt. Ist dies der Fall, so werden bereits erste Initialisierungen vorgenommen.



```
void ZFXD3D::PrepareShaderStuff(void) {
    D3DCAPS9 d3dCaps;

    if (FAILED(m_pDevice->GetDeviceCaps(&d3dCaps))) {
        m_bUseShaders = false;
        return;
    }

    if (d3dCaps.VertexShaderVersion < D3DVS_VERSION(1,1))
    {
        m_bUseShaders = false;
        return;
    }

    if (d3dCaps.PixelShaderVersion < D3DPS_VERSION(1,1))
    {
        m_bUseShaders = false;
        return;
    }
}
```



```

// Vertex-Deklarationen für die Shader
D3DVERTEXELEMENT9 declVertex[] =
{
    { 0, 0, D3DDECLTYPE_FLOAT3,
      D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_POSITION, 0 },
    { 0, 12, D3DDECLTYPE_FLOAT3,
      D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_NORMAL, 0 },
    { 0, 24, D3DDECLTYPE_FLOAT2,
      D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_TEXCOORD, 0 },
    D3DDECL_END()
};

D3DVERTEXELEMENT9 declLLVertex[] =
{
    { 0, 0, D3DDECLTYPE_FLOAT3,
      D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_POSITION, 0 },
    { 0, 12, D3DDECLTYPE_D3DCOLOR,
      D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_DIFFUSE, 0 },
    { 0, 16, D3DDECLTYPE_FLOAT2,
      D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_TEXCOORD, 0 },
    D3DDECL_END()
};

// Anlegen der Deklarationen
m_pDevice->CreateVertexDeclaration(declVertex,
                                   &m_pDeclVertex);
m_pDevice->CreateVertexDeclaration(declVertex,
                                   &m_pDeclLLVertex);
m_pDevice->SetFVF(NULL);

m_bUseShaders = true;
} // PrepareShaderStuff

```

Über die Device-Caps und die Makros `D3DVS_VERSION` und `D3DPS_VERSION` können wir abfragen, ob die Grafikkarte eine bestimmte Version von Vertex- bzw. Pixel-Shadern unterstützt. Wir setzen hier lediglich die Versionen 1.1 voraus. Diese gehören auch auf bereits älteren Karten zum Standard. Neuere Versionen dürften bisher kaum allgemeine Verbreitung gefunden haben, daher bauen wir auf dem auf, was jede Karte unterstützt, die überhaupt Shader beherrscht.

Shader-Versionen

Deklarationen und
Vertex-Typen

Unterstützt die Karte die von uns geforderte Shader-Version, dann legen wir die so genannten Vertex-Deklarationen an. Diese geben an, in welcher Reihenfolge welche Daten in der Struktur eines Vertex stehen. Aber schauen wir uns zunächst einmal die Struktur D3DVERTEXELEMENT9 an:

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream;
    BYTE Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

Felder der
Struktur

Das erste Feld `Stream` gibt an, welcher Input-Stream zu verwenden ist. Durch den Aufruf von `IDirect3DDevice9::SetStreamSource` kann man mehrere Vertex-Buffer gleichzeitig für eine Grafikkarte aktivieren, um beispielsweise zwischen zwei verschiedenen Modellen zu morphen. Wir bleiben aber zunächst bei einem einzigen Stream mit dem Index 0. Das Feld `Offset` sagt aus, mit welchem Offset in Bytes die Daten in der Vertex-Struktur zu finden sind. Das Feld `Type` ist schon interessanter, denn hier spezifiziert man die Art der Daten. In der Regel haben wir es dabei mit `float`-Vektoren von 1D bis 4D zu tun. Jedoch gibt es noch andere Datentypen, die sich in der Doku des DirectX SDK finden. Das Feld `Method` stellen wir einfach auf Default. Damit sind wir bei dem Feld `Usage`, das wiederum interessant ist. Dieses Feld sagt aus, wofür die Daten zu verwenden sind. Mögliche Werte finden sich in der Tabelle 6.2. Den letzten Eintrag der Struktur stellen wir in der Regel auch auf 0.

Tabelle 6.2:
Mögliche Werte für
D3DDECLUSAGE

D3DDECLUSAGE	Bedeutung
D3DDECLUSAGE_POSITION	Positionsdaten
D3DDECLUSAGE_BLENDWEIGHT	Blending-Gewichtungsfaktor
D3DDECLUSAGE_BLENDINDICES	Blending-Index-Daten
D3DDECLUSAGE_NORMAL	Vertex-Normalenvektor
D3DDECLUSAGE_PSIZE	Point-Size-Angabe
D3DDECLUSAGE_DIFFUSE	Diffuse Vertex-Farbe
D3DDECLUSAGE_SPECULAR	Spekuläre Vertex-Farbe
D3DDECLUSAGE_TEXCOORD	Textur-Koordinaten-Paar
D3DDECLUSAGE_TANGENT	Vertex-Tangente

D3DDECLUSAGE	Bedeutung
D3DDECLUSAGE_BINORMAL	Vertex-Binormale
D3DDECLUSAGE_TESSFACTOR	Tesselationsfaktor

Tabelle 6.2:
Mögliche Werte für
D3DDECLUSAGE
(Forts.)

Nachdem wir dann die entsprechende Informationsstruktur über die Beschaffenheit eines Vertex-Elements betankt haben, erstellen wir daraus eine Vertex-Deklaration, und zwar mit der Funktion:

*Erstellung des
Deklarations-
objekts*

```
IDirect3DDevice9::CreateVertexDeclaration(  
    const LPD3DVERTEXELEMENT9 pVertexElements,  
    IDirect3DDec19 *ppDecl );
```

Als ersten Parameter geben wir hier natürlich den Zeiger auf die Struktur an, die den Aufbau eines Vertex beschreibt. Der zweite Parameter ist dann ein Referenz-Pointer auf die Adresse, an der das Deklarationsobjekt erzeugt werden soll. Um es noch einmal deutlich zu formulieren: Ein solches Deklarationsobjekt benötigen wir nur bei der Arbeit mit Vertex-Shadern. Verwenden wir die Fixed-Function-Pipeline, dann geben wir stattdessen das im Code-Ausschnitt weiter oben definierte FVF-Format für das Direct3D-Device an. Da wir nach komplettem Durchlauf der Funktion `PrepareShaderStuff()` aber wissen, dass wir Vertex-Shader verwenden können, setzen wir hier stattdessen den Wert `NULL` durch die Funktion `IDirect3DDevice9::SetFVF`. Dadurch weiß das Device, dass wir einen Shader verwenden werden.

Vertex-Shader

Um etwaiger Enttäuschung vorzubeugen, sage ich es an dieser Stelle für Querleser noch einmal: Hier behandeln wir noch keine Vertex-Shader-Programme, sondern kümmern uns nur um den API-abhängigen Teil bei Vertex-Shadern. Das heißt, wir sehen uns an, wie wir einen Vertex-Shader laden und kompilieren und dann aktivieren können.

Was kommt nun?

Dabei werden wir es so handhaben, dass der Anwender mit dem fertig kompilierten Shader nichts zu tun hat. Unser `ZFXRenderDevice` wird die entsprechenden Objekte intern in Listen speichern und nach erfolgreicher Kompilierung eines Shaders lediglich eine ID an den Aufrufer zurückgeben. Über diese ID kann der Aufrufer den Shader dann später einfach aktivieren, ohne diesen selbst irgendwo speichern zu müssen.

IDs

Laden und Kompilieren eines Vertex-Shaders

Weiter oben hatte ich ja bereits erwähnt, dass wir Shader auf vier verschiedene Arten an unsere DLL übergeben können. Entweder haben wir den Shader in einer Datei gespeichert, oder wir haben ihn bereits irgendwo im RAM vorliegen. Letzteres ist beispielsweise der Fall, wenn ein Shader als `char-`

Die vier Fälle

String in einer Anwendung hardgecodet ist. Beide eben genannten Fälle können wir auch noch mal unterscheiden, nämlich danach, ob der Shader schon kompiliert ist oder nicht. Da ich grundsätzlich faul bin, möchte ich alles mit möglichst wenig Aufwand erledigen können. Daher schreiben wir eine Funktion, die alle vier Fälle abhandeln kann.

*Parameter der
Multi-Funktion*

Dazu brauchen wir erstaunlicherweise auch nur wenige Parameter, über die man alle Fälle abhandeln kann. Zwei `bool`-Parameter steuern die Entscheidung über den zu verwendenden Fall. Einer sagt dabei aus, ob der Shader aus einer Datei zu laden ist, der andere sagt aus, ob der Shader bereits kompiliert ist. Um nun die Daten auch in die Funktion zu bekommen, benötigen wir lediglich zwei weitere Parameter. Zuerst brauchen wir einen `void`-Pointer. Dieser kann nun drei verschiedene Dinge beinhalten. Entweder ist es ein `char`-String, der ein Dateinamen darstellt, oder ein `char`-String, der einen unkompilierten Shader darstellt, oder es ist ein Pointer auf einen bereits kompilierten Shader. Wie wir den Pointer zu interpretieren haben, das entscheiden wir über die `bool`-Parameter anhand des entsprechenden Falls. Zusätzlich brauchen wir aber auch noch einen Parameter, der die Größe des Speicherbereichs angibt, auf den der Pointer zeigt.

*Speichern des
Shaders*

Den fertigen Shader speichern wir in dem Attribut `m_pVShader`, einem Array aus `IDirect3DVertexShader9`-Pointern unserer Klasse `ZFXD3D`. Der Aufrufer erhält über einen Referenz-Parameter die ID zurück, über die er den Shader später aktivieren kann. Die einzelnen Funktion zur Assemblierung und Erzeugung eines Shaders, die wir gleich verwenden werden, besprechen wir dann im Anschluss an die Funktion.



```
HRESULT ZFXD3D::CreateVShader(const void *pData,
                              UINT nSize,
                              bool bLoadFromFile,
                              bool bIsCompiled,
                              UINT *pID) {

    LPD3DXBUFFER pCode=NULL;
    LPD3DXBUFFER pDebug=NULL;
    HRESULT      hrC=ZFX_OK, hrA=ZFX_OK;
    DWORD        *pVS=NULL;
    HANDLE        hFile, hMap;

    // Haben wir noch Platz für einen?
    if (m_nNumVShaders >= (MAX_SHADER-1))
        return ZFX_OUTOFMEMORY;

    // (1): BEREITS ASSEMBLIERTER SHADER
    if (bIsCompiled) {

        // aus Datei
        if (bLoadFromFile) {
            hFile = CreateFile((LPCTSTR)pData,
```

```

        GENERIC_READ, 0,
        0, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, 0);
if (hFile == INVALID_HANDLE_VALUE)
    return ZFX_FILENOTFOUND;

hMap = CreateFileMapping(hFile, 0, PAGE_READONLY,
                        0, 0, 0);
pVS = (DWORD*)MapViewOfFile(hMap,
                            FILE_MAP_READ, 0, 0, 0);
}

// aus RAM-Pointer
else { pVS = (DWORD*)pData; }
} // if

// (2): MUSS ASSEMBLIERT WERDEN
else {
    // aus Datei
    if (bLoadFromFile) {
        hrA = D3DXAssembleShaderFromFile((char*)pData,
                                         NULL, NULL, 0,
                                         &pCode,
                                         &pDebug);
    }
    // aus RAM-Pointer
    else {
        hrA = D3DXAssembleShader((char*)pData,
                                nSize-1,
                                NULL, NULL, 0,
                                &pCode, &pDebug);
    }

    // Fehler abfangen
    if (SUCCEEDED(hrA)) {
        pVS = (DWORD*)pCode->GetBufferPointer();
    }
    else {
        Log("error: AssembleShader[FromFile]()
            failed");
        if (pDebug->GetBufferPointer())
            Log("Shader debugger says: %s",
                (char*)pDebug->GetBufferPointer());
        return ZFX_FAIL;
    }
} // else

// erstelle das Shader-Objekt
if (FAILED(hrC=m_pDevice->CreateVertexShader(pVS,
                                             &m_pVShader[m_nNumVShaders]))) {

```

```

        Log("error: CreateVertexShader() failed");
        return ZFX_FAIL;
    }

    // speichere den Index dieses Shaders
    if (pID) (*pID) = m_nNumVShaders;

    // Ressourcen freigeben
    if (bIsCompiled && bLoadFromFile) {
        UnmapViewOfFile(pVS);
        CloseHandle(hMap);
        CloseHandle(hFile);
    }

    m_nNumVShaders++;
    return ZFX_OK;
} // CreateVShader

```

Assemblierung der Shaders

Um einen nicht kompilierten Shader zu kompilieren bzw. zu assemblieren, verwenden wir hier die D3DX-Hilfsfunktionen. Also müssen wir entsprechend die `d3dx9.lib` verlinken und den `d3dx9.h`-Header inkludieren. Um einen Shader zu assemblieren, stellt die D3DX-Bibliothek insbesondere die folgenden beiden Funktionen zur Verfügung:

```

HRESULT D3DXAssembleShader(
    LPCSTR          pSrcData,
    UINT            SrcDataLen,
    CONST D3DXMACRO* Defines,
    LPD3DXINCLUDE* Include,
    DWORD           Flags,
    LPD3DXBUFFER* ppShader,
    LPD3DXBUFFER* ppErrorMsgs);

HRESULT D3DXAssembleShaderFromFile(
    LPCSTR          pSrcFile,
    CONST D3DXMACRO* Defines,
    LPD3DXINCLUDE* Include,
    DWORD           Flags,
    LPD3DXBUFFER* ppShader,
    LPD3DXBUFFER* ppErrorMsgs);

```

Der Parameter `pSrcData` ist natürlich ein Pointer auf die Daten, die den unkompilierten Shader darstellen; im Parameter `SrcDataLen` muss man dazu dann die Länge der Daten angeben. Bei der zweiten Funktion genügt es stattdessen, im Parameter `pSrcFile` den Namen der zu ladenden Datei anzugeben. Die folgenden Parameter sind bei beiden Funktionen dann identisch. `Defines`, `Includes` und `Flags` brauchen wir in der Regel nicht. Wer sich dennoch dafür interessiert, den verweise ich auf die DirectX SDK-Dokumentation. Der Parameter `ppShader` ist dann wieder spannend. Beim Datentyp

LPD3DXBUFFER handelt es sich um einen Speicher, in dem Daten hinterlegt werden können. In diesem Fall handelt es sich dabei um den fertig assemblierten Shader.

Der letzte Parameter `ppErrorMsgs` speichert freundlicherweise Debug-Informationen für uns, falls bei der Assemblierung etwas schief gehen sollte. Wie bereits im Quellcode der Funktion oben zu ersehen ist, kann man über die Funktion `GetBufferPointer()` einen Zeiger auf den Speicherbereich eines `LPD3DXBUFFER`-Objekts erhalten, in diesem Fall einen String mit den Fehlermeldungen.

Und wenn's mal schief geht ...

Es wäre auch möglich, eine eigene Funktion für die Assemblierung eines Shaders zu schreiben und dafür die Definitionen der einzelnen Komponenten eines Shaders aus den Direct3D-Headern zu verwenden. Diesen Umweg zu wählen, macht hier aber wenig Sinn, da man davon ausgehen kann, dass die Shader später hauptsächlich in vorkompilierter Form verwendet werden.



Nachdem wir nun einen Vertex-Shader assembliert haben, müssen wir noch ein `Direct3D-Vertex-Shader-Objekt` daraus machen. Dazu müssen wir den assemblierten Shader der nun folgenden Funktion des `IDirect3DDevice9`-Interfaces übergeben:

Erzeugen des Shaders

```
HRESULT CreateVertexShader(
    const DWORD *pFunction,
    IDirect3DVertexShader9** ppShader);
```

Ist dieser Funktionsaufruf erfolgreich, so haben wir nach einem langen Weg endlich unser Ziel erreicht. Wie haben einen Shader, der zum Einsatz mit dem `Direct3D-Device` bereit ist.

Um einen Shader aus einer Datei zu laden, verwenden wir die WinAPI-Funktion `CreateFile()`, um die Datei zu öffnen und ein Handle auf sie zu erhalten. Dann rufen wir einfach die WinAPI-Funktion `CreateFileMapping()` auf, gefolgt von `MapViewOfFile()`, um eine gemappte Ansicht auf die binäre Datei zu erhalten. Mit dieser Ansicht können wir dann schließlich den Shader erstellen. Die Parameterlisten der Funktionen finden sich im Plattform-SDK bei der Beschreibung der WinAPI.

Laden eines Shaders aus einer Datei

Aktivieren eines Vertex-Shaders

Nun haben wir eine Reihe von Shadern in unserer Engine. Doch diese sind bisher nicht viel mehr als in Pointern gespeicherte Daten. Um diese Daten zur Grafikkarte zu bekommen, oder besser gesagt, um sie dort zu aktivieren, damit sie zum Rendern verwendet werden, müssen wir bei den Vertex-Shadern noch genau zwei Dinge tun: Zum einen brauchen wir die Funktion `IDirect3DDevice9::SetVertexDeclaration`. Durch diese aktivieren wir ein Deklarationsobjekt für Vertex-Strukturen. Nur so weiß die Grafikkarte, welche Daten an welcher Stelle in einem Vertex zu finden sind. Zum anderen brau-

Deklaration und Shader aktivieren

chen wir natürlich die Funktion `IDirect3DDevice9::SetVertexShader`. Durch diese können wir dann einen vorher erzeugten Vertex-Shader für das Direct3D-Device aktivieren.



```
HRESULT ZFXD3D::ActivateVShader(UINT nID,
                                ZFXVERTEXID VertexID)
{
    if (!m_bUseShaders) return ZFX_NOSHADERSUPPORT;
    if (nID >= m_nNumVShaders) return ZFX_INVALIDID;

    // leeren der Vertex-Caches
    m_pVertexMan->ForcedFlushAll();

    // hole Größe und Format des Vertex
    switch (VertexID) {
        case VID_UU: {
            if (FAILED(m_pDevice->SetVertexDeclaration(
                m_pDeclVertex)))
                return ZFX_FAIL;
            } break;
        case VID_UL: {
            if (FAILED(m_pDevice->SetVertexDeclaration(
                m_pDeclLVertex)))
                return ZFX_FAIL;
            } break;
        default: return ZFX_INVALIDID;
    } // switch

    if (FAILED(m_pDevice->SetVertexShader(
        m_pVShader[nID])))
        return ZFX_FAIL;

    return ZFX_OK;
} // ActivateVShader
```

Im Detail Unsere Funktion haben wir natürlich schlau genug gemacht, damit sie erkennt, ob wir ihr eine gültige ID übergeben haben bzw. ob die Engine überhaupt mit Shadern läuft. Die ID ist natürlich der Wert, den unsere Funktion `ZFXD3D::CreateVShader` für einen Shader geliefert hat. Natürlich wäre es lästig, wenn ein Anwender auch für die simpelsten Dinge einen Shader setzen müsste. Daher werden wir nachher bei der Initialisierung aus der DLL heraus einen ersten Shader mit der ID 0 erstellen und aktivieren. Dieser führt sozusagen die Standardtransformation der Vertices durch, so dass die Engine auch läuft, ohne dass der Anwender einen eigenen Shader schreiben muss. Aber Vorsicht: Wenn wir einen aktiven Vertex-Shader ändern, dann müssen wir auch dafür sorgen, dass wir sämtliche Vertices in unseren Zwischenspeichern zuerst rendern, da diese ja mit einem anderen aktiven Vertex-Shader gerendert werden sollten. Der Anwender unserer Engine weiß ja

nicht, dass wir diese Vertices nur gecacht, aber noch nicht wirklich gerendert haben. Daher rufen wir auch hier wieder die mysteriöse Funktion `ForcedFlushAll()` auf.

Pixel-Shader

Das Schöne an den Pixel-Shadern ist, dass wir sie von der Seite der Implementierung her gesehen ganz analog zu den Vertex-Shadern laden, kompilieren und aktivieren können. Letzten Endes wird es sogar noch ein wenig einfacher als bei den Vertex-Shadern, denn die lästigen Vertex-Strukturen und die daraus resultierenden Deklarationen fallen hier weg, so dass der Code beim Aktivieren noch übersichtlicher wird.

Analog zu Vertex-Shadern

Laden und Kompilieren eines Pixel-Shaders

Das Laden und Assemblieren eines Pixel-Shaders funktioniert genauso wie in der entsprechenden Funktion, die wir eben für Vertex-Shader geschrieben haben. Um uns also Wiederholungen wie im Fernsehen zu ersparen, drucke ich die Funktion `ZFXD3D::CreatePShader` hier nicht ab. Diese Funktion hat nämlich nur drei kleine Änderungen gegenüber der obigen Funktion für Vertex-Shader. Zum einen verwenden wir nun den Zähler `m_nNumPShaders` und das Array `m_pPShader`. Zum anderen brauchen wir jetzt für die Pixel-Shader natürlich die Funktion `IDirect3DDevice::CreatePixelShader` anstelle des Vertex-Shader-Äquivalents. Davon abgesehen ist die Funktion vollkommen identisch zu `ZFXD3D::CreateVShader`. Der Quelltext findet sich aber selbstverständlich komplett auf der CD zu diesem Buch.

Wie im Fernsehen!

Aktivieren eines Pixel-Shaders

Um einen von uns erzeugten Pixel-Shader für Direct3D zu aktivieren, müssen wir lediglich die Funktion `IDirect3D::SetPixelShader` aufrufen. Im Gegensatz zu den Vertex-Shadern müssen wir uns hier nicht mit Deklarationen oder Ähnlichem herumschlagen. Ein Pixel ist ein Pixel, daran gibt es nun mal nichts zu deklarieren.

Das ist ja einfach!

```
HRESULT ZFXD3D::ActivatePShader(UINT nID) {
    if (!m_bUseShaders) return ZFX_NOSHADERSUPPORT;
    if (nID >= m_nNumPShaders) return ZFX_INVALIDID;

    // leeren der Vertex-Caches
    m_pVertexMan->ForcedFlushAll();

    if (FAILED(m_pDevice->SetPixelShader(
                                                m_pPShader[nID])))
        return ZFX_FAIL;

    return ZFX_OK;
} // ActivatePShader
```



Error-Checking

Da es hier eigentlich nichts anderes mehr zu sagen gibt, weise ich noch einmal darauf hin, dass man das Error-Checking nicht allzu stiefmütterlich behandeln sollte. Daher prüft auch diese Funktion brav, ob wir überhaupt Shader verwenden können und ob die übergebene ID des Pixel-Shaders überhaupt korrekt ist. Diese kleinen, manchmal zugegebenermaßen lästigen Security-Checks zwischendurch sorgen dafür, dass unsere Engine wesentlich weniger Möglichkeiten hat, total zu crashen, wenn man beispielsweise durch eine falsche ID auf nicht validen Speicher zugreifen würde.

Flush!

Auch hier wird natürlich die mysteriöse Funktion `ForcedFlushAll()` aufgerufen. Hier gilt das Gleiche wie bei dem Aktivieren eines Vertex-Shaders. Bevor wir den aktiven Shader ändern, müssen wir sicherstellen, auch wirklich alles gerendert zu haben, was mit dem nun abzulösenden Shader gerendert werden sollte.

6.6 Aktivierung von Renderstates

Wir nähern uns nun dem eigentlichen Rendern von Grafik mit großen Schritten. Eine Sache müssen wir aber auf alle Fälle noch klären, die in direktem Zusammenhang zum Rendern steht. Und das sind die Renderstates, also die Zustände, unter denen das Direct3D-Device rendert. Dabei handelt es sich beispielsweise um Einstellungen für die Beleuchtung, den Depth-Buffer usw. Ich möchte hier nicht weiter ins Detail gehen und auch nicht alle Methoden auflisten, die wir im Laufe dieses Buches benötigen werden.

Auch wenn wir unsere Engine so intelligent wie möglich gestalten, kommen wir an einigen Stellen absolut nicht darum herum, dem Anwender unserer Render-DLL die Möglichkeit zu bieten, bestimmte Renderstates direkt selbst zu setzen. Dafür schreiben wir die folgende Enumeration in den Header `ZFX.h`:



```
typedef enum ZFXRENDERSTATE_TYPE {
    RS_CULL_CW,           // Culling im Uhrzeigersinn
    RS_CULL_CCW,          // Culling im Gegenuhrzeigersinn
    RS_CULL_NONE,         // Front- und Back-Side rendern
    RS_DEPTH_READWRITE,  // Z-Buffer lesen und schreiben
    RS_DEPTH_READONLY,   // Z-Buffer nur lesen
    RS_DEPTH_NONE        // kein Z-Buffer
    RS_SHADE_POINTS,     // Nur Vertices rendern
    RS_SHADE_TRIWIRE,    // Wireframe trianguliert rendern
    RS_SHADE_HULLWIRE,   // Wireframe der Polygone rendern
    RS_SHADE_SOLID       // Solide, texturierte Polygone
} ZFXRENDERSTATE;
```

Die obige Auflistung erhebt dabei keinerlei Anspruch auf Vollständigkeit. Ich habe hier nur beispielhaft ein paar Renderstates aufgeführt, aber im Verlauf des Buches wird unsere Engine unter Umständen noch mehr benötigen. Wir erkennen unsere Renderstates an dem Präfix `RS_`. Hier haben wir Renderstates definiert, mit denen wir das Backface-Culling, den Depth-Buffer und den Füllmodus der Polygone einstellen können.

Ein Polygon – und so auch die Dreiecke, die eine 3D-API rendert – hat natürlich immer zwei Seiten. Für gewöhnlich ist aber nur eine davon sichtbar. Ein Würfel beispielsweise hat sechs Seiten, von denen die Innenseiten jeweils nicht sichtbar sind. Es wäre eine Verschwendung von Rechenpower, diese Seiten dennoch durch die 3D-Pipeline zu jagen und sie auf 2D zu projizieren. Daher entscheidet die Reihenfolge, in der man die Vertices angibt, darüber, welches die sichtbare Vorderseite (engl. Front Face) und welches die unsichtbare Rückseite (engl. Back Face) ist.

Das liegt daran, dass der Normalenvektor des Polygons die Richtung der Vorderseite angibt. Die Richtung des Normalenvektors wiederum wird berechnet als das Kreuzprodukt zwischen den beiden nicht-parallelen Vektoren, die sich aus den ersten möglichen drei Vertices eines Polygons formen lassen. Verwendet man diese Vertices in umgedrehter Reihenfolge, so wird auch die Richtung des Normalenvektors umgedreht.

Dazu integrieren wir in unser Interface die folgenden Methoden, mit denen wir unsere Renderstates zum Einsatz bringen:

```
// aus dem Interface ZFXRenderDevice.h
ZFXCOLOR      m_clrWire;
ZFXRENDERSTATE m_ShadeMode;

virtual void SetBackfaceCulling(ZFXRENDERSTATE)=0;
virtual void SetDepthBufferMode(ZFXRENDERSTATE)=0;

virtual void SetShadeMode (ZFXRENDERSTATE, float,
                           const ZFXCOLOR*)=0;
virtual ZFXRENDERSTATE GetShadeMode (void)=0;
```

Diese Methoden umzusetzen ist natürlich vollkommen trivial, denn wir müssen nur die entsprechenden Renderstates von Direct3D einstellen. Wer in diesen Dingen noch Nachholbedarf hat, der sollte einen Blick in die DirectX SDK-Dokumentation werfen, um zu sehen, was die einzelnen Renderstates bewirken und welche weiteren Renderstates es gibt.

```
void ZFXD3D::SetBackfaceCulling(ZFXRENDERSTATE rs) {
    m_pVertexMan->ForcedFlushAll();
    if (rs == RS_CULL_CW)
        m_pDevice->SetRenderState(D3DRS_CULLMODE,
                                   D3DCULL_CW);
}
```



```

        else if (rs == RS_CULL_CCW)
            m_pDevice->SetRenderState(D3DRS_CULLMODE,
                                      D3DCULL_CCW);
        else if (rs == RS_CULL_NONE)
            m_pDevice->SetRenderState(D3DRS_CULLMODE,
                                      D3DCULL_NONE);
    } // SetBackfaceCulling
/*-----*/

void ZFXD3D::SetDepthBufferMode(ZFXRENDERSTATE rs) {
    m_pVertexMan->ForcedFlushAll();
    if (rs == RS_DEPTH_READWRITE) {
        m_pDevice->SetRenderState(D3DRS_ZENABLE,
                                  D3DZB_TRUE);
        m_pDevice->SetRenderState(D3DRS_ZWRITEENABLE,
                                  TRUE);
    }
    else if (rs == RS_DEPTH_READONLY) {
        m_pDevice->SetRenderState(D3DRS_ZENABLE,
                                  D3DZB_TRUE);
        m_pDevice->SetRenderState(D3DRS_ZWRITEENABLE,
                                  FALSE);
    }
    else if (rs == RS_DEPTH_NONE){
        m_pDevice->SetRenderState(D3DRS_ZENABLE,
                                  D3DZB_FALSE);
        m_pDevice->SetRenderState(D3DRS_ZWRITEENABLE,
                                  FALSE);
    }
    } // SetDepthBufferMode
/*-----*/

void ZFXD3D::SetShadeMode(ZFXRENDERSTATE smd, float f,
                          const ZFXCOLOR *pClr) {
    m_pVertexMan->ForcedFlushAll();

    // kopiere neue Farbe, falls gegeben
    if (pClr) {
        memcpy(&m_clrWire, pClr, sizeof(ZFXCOLOR));
        m_pVertexMan->InvalidateStates();
    }

    // keine Änderung am aktuellen Modus
    if (smd == m_ShadeMode) {
        // eventuell Änderung der Größe
        if (smd==RS_SHADE_POINTS)
            m_pDevice->SetRenderState(D3DRS_POINTSIZE,
                                      FtoDW(f));
        return;
    }
}

```

```

if (smd == RS_SHADE_TRIWIRE) {
    // echter Wireframe-Modus von Direct3D
    m_pDevice->SetRenderState(D3DRS_FILLMODE,
                             D3DFILL_WIREFRAME);

    m_ShadeMode = smd;
}
else {
    m_pDevice->SetRenderState(D3DRS_FILLMODE,
                             D3DFILL_SOLID);

    m_ShadeMode = smd;
}

if (smd == RS_SHADE_POINTS) {
    if (f > 0.0f) {
        m_pDevice->SetRenderState(
            D3DRS_POINTSPRITEENABLE, TRUE);
        m_pDevice->SetRenderState(
            D3DRS_POINTSCALEENABLE, TRUE);
        m_pDevice->SetRenderState(D3DRS_POINTSIZE,
            FtoDW( f ));
        m_pDevice->SetRenderState(D3DRS_POINTSIZE_MIN,
            FtoDW(0.00f));
        m_pDevice->SetRenderState(D3DRS_POINTSCALE_A,
            FtoDW(0.00f));
        m_pDevice->SetRenderState(D3DRS_POINTSCALE_B,
            FtoDW(0.00f));
        m_pDevice->SetRenderState(D3DRS_POINTSCALE_C,
            FtoDW(1.00f));
    }
    else {
        m_pDevice->SetRenderState(
            D3DRS_POINTSPRITEENABLE, FALSE);
        m_pDevice->SetRenderState(
            D3DRS_POINTSCALEENABLE, FALSE);
    }
}
else {
    m_pDevice->SetRenderState(D3DRS_POINTSPRITEENABLE,
        FALSE);
    m_pDevice->SetRenderState(D3DRS_POINTSCALEENABLE,
        FALSE);
}

// Update der abhängigen States
m_pVertexMan->InvalidateStates();
} // SetShadeMode
/*-----*/

ZFXRENDERSTATE GetShadeMode() { return m_ShadeMode; }

```

Point-Modus

Hoppla, die Funktion für den Shade-Modus ist ja doch ein wenig komplexer geworden. Das liegt daran, dass wir in unserer Engine relativ leicht zwischen den verschiedenen Modi wechseln können müssen. Insbesondere benötigen wir bei der Programmierung unseres Level-Editors in einem späteren Kapitel die komfortable Umschaltung zwischen dem Rendern von Punkten, im Wireframe-Modus usw. Hier kann man sehen, dass wir Punkte beispielsweise in Form von Point-Sprites rendern. Ebenso wie bei unseren Wireframe-Modi kann man optional eine Farbe angeben, die für die Vertices verwendet werden soll. Der `float`-Parameter ist einzig und allein für die Größe der Point-Sprites gedacht und hat für die anderen Modi keine Bedeutung.



Point Sprites bezeichnen eine besondere Fähigkeit der Hardware, Billboards automatisch darzustellen. Dabei muss der Benutzer nur einen einzigen Punkt im 3D-Raum angeben und eine zugehörige Größe. Die Grafikkarte zieht dann automatisch ein Rechteck um diesen Punkt in der angegebenen Größe. Dieses Rechteck ist immer parallel zum Bildschirm ausgerichtet.

Solche Point-Sprites verwendet man normalerweise, um Partikel eines Partikelsystems anzuzeigen, ohne selbst die Billboards berechnen zu müssen. Die Grafikkarte muss diese Option jedoch unterstützen, was heutzutage aber Standard sein sollte.

Wireframe-Modi

Für den Wireframe-Modus haben wir zwei Optionen. Wählt man den Direct3D-Wireframe-Modus, so sieht man ein Polygon, beispielsweise ein Rechteck, in der triangulierten Form, also als zwei Drahtgitter-Dreiecke. Das ist an vielen Stellen nicht gewünscht und in einem Editor auch relativ störend. Daher haben wir noch den Shade-Modus `RS_SHADE_HULLWIRE` definiert. Dieser verwendet explizit nicht den Direct3D-Wireframe-Modus, sondern rendert das Polygon als Line-Strip. Die Vertices des Polygons werden dabei einfach als Punkte einer Linie interpretiert, die gerendert wird. Dazu ist es allerdings notwendig, dass man eine entsprechende Indexliste angibt. Für ein Rechteck mit vier Vertices bräuchte man die Indices `[0, 1, 2, 3, 0]`, um einen geschlossenen Linienzug durch die Vertices zu ziehen.

Wie immer müssen wir daran denken, alle gecachten Daten zu flushen, bevor wir an den grundlegenden Renderstates etwas verändern. Ebenso müssen wir dem Vertex-Cache-Manager mitteilen, dass sich nun einiges geändert hat, damit dieser weiß, dass seine gespeicherten Zustände nicht mehr gültig sind.

Da ist noch mehr drin.

Wie bereits weiter oben erwähnt wurde, gibt es noch andere Renderstates, die wir später setzen müssen. Da das Prinzip an dieser Stelle aber klar sein sollte, spare ich mir das Auflisten der weiteren Funktionen, die wir dazu über das Interface exponieren.

6.7 Effizientes Rendern von grafischen Primitiven

Nun verschaffen wir uns einen kurzen Überblick darüber, was wir bisher eigentlich erreicht haben. Unser Render-Device enthält bereits eine ganze Menge an miteinander verwobenem Quellcode. Doch alles, was wir da bisher haben, dient lediglich einem Zweck, nämlich der Initialisierung der Grafik-API unserer Wahl (hier Direct3D) bzw. natürlich auch der dynamischen Änderung der Einstellungen (Sicht und Projektion) zur Laufzeit der Engine. Doch was ist eine Grafik-Engine ohne die Möglichkeit, Grafik zu rendern?

*Standort-
Bestimmung*

Deshalb wollen wir uns jetzt an die Aufgabe machen, Grafik zu rendern – oder wenigstens die Voraussetzungen dafür zu schaffen, dass unsere DLL auch Funktionen anbietet, über die eine Applikation, die die DLL verwendet, grafische Primitive rendern kann. Wir beschränken uns hier zunächst auf Dreiecke bzw. Listen von Dreiecken, die unsere Engine rendern können soll. Daneben gibt es noch die Optionen Punkte, Linien, Fans usw. zu rendern. Doch diese Funktionalitäten lassen wir hier außen vor. Es ist wichtiger, die Grundprinzipien zu verstehen. Dann kann man später seine eigenen Render-Methoden ergänzen.

Methodik

Grundlagen zu Hardware und Performance

Die Überschrift dieses Abschnitts spricht bereits von Effizienz. Ich kann es gar nicht oft genug erwähnen, aber selbst mit einem einzigen Modell mit nur wenigen hundert Polygonen kann man eine top-aktuelle Grafikkarte so in die Knie zwingen, dass diese sich wünschen wird niemals produziert worden zu sein – beispielsweise, wenn man jedes Dreieck dieses Modells separat rendert und Dutzende hochauflösender Texturen kreuz und quer verwendet. Um jedoch zu verstehen, warum welcher Ansatz effizient bzw. nicht effizient ist, braucht man schon ein wenig Verständnis für den Aufbau der Hardware. Im Kapitel über Vertex- und Pixel-Shader werde ich noch näher auf die 3D-Pipeline eingehen, aber bereits an dieser Stelle müssen wir einige Grundkenntnisse haben. Dazu schauen wir uns Abbildung 6.5 an.

Effizienz

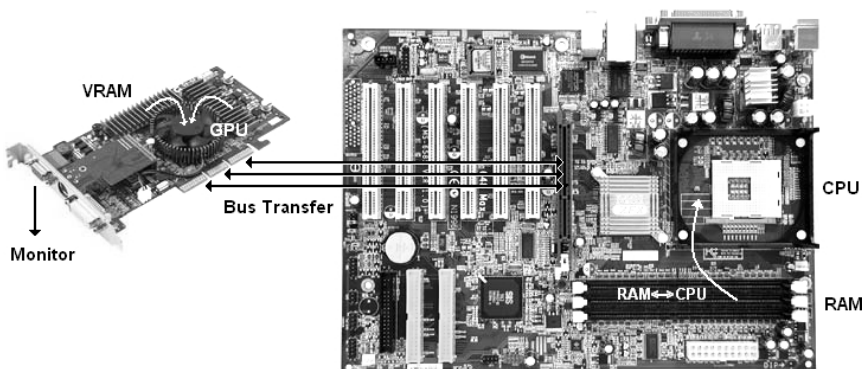


Abbildung 6.5:
Mainboard und
Grafikkarte

**Hardware-
Architektur**

Dort sehen wir ein Mainboard und eine Grafikkarte. Rechts auf dem Mainboard ist der Sockel für die CPU, direkt darunter befindet sich der System-RAM. Im AGP-Slot steckt dann die Grafikkarte, auf der zwei Blöcke Video-RAM rechtwinklig um die GPU angeordnet sind.

Nun haben wir quasi zwei Kommunikationszentren in dieser Anordnung: Zum einen muss die CPU Daten aus dem RAM ziehen, Berechnungen auf den Daten durchführen und diese dann in den RAM zurückschreiben. Das zweite Zentrum liegt auf der Grafikkarte. Hier zieht sich die GPU Daten aus dem VRAM, führt Berechnungen aus und schreibt gegebenenfalls Daten zurück in den VRAM. Die Kommunikation in diesen beiden Zentren läuft auf der jeweiligen Hardwarekarte sehr rasant ab. Die Geschwindigkeit der Datenübertragung ist extrem schnell. Leider gibt es aber einen Engpass in dieser Architektur, und das ist der so genannte Bus. Über die heutzutage aktuellen AGP-Slots ist die Datenübertragung bereits schneller als über PCI, VESA Local Bus oder die noch älteren Slot-Typen. Aber die Transferraten über den Bus sind um ein Vielfaches geringer als die Kommunikation zwischen CPU und RAM bzw. zwischen GPU und VRAM.

Die Probleme

Damit sind wir schon bei der Effizienz heutiger Grafik-Applikationen. Wann immer Daten über den Bus zur Grafikkarte wandern (oder umgekehrt) ist das so, als ob man in voller Fahrt die Handbremse anzieht. Früher war der Bottleneck, also die Performance-Bremse, eher in der Transformation von 3D-Daten und dem Setzen von Pixeln zu sehen. Daher verwendeten alte Spiele, wie beispielsweise *Doom* und *Quake*, insbesondere Optimierungen zur Minimierung des Pixel-Overdraws (HSR, Hidden Surface Removal). Heutzutage transformieren und rendern Grafikkarten 140 Millionen Vertices pro Sekunde, und die Transferrate des VRAM liegt bei über 10 Gbyte an Daten pro Sekunde. Die Berechnungen der CPU, die man für HSR ausführen müsste, würden länger dauern als das Brute-Force-Rendern aller Daten – ob Overdraw oder nicht. So weit, so gut, das Problem ist aber, dass die Transferrate über einen AGP-Bus bei ca. einem Zehntel dieses Wertes liegt (x4 AGP). Und dabei ist der AGP-Slot bereits durch einen direkten Zugang zum RAM auf dem Mainboard für 3D-Anwendungen optimiert. Heutzutage liegt daher die Herausforderung bei der Optimierung also eher darin, den Verkehr auf dem Bus zu minimieren.

Lösungsansatz

Was geht überhaupt über den Bus zur Grafikkarte? Nun, über den Bus wandert alles, was die Grafikkarte zur Anzeige der Grafik benötigt. In erster Linie sind das die Vertex- und Indexdaten sowie die Texturen. Eine schlecht entwickelte 3D-Engine würde beispielsweise die zum Rendern notwendigen Vertex- und Indexdaten und die Texturen in jedem Frame über den Bus zur Grafikkarte schieben. Eine gut programmierte 3D-Engine hingegen lädt die Texturen und die Vertex- und Indexdaten bei der Initialisierung in den VRAM der Grafikkarte und hat ab dann – abgesehen von ein paar Befehlen an die Grafikkarte – keinerlei Verkehr über den Bus.

Natürlich ist es nicht wirklich so einfach. Der VRAM auf der Grafikkarte ist extrem begrenzt. Als Standard kann man heute bereits 64 Mbyte voraussetzen, auch wenn einige ältere Karten noch 32 Mbyte haben. Das klingt zunächst nach einer ganzen Menge, aber eine einfache Rechnung relativiert dies wieder. Bereits der Front- und Back-Buffer belegen im VRAM einen Menge Platz. Je höher die Auflösung und die Farbtiefe, desto mehr Speicherplatz verbrauchen sie im VRAM. Gleiches gilt für die Grafiken, die als Texturen verwendet werden. Nicht umsonst bieten viele Spiele beispielsweise die Option, Texturen in 16 oder 32 Bit zu verwenden. Je geringer die Farbtiefe, umso mehr Texturen passen in den VRAM. Und dann sind da noch die Vertex- und Indexdaten. Je nach Aufbau der Vertex-Struktur (mehrere Texturkoordinaten-Paare, Normalenvektoren usw.) belegen diese dann auch entsprechend Speicher. Schließlich hat man in der Regel um die zehn- bis fünfzehntausend Polygone in einem ordentlichen Level eines Ego-Shooters. Selbst wenn jedes Polygon nur ein Dreieck ist und wir nur drei `float`-Koordinaten je Vertex annehmen, ist das bereits ca. ein halbes Megabyte an Daten. Nehmen wir jetzt für jeden Vertex noch einen Satz Texturkoordinaten und einen Normalenvektor hinzu, dann haben wir fast eine Verdreifachung der Datenmenge – und dabei haben wir noch keine einzige Textur geladen. Eine 32-Bit-Grafik in der Größe 512x512 würde beispielsweise einen Megabyte Speicher belegen. Und ein Level aus nur einer Textur wäre zudem stinklangweilig. Allein für jeden Gegnertyp hätten wir gern eine eigene Textur, dazu ein Set von mindestens zwanzig Texturen für einen Level, damit er nicht zu langweilig wirkt, plus Texturen für Waffen und Ähnliches. So ist man schnell bei zwanzig bis dreißig Megabyte allein für Texturen.

Ist es wirklich so einfach?

Was passiert also, wenn mehr Daten vorhanden sind, als in den VRAM hineinpassen? Glücklicherweise gibt es dafür heutzutage viele Erleichterungen, beispielsweise verwalten die gängigen 3D-APIs das Management der Daten. Wenn nämlich beispielsweise eine Textur nicht mehr in den VRAM hineinpasst, dann wird sie (so man denn die API-Aufrufe entsprechend gestaltet hat) über den Bus in den RAM geschubst. Möchte man nun Daten rendern, die diese Textur verwenden, so wird sie über den Bus wieder in den VRAM geladen. Da dieser sich aber in der Zwischenzeit leider nicht magisch vergrößert hat, würde die Textur dort immer noch nicht hineinpassen. Also nimmt die API eine andere Textur, schmeißt diese aus dem VRAM und schickt sie über den Bus in den RAM. Wie wir sehen, haben wir bereits jetzt eine Menge Transfer über den Bus. Und wenn man keinen vernünftigen Plan hat, dann ist der Busverkehr logischerweise sehr ineffizient.

Busfahrplan

Das eben geschilderte Beispiel ist genau der Grund dafür, warum man seine Polygone vor dem Rendern so weit wie möglich sortieren sollte. Beispielsweise fasst man idealerweise *alle* Dreiecke in seiner Engine zusammen, die dieselbe Textur verwenden. Dann rendert man alle diese Dreiecke nacheinander und wiederholt dasselbe Prinzip für die nächste Textur. So verhindert

Effizienz auf dem Bus

man, dass ein und dieselbe Textur beispielsweise in einem Frame mehrfach über den Bus wandern muss, weil sie mal gebraucht wird und mal im Weg ist. So hat sie maximal eine Hinfahrt und eine Rückfahrt über den Bus.

Noch mehr Traffic

Neben Texturen sind auf dem Bus aber noch andere Daten unterwegs, beispielsweise Vertex- und Indexdaten. Nehmen wir mal ein einfaches Beispiel, das eine Option aufzeigen soll, wie man heutzutage seine Engine optimieren kann und worüber man noch vor einigen Jahren die Hände über dem Kopf zusammengeschlagen hätte. Man hat ein 3D -Modell eines Terrains. Dessen Vertex- und Indexdaten sind gut in einem Vertex- und Indexbuffer im VRAM verstaut, damit wir schneller rendern können. In jedem Frame muss man die Höhe des Spielers und der anderen Objekte auf dem Terrain bestimmen. Dazu braucht man aber die Vertex-Daten. Also schickt man über den Bus eine Anfrage an den Buffer und erhält, ebenfalls über den Bus, eine Kopie der Vertex-Daten. Und hui ... weg ist die Geschwindigkeit. Sinnvoller wäre es heutzutage, von Anfang an eine Kopie der Vertex-Daten in seinen eigenen Strukturen im RAM zu halten und die Buffer im VRAM möglichst nicht anzufassen. Dann hat man halt mal ein paar Megabyte an Daten doppelt. Was zunächst wie Verschwendung aussieht, ist eine sehr gute Art der Performance-Optimierung für heutige Hardware.

Nun gut, ich glaube, jetzt habe ich genug über Grundlagen geredet. Jetzt sollten wir uns darum kümmern, wie wir ein effizientes System zum Rendern von Grafik in unsere DLL einbauen.

Caching beim Rendern

Alles ins Töpfchen

Man könnte nun einen sehr umfangreichen Code entwerfen, der dem Anwender sämtliche Denkarbeit abnimmt. Das würde wie folgt aussehen: Der Aufrufer wirft unserem Render-Device nach und nach einzelne Dreiecke oder sogar ganze Listen von Dreiecken in den Rachen. Unser Device sammelt alle diese Daten in eigenen Listen ein, bis ein Frame am Bildschirm angezeigt werden soll. Erst dann weiß das schlaue Device, dass keine weiteren Daten mehr folgen, und sortiert seine intern gespeicherten Daten nach Texturen usw. Erst dann sendet das Device die zu rendernden Daten in vernünftig sortierten Häppchen an die API zum tatsächlichen Rendern. Auf diese Art und Weise kann man beispielsweise die berüchtigten Textur-Switches vermeiden, wo sie vermeidbar sind.

Schön wär's ja

Doch was wäre die Welt, wenn alles so einfach wäre? Der Haken an diesem Ansatz ist, dass es einfach viel zu viel zu sortieren gäbe. Beim Rendern eines Dreiecks besteht eben nicht nur eine Abhängigkeit zu einer bestimmten Textur. Zu einem zu rendernden Dreieck gehört neben der Textur auch ein spezifisches Material, eine spezifische Weltmatrix, gegebenenfalls ein Vertex- und ein Pixel-Shader. Unter Umständen soll auch ein Teil der Daten in einen

anderen Viewport gerendert werden usw. Hier einen sinnvollen Weg zur optimalen Sortierung zu finden ist ein hoffnungsloses Unterfangen. Und nun? Sollen wir von vornherein die Waffen strecken?

Natürlich nicht. Aber wir setzen hier einfach voraus, dass der Anwender unserer API in begrenztem Umfang auch Kenntnis davon hat, dass er seine Objekte idealerweise ein wenig vorsortiert.

Einige Computerspiele wurden so programmiert, dass sie in einem Frame alle Objekte in einer bestimmten Reihenfolge renderten und im folgenden Frame genau in umgekehrter Reihenfolge. Dadurch konnte die Anzahl der Textur-Switches verringert werden, da die zuerst verwendeten Texturen in einem Frame die zuletzt verwendeten Texturen aus dem vorhergehenden Frame waren. Daher waren sie auf alle Fälle noch im VRAM und mussten nicht erst neu geladen werden.

:-)
TIPP

Hier lernen wir auch wieder eine Grundregel der Optimierung: Bevor man etwas optimiert, wo man Optimierungspotenzial erkennt, sollte man sich fragen, ob eine Optimierung an dieser Stelle wirklich sinnvoll ist. Bleiben wir noch einen Moment bei den Vertex-Daten. Diese durch ein Render-Device sortieren zu lassen macht wenig Sinn, da spezifische Applikationen spezifische Anforderungen haben können. Wir sollten an dieser Stelle den einfacheren Weg wählen und unserer DLL noch zusätzliche Bibliotheken mit Objekt-Loadern beilegen. Diese übernehmen dann den Job, die Vertex-Daten möglichst effizient zu initialisieren und so zu unterteilen, dass wir sie effizient rendern können. Dennoch ist es aber sinnvoll, eine Form des Cachings von Vertex-Daten auch in unser Device zu implementieren. Dazu müssen wir aber ein bisschen mehr über Vertex- und Index-Buffer wissen.

Problem-
Erkennung

Statische vs. dynamische Vertex- und Index-Buffer

In Direct3D läuft das Rendern jeglicher grafischer Primitive über so genannte Vertex- und Index-Buffer. Diese sind nichts weiter als eine Art Array mit den entsprechenden Daten. Bei OpenGL bezeichnet man diese Objekte als Vertex-Arrays. Während OpenGL aber diverse Abläufe allein intern regelt, erlaubt Direct3D eine flexible Spezifikation der Vertex-Buffer. Grundsätzlich ist es so, dass man die Bufferobjekte zunächst einmal erzeugen muss. Idealerweise verwendet man dabei ein entsprechendes Flag, das Direct3D veranlasst, den Buffer im geeignetsten Speicher anzulegen und zu verwalten. Das wird in der Regel der VRAM der Grafikkarte sein. Wenn man nun Daten in den Buffer schieben oder aus ihm lesen will, dann muss man den Buffer verriegeln – so wie wir es auch von den Direct3D-Texturen her kennen. So lange der Buffer verriegelt ist, weiß Direct3D, dass wir an den Daten arbeiten und die Konsistenz der Daten so lange nicht gewährleistet ist. Folglich kann Direct3D einen verriegelten Buffer nicht verwenden. Haben wir den Datentransfer mit dem Buffer abgeschlossen, müssen wir ihn wieder

Was sind Buffer?

entriegeln, um ihn für Direct3D wieder freizuschalten. Wenn wir nun etwas rendern wollen, dann geben wir Direct3D den entsprechenden Vertex-Buffer und gegebenenfalls den Index-Buffer bekannt und rufen dann einfach die `DrawPrimitive()`- bzw. `DrawIndexedPrimitive()`-Funktion des Direct3D-Devices auf.

Verschiedene Buffer-Varianten

Das funktioniert auch alles wunderbar, bis auf einen kleinen Haken: Performant läuft das nur für statische Geometrie, also bei Vertex-Daten und Index-Daten, die sich im Verlauf des Programms nicht verformen. Das liegt daran, dass wir natürlich bei jedem Aufruf zur Verriegelung des Buffers Daten über den Bus schicken. Schließlich verriegeln wir den Buffer nur, um Daten aus ihm zu lesen oder in ihn zu schreiben. Schlimmstenfalls noch beides gleichzeitig. Das ist aber in den seltensten Fällen notwendig. Daher kann man in DirectX über verschiedene Flags beim Anlegen eines Buffers angeben, wie man diesen zu verwenden gedenkt. Wenn man von vornherein weiß, dass man die Daten in einem bestimmten Buffer nie wieder anfassen wird, dann kann man sich per Flag quasi selbst verbieten, Daten aus dem Buffer lesen zu können. Und wenn man genau weiß, dass man einen Buffer oft verriegeln und mit vielen Daten betanken will, dann kann man durch Angabe gewisser Flags dafür sorgen, dass der Buffer zwar insgesamt an Performance verliert, aber die Betankung des Buffers wesentlich schneller vonstatten gehen kann.

Dynamisch versus statisch

Einen Buffer im VRAM, auf den man nach der Initial-Betankung mit Daten möglichst gar nicht mehr zugreift, nennt man im Direct3D-Jargon einen *statischen* Buffer. Dies ist die schnellste Art von Buffer, die man verwenden kann. Dabei unterscheidet man die Art eines Buffers nur anhand der Flags, die man bei der Initialisierung angibt. Durch bestimmte Flags kann man aber auch einen so genannten *dynamischen* Buffer erstellen. Dieser zeichnet sich dadurch aus, dass die Betankung des Buffers mit Daten schneller vonstatten gehen kann, als es bei einem statischen Buffer der Fall wäre. Das liegt daran, dass der Buffer dann nicht im VRAM angelegt wird, sondern in der Regel im schnellen AGP-Speicher platziert wird. Dadurch ist der Datenaustausch mit dem RAM schneller, das Rendern des Buffers jedoch langsamer als das Rendern eines statischen Buffers, der im VRAM sitzt. Wenn man jedoch den Buffer mindestens einmal je Frame verriegeln und große Datenmengen austauschen muss, dann ist ein dynamischer Buffer insgesamt betrachtet in der Regel schneller als ein statischer, auf dem dieselben Operationen ausgeführt werden.



Direct3D bietet auch die Möglichkeit, grafische Primitive über die beiden Device-Interface-Funktionen `DrawPrimitiveUP()` beziehungsweise `DrawIndexedPrimitiveUP()` zu rendern. Das UP steht dabei für User Pointer. Das bedeutet, dass man zuvor keinen Vertex-Buffer anlegen muss, sondern einen Zeiger auf ein ganz normales C/C++-Array angeben kann, in dem die Vertex- bzw. Index-Daten gespeichert sind.

Dadurch entsteht oftmals der Eindruck, dass man in Direct3D auch ohne Buffer-Objekte rendern könnte. Das ist aber nicht so, da Direct3D bei der Initialisierung jeweils ein eigenes internes dynamisches Buffer-Objekt für Vertices und Indices anlegt. Und genau über diese beiden Objekte schicken diese beiden Funktionen die ihnen übergebenen Daten.

Für uns bedeutet das zunächst, dass wir möglichst immer statische Buffer verwenden sollten. Es kann aber durchaus auch die Situation geben, in der ein Anwender Daten rendern möchte, die er nicht vorher sortieren kann oder möchte. Wir werden daher mit unserer DLL zwei Möglichkeiten anbieten, grafische Primitive zu rendern. Zum einen soll man von dem Render-Device einen statischen Buffer beantragen können, den man später nur noch zum Rendern aufzurufen braucht. Zum anderen wollen wir aber auch eine Funktion haben, die jederzeit ein Array von Vertex- und Indexdaten übernehmen und rendern kann. Und da wir optimieren wollen, leiten wir die Daten bei der zweiten Option nicht einfach an die entsprechende UP-Funktion von Direct3D weiter, sondern bauen an genau dieser Stelle ein Caching der Daten ein. Für alles, was durch einen dynamischen Buffer gezogen werden soll, stellen wir also in unserem Render-Device ein paar Töpfchen bereit, in die wir die Daten, die der Aufrufer rendern will, erst einmal einsortieren. Im schlimmsten Fall schickt ein Aufrufer Hunderte von Polygonen einzeln an unsere Funktion, die aber alle dieselbe Textur verwenden. Würden wir einfach die UP-Funktion von Direct3D verwenden, so wäre das sehr langsam. Wir sammeln die Daten aber zunächst alle nach einem mehr oder weniger intelligenten System ein und rendern sie erst, wenn wir eine gewisse Menge an Daten haben oder keine neuen mehr dazukommen. Aber dafür brauchen wir zunächst die ...

Was bedeutet das für uns?

Interface-Definition für einen Vertex-Cache-Manager

Neben unserem Render-Device und dem Skin-Manager haben wir hier das dritte Interface, das von unserer DLL implementiert werden wird. Unseren so genannten Vertex-Cache-Manager werden wir später als Attribut der Klasse ZFXD3D verwenden, um sämtliche Render-Aufrufe abzuarbeiten. Werfen wir zunächst einen kurzen Blick auf das recht kompakt gehaltene Interface.

VC-Manager im Gesamtbild

```
class ZFXVertexCacheManager {
public:
    ZFXVertexCacheManager(void) {};
    virtual ~ZFXVertexCacheManager(void) {};

    virtual HRESULT CreateStaticBuffer(
        ZFXVERTEXID VertexID,
        UINT    nSkinID,
        UINT    nVerts,
        UINT    nIndis,
        const void *pVerts,
```



```

const WORD *pIndis,
UINT *pnID)=0;

virtual HRESULT Render(ZFXVERTEXID VertexID,
    UINT nVerts,
    UINT nIndis,
    const void *pVerts,
    const WORD *pIndis,
    UINT SkinID)=0;

virtual HRESULT Render(UINT nSBufferID)=0;

virtual HRESULT ForcedFlushAll(void)=0;

virtual HRESULT ForcedFlush(ZFXVERTEXID)=0;
}; // class

```

**Erstellen, Rendern
und Flushen**

Im ersten Moment erscheint die Trennung von dynamischen und statischen Buffern in diesem Interface nicht ganz so offensichtlich zu sein. Schauen wir uns den Sinn und Zweck der einzelnen Funktionen daher etwas genauer an. Die Bedeutung der Parameterlisten klären wir dann später, wenn wir zu der realen Implementierung kommen.

Die Funktion `CreateStaticBuffer()` dient natürlich dazu, einen statischen Buffer anzulegen. Dabei geben wir mindestens den Vertex-Buffer an, können aber auch einen Index-Buffer mit anlegen. Man beachte aber insbesondere, dass das Interface keine Möglichkeit vorsieht, auf statisch angelegte Buffer zuzugreifen. Wie bereits weiter oben erwähnt wurde, ist das Verriegeln eines Buffers durch die `Lock()`-Funktion ein unschlagbarer Performance-Killer, den wir nicht in unserer Engine zulassen wollen.



In vielen Implementierungen einfacher 3D-Engines aus Hobby-Projekten sieht man auch eine frameweise Verwendung der `Lock()`-Methoden von Vertex- und Index-Buffern. Dies sollte man so weit es geht vermeiden, da die Daten sonst über den Bus geschickt werden müssen. Hier macht es oft mehr Sinn, eine Kopie der Daten des Buffers in einem eigenen Array im Sys-RAM zu halten und dieses Array für Kollisionsberechnungen und Ähnliches zu verwenden. Die Daten innerhalb eines Buffers im VRAM sollten ausschließlich zum Rendern verwendet werden.

Für die statischen Buffer haben wir dann lediglich eine weitere Funktion in unserem Interface, nämlich eine `Render()`-Funktion. Bei der Erstellung des Buffers erhalten wir eine ID für den Buffer, und diese geben wir als einzigen Parameter der Funktion zum Rendern. Diese sorgt dann dafür, dass der Inhalt des Buffers am Bildschirm erscheint.

Für die dynamischen Buffer haben wir eigentlich so gut wie gar keinen Initialisierungsaufwand, daher gibt es für sie keine Erstellungsfunktion. Die zweite Renderfunktion in unserem Interface dient dann dazu, Vertex- und Indexdaten über einen dynamischen Buffer zu rendern. Dieser Renderfunktion geben wir daher direkt die Vertex- und Indexdaten an. Nun haben wir noch zwei weitere Funktionen in unserem Interface, die zum Flushen eines oder aller dynamischen Buffer eines Vertex-Cache-Managers dienen. Damit ist gemeint, dass alle Daten, die bereits über die Renderfunktion für dynamische Buffer übergeben wurden, aber noch nicht gerendert sind, nun sofort gerendert werden sollen.

*Dynamische
Buffer*

Dass wir diese Option brauchen werden, das sehen wir gleich. Weiter oben hatte ich ja bereits angedeutet, dass unser Vertex-Cache-Manager die Daten zunächst zwischenspeichert und nicht sofort rendert. Aber spätestens, wenn wir eine Szene abschließen, müssen wir ja sicherstellen, dass wirklich auch alles gerendert wurde.

Als Nächstes sehen wir uns an, wie wir dynamische Vertex- und Indexdaten in unserer Implementierung behandeln. Dies ist etwas komplizierter als die statischen Daten. Für diese müssen wir ja lediglich bei der Initialisierung die jeweiligen Buffer erzeugen und sie bei Aufforderung stur herunterrendern. Um jedoch dynamisch effizient zu rendern, brauchen wir noch eine weitere Klasse als Hilfsmittel, und um diese kümmern wir uns im folgenden Abschnitt.

Vertex-Cache-Objekt

Im vorigen Abschnitt haben wir ein Interface für einen Manager definiert. Ein Manager braucht aber immer etwas, das er auch managen kann. Zum einen muss er ja die statischen Vertex- und Index-Buffer verwalten. Zum anderen soll er aber auch Vertex- und Indexdaten durch dynamische Buffer rendern können. Das ist mit mehr Aufwand verbunden als bei den statischen Buffern, und daher lagern wir diese Funktionalität in ein Vertex-Cache-Objekt aus. Es ist auch nicht sehr performant, wenn man nur ein paar Dreiecke zum Rendern schickt, daher legen wir das Vertex-Cache-Objekt eben so aus, dass die Daten dort zunächst lediglich gecacht werden. Das bedeutet, wir speichern so viel wie möglich an Daten zwischen und rendern diese erst über Direct3D, wenn wir genügend Dreiecke zusammen haben oder uns der Platz ausgeht.

*Cache für
dynamische Daten*

Diese Funktionalität implementieren wir aber direkt über eine Klasse in der DLL und schreiben dieses Verfahren explizit nicht durch ein Interface vor. Wie eine tatsächliche Implementierung für beispielsweise OpenGL die dynamischen Daten abhandelt, überlassen wir ganz dem Programmierer der Render-DLL. Wir verwenden für unsere Direct3D-Implementierung die folgende Klasse.



```
class ZFXD3DVCache {
public:

    ZFXD3DVCache(UINT nVertsMax, UINT nIndisMax,
                  UINT nStride,
                  ZFXD3DSkinManager *pSkinMan,
                  LPDIRECT3DDEVICE9 pDevice,
                  ZFXD3DVCManager *pDad,
                  DWORD dwID, FILE *pLog);

    ~ZFXD3DVCache(void);

    HRESULT Flush(bool bUseShaders);

    HRESULT Add(UINT nVerts, UINT nIndis,
                const void *pVerts,
                const WORD *pIndis,
                bool bUseShaders);

    void SetSkin(UINT SkinID, bool bUseShader);

    bool UsesSkin(UINT SkinID)
    { return (m_SkinID == SkinID); }
    bool IsEmpty(void)
    { if (m_nNumVerts>0) return false; return true; }
    int NumVerts(void)
    { return m_nNumVerts; }

private:
    LPDIRECT3DVERTEXBUFFER9 m_pVB;
    LPDIRECT3DINDEXBUFFER9 m_pIB;
    LPDIRECT3DDEVICE9 m_pDevice;
    ZFXD3DSkinManager *m_pSkinMan;
    ZFXD3DVCManager *m_pDad;
    ZFXSKIN m_Skin;
    UINT m_SkinID;
    DWORD m_dwID;
    FILE *m_pLog;

    UINT m_nNumVertsMax; // max. Verts. im Buffer
    UINT m_nNumIndisMax; // max. Indices im Buffer
    UINT m_nNumVerts; // moment. Anz. im Buffer
    UINT m_nNumIndis; // moment. Anz. im Buffer
    UINT m_nStride; // Stride eines Vertex
}; // class
```

Dynamische Buffer

Grundsätzlich stellt jede Instanz dieser Klasse einen dynamischen Buffer dar bzw. eine Kombination aus einem dynamischen Vertex- und einem dynamischen Indexbuffer. Diese sind daher als Attribute der Klasse vorhanden, zusammen mit verschiedenen Zählern über den aktuellen und den maxima-

len Füllstand der Buffer. Der Konstruktor der Klasse dient dazu, diese beiden Buffer zu initialisieren. Ein Vertex-Cache-Manager verfügt, wie wir gleich sehen werden, über ein ganzes Array solcher ZFXD3DVCache-Instanzen. Die Funktion `ZFXD3DVCache::Add` wird von dem Manager dazu verwendet, Daten in die Buffer eines Cache-Objekts zu stecken. Dies passiert immer dann, wenn der Manager Vertex- und Indexdaten vom Anwender der DLL über die Funktion `ZFXVertexCacheManager::Render` geschickt bekommt. Diese werden dann einem bestimmten Cache-Objekt hinzugefügt. Soll der Inhalt eines Caches tatsächlich gerendert werden, dann verwenden wir die Funktion `ZFXVCache::Flush`, die den gesamten Inhalt des Vertex- und Indexbuffers der Instanz über `Direct3D` rendert.

Damit haben wir auch schon alle Funktionen, die wir in dieser Klasse auf alle Fälle brauchen werden. Die anderen vier Funktionen der Klasse sind nur kleine Hilfsmethoden für die Abfrage und das Setzen der Attribute der Klasse. Insbesondere sehen wir hier den Begriff *Skin* in unserem Vertex-Cache-Objekt.

Doch was haben die Skins nun mit den Cache-Objekten zu tun? Diese Frage zu beantworten sollte nun für uns trivial sein. Der Sinn und Zweck unseres Caches ist es, alle ankommenden Dreiecke erst einmal zwischenspeichern und erst dann zu rendern, wenn eine bestimmte Menge erreicht ist. So minimieren wir den Traffic über den Bus und unterfordern die Grafikkarte nicht. Das würde sie nur unnötig ausbremsen. Doch das Problem beim Rendern der Dreiecke ist leider folgendes: Jeder Aufruf zum Rendern von Dreiecken legt auch eine bestimmte Textur fest, die für die Dreiecke zu verwenden ist. Wir können nicht einfach die Dreiecke zwischenspeichern und dann zusammen rendern, weil dann einige Dreiecke die falsche Textur haben werden.

*Skins und der
Cache*

Wenn wir unsere Dreiecke also cachen wollen, dann müssen wir diese zusätzlich nach Texturen bzw. Skins ordnen. Das tun wir in unserem Fall über den Vertex-Cache-Manager. Dieser hat nämlich für eine bestimmte Anzahl x von Texturen genau x Vertex-Cache-Objekte, also Instanzen der oben gezeigten Klasse. Alle Dreiecke, die dieselbe Skin verwenden, werden im selben Cache gespeichert. Kommen Dreiecke an, die eine andere Textur verwenden, dann kommen diese in einen anderen Cache. Zu diesem Management-Prozess kommen wir dann nachher ausführlich. Im Folgenden implementieren wir erst einmal ein solches Vertex-Cache-Objekt.

Erstellung und Freigabe eines Vertex-Cache-Objekts

Im Konstruktor dieser Klasse tun wir gewohnt wenig. Wir initialisieren die Attribute mit Startwerten und legen den Vertex- und den Index-Buffer an. Wichtig ist hierbei einerseits, dass ein Vertex-Cache-Objekt auch über seine *Verwandtschaft* informiert ist. Das Objekt muss also das `Direct3D`-Device kennen ebenso wie das `Skin-Manager`-Objekt der DLL und den Vertex-

Cache-Manager, zu dem es gehört. Die jeweiligen Pointer auf diese Objekte müssen dem Konstruktor daher übergeben werden.

Counter und IDs

Auch muss bekannt sein, wie viele Vertices und Indices ein Cache maximal aufnehmen soll. Diese Zahl ist extrem wichtig, denn wenn das Maximum überschritten ist, dann wird der Inhalt des Buffers gerendert. Das bedeutet: Je kleiner das Maximum ist, desto weniger Dreiecke werden jeweils zusammen gerendert. Zu guter Letzt hat jeder Cache noch eine eindeutige ID. Dies dient dazu, dass man in der DLL feststellen kann, welcher Cache gerade aktiv ist.



```
ZFXD3DVCache::ZFXD3DVCache(UINT nVertsMax,
                             UINT nIndisMax,
                             UINT nStride,
                             ZFXD3DSkinManager *pSkinMan,
                             LPDIRECT3DDEVICE9 pDevice,
                             ZFXD3DVCManager *pDad,
                             DWORD dwID, FILE *pLog) {

    HRESULT hr;

    m_pDevice      = pDevice;
    m_pSkinMan     = pSkinMan;
    m_pDad         = pDad;
    m_nNumVertsMax = nVertsMax;
    m_nNumIndisMax = nIndisMax;
    m_nNumVerts    = 0;
    m_nNumIndis    = 0;
    m_dwID         = dwID;
    m_nStride      = nStride;
    m_pLog         = pLog;

    memset(&m_Skin, MAX_ID, sizeof(ZFXSKIN));
    m_SkinID = MAX_ID;

    // Erstelle den Buffer
    m_pVB = NULL;
    m_pIB = NULL;

    hr = pDevice->CreateVertexBuffer(nVertsMax * nStride,
                                     D3DUSAGE_DYNAMIC |
                                     D3DUSAGE_WRITEONLY,
                                     0, D3DPool_DEFAULT,
                                     &m_pVB, NULL);

    if (FAILED(hr)) m_pVB = NULL;

    hr = pDevice->CreateIndexBuffer(nIndisMax *
                                   sizeof(WORD),
                                   D3DUSAGE_DYNAMIC |
                                   D3DUSAGE_WRITEONLY,
                                   D3DFMT_INDEX16,
```

```

D3DPPOOL_DEFAULT,
&m_pIB, NULL);

if (FAILED(hr)) m_pIB = NULL;
}

```

Man beachte hier, dass wir den Vertex- und den Index-Buffer mit den Flags `D3DUSAGE_DYNAMIC` und `D3DUSAGE_WRITEONLY` anlegen. Das erstgenannte Flag sorgt dafür, dass Direct3D die Buffer im richtigen Speicher platziert. Statische Buffer (implizit, also kein entsprechendes Flag) werden für gewöhnlich im schnellen VRAM der Grafikkarte angelegt. Dynamische Buffer hingegen sitzen im AGP-Speicher vor dem Bus. Bei einem dynamischen Buffer geht man davon aus, dass oft auf ihn zugegriffen werden soll. Daher liegt er nicht im VRAM, weil man so bei einem lesenden Zugriff alle Daten über den Bus schicken müsste. Aber die Platzierung im AGP-Speicher sorgt dafür, dass beispielsweise der schreibende Zugriff auf den Buffer schneller erfolgen kann.

*Die richtigen
Create-Flags!*

Beim Freigeben eines Vertex-Cache-Objekts haben wir aufgrund des einfachen Konstruktors auch nichts anderes zu tun, als den Vertex- und den Index-Buffer der Instanz wieder freizugeben.

Aufräumen

```

ZFXD3DVCache::~ZFXD3DVCache(void) {
    if (m_pVB) {
        m_pVB->Release();
        m_pVB = NULL;
    }
    if (m_pIB) {
        m_pIB->Release();
        m_pIB = NULL;
    }
}

```



Wie wir bereits sehen können, ist die Vertex-Cache-Klasse kaum komplizierter als Pfannkuchenessen. Das wird sich auch im Folgenden nicht großartig ändern, denn die gesamte Logik für die Effizienz steckt ja in dem Manager, der diese Objekte später verwalten wird.

Skin des Vertex-Cache-Objekts einstellen

Ein Vertex-Cache-Objekt ist direkt abhängig von der für ihn eingestellten Skin, also der Kombination aus Material und Textur. Schließlich kann ein Vertex-Cache-Objekt nur diejenigen Dreiecke aufnehmen, die dieselbe Skin verwenden wie die Dreiecke, die bereits in dem Cache-Objekt sitzen. Allerdings können wir jederzeit auch eine andere Skin für einen Cache festlegen, beispielsweise wenn wir das Objekt für eine andere Skin verwenden wollen. Ist ein Cache-Objekt bereits leer, so können wir die Skin, mit der das Objekt verwendet wird, problemlos ändern. Befinden sich jedoch noch Dreiecke in dem Cache, dann müssen diese zuerst noch gerendert werden, bevor wir die Skin des Objekts ändern dürfen. Das erledigt die folgende Funktion für uns.

Skin-Integrität



```
void ZFXD3DVCache::SetSkin(UINT SkinID,
                           bool bUseShaders) {
    if (!UsesSkin(SkinID)) {
        ZFXSKIN *pSkin = &m_pSkinMan->GetSkin(SkinID);

        if (!IsEmpty()) Flush(bUseShaders);

        memcpy(&m_Skin, pSkin, sizeof(ZFXSKIN));
        m_SkinID = SkinID;

        m_pDad->SetActiveCache(MAX_ID);
    }
}
```

Tracking aktiver States

Falls das Objekt die nun einzustellende Skin nicht bereits verwendet, so lassen wir uns von dem Skin-Manager einen Pointer auf die gewünschte Skin geben. Befinden sich noch Daten in unserem Cache, rufen wir die `Flush()`-Funktion auf, um alle Daten noch mit der alten Skin zu rendern. Dann kopieren wir die Daten aus der `ZFXSKIN`-Struktur in den entsprechenden Member des Cache-Objekts, damit nun die neue Skin von dem Objekt verwendet wird. Um nun die Integrität aller Datenstrukturen zu erhalten, haben wir im Vertex-Cache-Manager die Funktion `ZFXD3DVCManager::SetActiveCache`. Mit dieser Funktion wird lediglich die ID des momentan aktiven Cache-Objekts gesetzt. `MAX_ID` ist dabei ein Wert, der so interpretiert wird, dass kein Objekt aktiv ist. Aktiv bedeutet dabei, dass dieses Objekt beispielsweise seine Skin für das Device gesetzt hat. Wir müssen also nur eine Textur für das Direct3D-Device setzen, wenn sich das aktive Objekt ändert. Wenn wir eine neue Skin für das Objekt einstellen, dann teilen wir unserer DLL auf diese Weise mit, dass die aktuellen Einstellungen von Material und Textur für das Direct3D-Device keine Gültigkeit mehr besitzen.

Hinzufügen von Daten

Nun kommt endlich eine etwas interessantere Funktion. Wenn der Anwender unserer DLL über den Vertex-Cache-Manager die Funktion `ZFXD3DVCManager::Render` aufruft, der er Vertex- und Indexdaten angibt, dann werden diese durch die folgende Funktion in ein entsprechendes Cache-Objekt des Managers einsortiert.



```
HRESULT ZFXD3DVCache::Add(UINT nVerts, UINT nIndis,
                          const void *pVerts,
                          const WORD *pIndices,
                          bool bUseShaders) {
    BYTE *tmp_pVerts=NULL;
    WORD *tmp_pIndis=NULL;
    int nSizeV = m_nStride*nVerts;
    int nSizeI = sizeof(WORD)*nIndis;
    int nPosV;
```

```

int    nPosI;
DWORD dwFlags;

// Ist die Buffer-Größe überhaupt ausreichend?
if (nVerts>m_nNumVertsMax || nIndis>m_nNumIndisMax)
    return ZFX_BUFFERSIZE;

// Der Cache ist schon voll, also leeren wir ihn
if ( (nVerts+m_nNumVerts > m_nNumVertsMax) ||
     (nIndis+m_nNumIndis > m_nNumIndisMax) ) {
    if ( Flush(bUseShaders) != ZFX_OK)
        return ZFX_FAIL;
}

// DISCARD-Flag, falls Buffer leer ist
if (m_nNumVerts == 0) {
    nPosV = nPosI = 0;
    dwFlags = D3DLOCK_DISCARD;
}

// sonst per OVERWRITE-Flag anhängen
else {
    nPosV = m_nStride*m_nNumVerts;
    nPosI = sizeof(WORD)*m_nNumIndis;
    dwFlags = D3DLOCK_NOOVERWRITE;
}

// Buffer verriegeln
if (FAILED(m_pVB->Lock(nPosV, nSizeV,
                     (void**)&tmp_pVerts,
                     dwFlags)))
    return ZFX_BUFFERLOCK;
if (FAILED(m_pIB->Lock(nPosI, nSizeI,
                     (void**)&tmp_pIndis,
                     dwFlags))) {
    m_pVB->Unlock();
    return ZFX_BUFFERLOCK;
}

// Vertexdaten umkopieren
memcpy(tmp_pVerts, pVerts, nSizeV);

// Indices kopieren
int nBase = m_nNumVerts;
if (!pIndices) nIndis = nVerts;

for (UINT i=0; i<nIndis; i++) {
    if (pIndices != NULL)
        tmp_pIndis[i] = pIndices[i] + nBase;
    else
        tmp_pIndis[i] = i + nBase;
}

```

```

        m_nNumIndis++;
    }

    // inkrementieren
    m_nNumVerts += nVerts;

    m_pVB->Unlock();
    m_pIB->Unlock();
    return ZFX_OK;
}

```

Lock-Flags

In dieser Funktion testen wir zunächst, ob wir noch genug Platz in den Buffern haben, um die neu ankommenden Daten aufzunehmen. Ist das nicht der Fall, so leeren wir den Buffer durch einen Flush. Dann verriegeln wir die Buffer und kopieren die Daten entsprechend. Die richtige Verwendung der Lock-Flags ist in dem folgenden Exkurs erläutert. Durch die richtige Anwendung dieser Flags kann man ganz erhebliche Performance-Gewinne erzielen. Die tatsächliche Anwendung eines etwas komplexeren Lock-Systems verdeutlicht das *Point-Sprite-Sample* des DirectX SDK. Dort werden jeweils kleine Vertex-Mengen in einen dynamischen Buffer kopiert und gerendert.

Ein Problem hat man jedoch dann, wenn man nicht nur einen Vertex-, sondern auch einen zugehörigen Index-Buffer verwendet. In diesem Fall kann man nicht so leicht nur einen Teil der Vertex-Daten kopieren und gleich rendern, da die Vertices in der Liste nicht unbedingt in der Reihenfolge kommen, wie sie zu rendern sind. Man müsste hier anhand der Indexliste die gerade benötigten Daten einzeln aus dem Vertex-Buffer herauskopieren, was einen Teil der Performance wieder zunichte macht.

In unserer Funktion bieten wir daher ein nicht ganz so komplexes System von Lock-Aufrufen an. Vielmehr wollen wir später über unseren Vertex-Cache-Manager in so einem Vertex-Cache-Objekt insbesondere Polygone gleicher Texturen zusammenbatchen.

Die richtigen Lock-Flags

Bei der Verwendung dynamischer Vertex- und Index-Buffer (durch Angabe des Flags `D3DUSAGE_DYNAMIC` bei der Erstellung der Buffer) kommt es darauf an, die `Lock()`-Methode des jeweiligen Interfaces mit dem passenden Flag zu verwenden. Zur Auswahl stehen dabei die Flags `D3DLOCK_DISCARD` und `D3DLOCK_NOOVERWRITE`.

Das erste Flag sagt aus, dass der Anwender bei einem Lock-Aufruf vorhandene Daten in dem Buffer überschreibt. Dieses Flag muss man verwenden, wenn man den Buffer zum ersten Mal befüllt oder wenn man ihn vom Index 0 an neu befüllen will.

Sämtliche alten Daten im Buffer gehen dabei verloren. Das ist mit einem normalen Lock vergleichbar, wie man ihn bei einem statischen Buffer macht. Direct3D weiß dann, dass sich die Daten in dem Buffer nun ändern werden, und kann den Buffer so lange nicht verwenden.

Echte Performance bringt dann das zweitgenannte Flag. Durch Angabe dieses Flags verpflichtet sich der Benutzer, die Integrität der im Buffer bereits vorhandenen Daten nicht zu verletzen. Dieses Flag verwendet man, wenn man neue Daten an die bestehenden im Buffer anhängen möchte. Beispielsweise befinden sich in einem Vertex-Buffer bereits 1000 Vertices, und wir möchten zusätzlich noch mal 500 an diese anhängen. Wir verriegeln also den Buffer ab dem Index 1000 und kopieren 500 neue Vertices in den Speicher des Buffers. Nach dem Lock haben wir dann 1500 Vertices im Buffer. Nun stellt sich die Frage, warum das performanter ist als der Discard-Lock. Die Antwort ist denkbar einfach: Direct3D weiß nun, dass sich die alten Daten im Buffer nicht ändern. Das bedeutet, dass Direct3D die alten Daten bereits rendern kann, während gleichzeitig neue Daten an den Buffer angehängt werden. Das ist bei einem anderen Lock nicht möglich. Wenn wir beispielsweise eine Liste von 1500 Vertices haben, dann können wir jeweils 500 Vertices mit einem Lock in den Buffer schieben und die Render-Funktion aufrufen. Dann hängen wir die nächsten 500 Vertices an und rufen wieder die Render-Funktion auf. Jetzt können zwei Prozesse gleichzeitig ablaufen. Zum einen kann die Grafikkarte die Daten aus dem Buffer bereits rendern (die ersten 500 Vertices), und zum anderen kopieren wir bereits die nächsten 500 Vertices in den Buffer. Wir blockieren das System also nicht damit, dass wir warten müssen, bis die Render-Funktion fertig ist. Direct3D hängt also nicht so lange, bis jeweils 500 Vertices gerendert sind, sondern lässt unser Programm gleich weiterlaufen. Das erhöht die Performance beträchtlich.

Der folgende Pseudo-Code zeigt, wie man kleine Mengen an Polygonen durch dynamische Buffer effizient rendern kann. Den Performance-Schub erreicht man hier, indem man seine Polygonmenge in kleine Häppchen unterteilt, die dann parallel gerendert werden, während man den nächsten Happen in die Buffer kopiert.

```
for loop()
{
    if there is space in the buffer
    {
        // Vertices/Indices anhängen.
        pBuffer->Lock(...D3DLOCK_NOOVERWRITE...);
    }
    else
    {
        // Reinitialisieren
        pBuffer->Lock(...D3DLOCK_DISCARD...);
    }
}
```

```

    }
    Fill few 10s of vertices/indices in pBuffer
    pBuffer->Unlock
    Change State
    DrawPrimitive() or DrawIndexedPrimitive()
}

```

Sendet man jedoch größere Polygonmengen auf einmal zum Rendern an einen dynamischen Buffer, dann ist es besser, die folgende Methode zu verwenden:

```

for loop()
{
    // neuen Zeiger zurückgeben
    pBuffer->Lock(...D3DLOCK_DISCARD...);

    Fill data (optimally 1000s of vertices/indices,
               no fewer) in pBuffer.
    pBuffer->Unlock
    for loop( 100s of times )
    {
        Change State
        // Zehnerpakete von Primitiven
        DrawPrimitive() or DrawIndexPrimitives()
    }
}

```

Letzten Endes kann man nicht sagen, welche Variante der Buffer-Betankung die beste ist, da dies von der tatsächlichen Füllmenge und Frequentierung der Buffer abhängt. Man sollte hier verschiedene Testläufe bei einem konkreten Projekt fahren, um die beste Lösung für eine konkrete Aufgabe zu finden.

Rendern aus dem Vertex-Cache-Objekt

Das ganze Geheimnis, wie man eine gute Performance bei dem dynamischen Rendern von Triangles erzielt, haben wir nun bereits geklärt. Der Performance-Boost, verglichen mit dem Anwenden der normalen `IDirect3DDevice9::Draw (Indexed)PrimitiveUP`-Funktion, beruht auf der Art und Weise, wie wir einen Buffer in der Methode `ZFXD3DVCache::Add` verriegeln und betanken. Das Rendern der Daten ist eigentlich nur noch der schlichte Aufruf der Funktion `IDirect3DDevice9::DrawIndexedPrimitive`. Allerdings gibt es auch hier die Möglichkeit, noch ein wenig Speed herauszuholen, indem wir prüfen, welche Daten bereits auf dem Direct3D-Device aktiv sind. Die folgende Funktion prüft zunächst, ob der Cache bereits aktiv ist. Wenn wir viele Daten in dasselbe Cache-Objekt pumpen und zwischendurch immer wieder dieses Objekt rendern, dann brauchen wir beispielsweise die Textur und das Material nicht jedes Mal neu zu setzen, da diese bereits aktiv sind.

Die folgende Funktion ist daher auf den ersten Blick ein wenig umfangreicher, da sie genau evaluiert, welche Daten für das Device wirklich noch gesetzt werden müssen. Das FVF müssen wir beispielsweise auch nur setzen, wenn wir keine Vertex-Shader verwenden. Ein weiterer Spezialfall ist das Rendern im Wireframe-Modus. Normalerweise rendert Direct3D dann die Objekte nicht mehr mit gefüllten Polygonen, sondern mit texturierten Linien. Wir programmieren unsere Engine aber so, dass sie keine Texturen verwendet, wenn sie im Wireframe-Modus ist. Stattdessen lassen wir den Anwender ein Attribut der Klasse auf eine bestimmte Farbe setzen, die wir dann verwenden. Das haben wir ja eben im vorherigen Abschnitt gesehen. Über die Methode `ZFXD3D::GetWireColor` erhalten wir diese Farbe.

```
HRESULT ZFXD3DVCache::Flush(bool bUseShaders) {
    ZFXRENDERSTATE sm;
    HRESULT hr = ZFX_FAIL;
    if (m_nNumVerts <= 0) return ZFX_OK;

    // falls dieser Cache nicht aktiv ist
    if (m_pDad->GetActiveCache() != m_dwID) {
        // keine Shader
        if (!bUseShaders)
            m_pDevice->SetFVF(FVF_VERTEX);

        m_pDevice->SetIndices(m_pIB);
        m_pDevice->SetStreamSource(0, m_pVB, 0,
                                   m_nStride);
        m_pDad->SetActiveCache(m_dwID);
    } // [device->cache]

    // falls diese Skin nicht aktiv ist
    if (m_pDad->GetZFXD3D()->GetActiveSkinID() !=
        m_SkinID) {
        LPDIRECT3DTEXTURE9 pTex=NULL;
        ZFXMATERIAL *pMat = &m_pSkinMan->m_pMaterials[
                                   m_Skin.nMaterial];

        // WIREFRAME-MODUS; SPEZIAL-FALL
        if (!m_pDad->GetZFXD3D()->GetWireframeMode()) {

            // setze das Material für das Device
            D3DMATERIAL9 mat = {
                pMat->cDiffuse.fR, pMat->cDiffuse.fG,
                pMat->cDiffuse.fB, pMat->cDiffuse.fA,
                pMat->cAmbient.fR, pMat->cAmbient.fG,
                pMat->cAmbient.fB, pMat->cAmbient.fA,
                pMat->cSpecular.fR, pMat->cSpecular.fG,
                pMat->cSpecular.fB, pMat->cSpecular.fA,
                pMat->cEmissive.fR, pMat->cEmissive.fG,
                pMat->cEmissive.fB, pMat->cEmissive.fA,
            };
        }
    }
}
```



```

        pMat->fPower };
m_pDevice->SetMaterial(&mat);

// setze die Textur für das Device
for (int i=0; i<8; i++) {
    if (m_Skin.nTexture[i] != MAX_ID) {
        pTex = (LPDIRECT3DTEXTURE9)
            m_pSkinMan->m_pTextures[
                m_Skin.nTexture[i]].pData;
        m_pDevice->SetTexture(i, pTex);
    }
    else break;
} // for
}
else {
    ZFXCOLOR clrWire = m_pDad->GetZFXD3D()->
        GetWireColor();
    // setze das Material für das Device
    D3DMATERIAL9 matW = {
        clrWire.fR,clrWire.fG,clrWire.fB,clrWire.fA,
        clrWire.fR,clrWire.fG,clrWire.fB,clrWire.fA,
        0.0f,          0.0f,          0.0f,          1.0f,
        0.0f,          0.0f,          0.0f,          1.0f,
        1.0f };
    m_pDevice->SetMaterial(&matW);

    // keine Textur für das Device
    m_pDevice->SetTexture(0, NULL);
}

// Alphablending aktivieren
if (m_Skin.bAlpha) {
    m_pDevice->SetRenderState(D3DRS_ALPHAREF, 50);
    m_pDevice->SetRenderState(D3DRS_ALPHAFUNC,
        D3DCMP_GREATEREQUAL);
    m_pDevice->SetRenderState(D3DRS_SRCBLEND,
        D3DBLEND_SRCALPHA);
    m_pDevice->SetRenderState(D3DRS_DESTBLEND,
        D3DBLEND_INVSRCALPHA);
    m_pDevice->SetRenderState(
        D3DRS_ALPHATESTENABLE, TRUE);
    m_pDevice->SetRenderState(
        D3DRS_ALPHABLENDENABLE, TRUE);
}
else {
    m_pDevice->SetRenderState(
        D3DRS_ALPHATESTENABLE, FALSE);
    m_pDevice->SetRenderState(
        D3DRS_ALPHABLENDENABLE, FALSE);
}

```

```

    // Skin als aktiv markieren
    m_pDad->GetZFXD3D()->SetActiveSkinID(m_SkinID);
} // [device->skin]

// ENDLICH RENDERN!!!
sm = m_pDad->GetZFXD3D()->GetShadeMode();

// POINT-SPRITES RENDERN
if ( sm == RS_SHADE_POINTS ) {
    hr = m_pDevice->DrawPrimitive(
        D3DPT_POINTLIST,
        0, m_nNumVerts);
}
// LINESTRIP RENDERN
else if ( sm == RS_SHADE_HULLWIRE ) {
    hr = m_pDevice->DrawIndexedPrimitive(
        D3DPT_LINESTRIP,
        0, 0, m_nNumVerts,
        0, m_nNumVerts);
}
// POLYGONLISTE RENDERN
else { // RS_SHADE_SOLID || RS_SHADE_TRIWIRE
    hr = m_pDevice->DrawIndexedPrimitive(
        D3DPT_TRIANGLELIST,
        0, 0, m_nNumVerts,
        0, m_nNumIndis/3);
}

if ( FAILED(hr) ) return ZFX_FAIL;

// Zähler zurückstellen
m_nNumVerts = 0;
m_nNumIndis = 0;
return ZFX_OK;
}

```

Am Ende der Funktion müssen wir noch entscheiden, welchen Primitiventyp wir rendern wollen. Für Point-Sprites, also das Rendern von Vertices, benötigen wir keine Indexliste und ignorieren diese daher einfach. Man beachte auch die unterschiedlichen Zähler, die für den jeweiligen Primitiventyp benötigt werden.

*Das war's auch
schon*

Damit haben wir jetzt wirklich alles komplett, um große Mengen an Triangles effizient rendern zu können. Wir werden die Performance des Systems auch noch durch den Manager ein wenig ausbauen, so dass wir selbst dann effizient rendern können, wenn ein Benutzer seine Polygone mit ein paar verschiedenen Texturen kreuz und quer unsortiert zum Rendern schickt. Und genau das machen wir jetzt.

Vertex-Cache-Manager

On-the-fly Rendering-Management

Bislang haben wir lediglich eine Klasse implementiert, die uns das *On-the-fly*-Rendern von Vertexlisten erlauben soll. Damit ist gemeint, dass wir diese Klasse nur dazu benutzen, um zur Laufzeit des Programms jederzeit eine Vertexliste zum Rendern senden zu können. Dies ist aber eine Option, die man vermeiden sollte, wenn es nur irgendwie geht. Wir werden nun eine Klasse implementieren, die solche Vertex-Cache-Objekte managt, um das *On-the-fly*-Rendering dennoch halbwegs performant zu halten. Dazu werden die ankommenden Vertexlisten auf verschiedene Instanzen der Vertex-Cache-Klasse verteilt, um zunächst möglichst viele Polygone zu sammeln und diese dann zusammen zum Rendern zu schicken. Wie wir gleich sehen werden, sind die Möglichkeiten dieses *Batchings* jedoch recht begrenzt.

Weitere Aufgaben des Managers

Wenn man es so jedoch nicht machen sollte, dann muss es ja auch logischerweise einen besseren Weg geben, um etwas zu rendern. Den gibt es in der Tat, nämlich in Form der statischen Buffer. Hier gibt man die Vertexlisten zur Initialisierungszeit eines Programms an und lässt die Daten dann über die Laufzeit des Programms unverändert. Auch diese Möglichkeit soll unser Manager bieten, da dies die schnellste und beste Möglichkeit ist, etwas zu rendern. Das Folgende ist die Implementierung des eingangs definierten Interfaces für einen Vertex-Cache-Manager.



```
#define NUM_CACHES 10

class ZFXD3DVCManager : public ZFXVertexCacheManager {
public:
    ZFXD3DVCManager(ZFXD3DSkinManager *pSkinMan,
                    LPDIRECT3DDEVICE9 pDevice,
                    ZFXD3D *pZFXD3D, UINT nMaxVerts,
                    UINT nMaxIndis, FILE *pLog);
    ~ZFXD3DVCManager(void);

    HRESULT CreateStaticBuffer(ZFXVERTEXID VertexID,
                              UINT nSkinID,
                              UINT nVerts,
                              UINT nIndis,
                              const void *pVerts,
                              const WORD *pIndis,
                              UINT *pnID);

    HRESULT Render(ZFXVERTEXID VertexID,
                  UINT nVerts,
                  UINT nIndis,
                  const void *pVerts,
                  const WORD *pIndis,
                  UINT SkinID);
};
```

```

HRESULT Render(UINT nSBufferID);
HRESULT ForcedFlushAll(void);
HRESULT ForcedFlush(ZFXVERTEXID VertexID);

DWORD    GetActiveCache(void)
        { return m_dwActiveCache; }
void     SetActiveCache(DWORD dwID)
        { m_dwActiveCache = dwID; }
ZFXD3D*  GetZFXD3D(void) { return m_pZFXD3D; }

private:
    ZFXD3DSkinManager *m_pSkinMan;
    LPDIRECT3DDEVICE9  m_pDevice;
    ZFXD3D             *m_pZFXD3D;

    ZFXSTATICBUFFER    *m_pSB;
    UINT                m_nNumSB;
    ZFXD3DVCache        *m_CacheUU[NUM_CACHES];
    ZFXD3DVCache        *m_CacheUL[NUM_CACHES];
    DWORD               m_dwActiveCache;
    DWORD               m_dwActiveSB;
    FILE                *m_pLog;
}; // class

```

Unsere tatsächliche Implementierung verwendet also auch nur die Methoden, die das Interface vorgesehen hat. Zusätzlich gibt es hier lediglich Accessor-Methoden für das Attribut `m_dwActiveCache`, in dem gespeichert wird, welches der Vertex-Cache-Objekte zurzeit für das Direct3D-Device seinen Vertex- und Index-Buffer eingestellt hat.

Methoden

An Attributen haben wir zunächst einen ganzen Satz an Pointern auf die benötigten Objekte wie beispielsweise das `IDirect3DDevice9`-Objekt oder unser `ZFXD3D`-Objekt. Dann haben wir noch zwei Arrays für Vertex-Cache-Objekte und einen Pointer vom ominösen Datentyp `ZFXSTATICBUFFER`, den wir auch als Array nutzen werden. Diesen Datentyp nutzen wir, wer hätte es gedacht, für statische Buffer, und er ist wie folgt definiert:

Attribute

```

typedef struct ZFXSTATICBUFFER_TYPE {
    int     nStride;
    UINT    nSkinID;
    bool    bIndis;
    int     nNumVerts;
    int     nNumIndis;
    int     nNumTris;
    DWORD   dwFVF;
    LPDIRECT3DVERTEXBUFFER9 pVB;
    LPDIRECT3DINDEXBUFFER9  pIB;
} ZFXSTATICBUFFER;

```



**ZFXSTATIC-
BUFFER**

In dieser Struktur haben wir alle benötigten Daten, die wir für einen statischen Buffer brauchen. Das sind insbesondere Zähler für die Anzahl an Vertices und Indices und je ein Vertex- und Index-Buffer von Direct3D. Daneben haben wir aber auch mit `nStride` die Größe eines einzelnen Vertex-Objekts und mit `nSkinID` die Skin, die beim Rendern verwendet werden soll. Die statischen Buffer stellen wir zunächst ein wenig zurück, denn diese können wir später recht zügig abhandeln. Jetzt kümmern wir uns erst einmal darum, die dynamischen Buffer richtig in den Griff zu bekommen.

Erstellung und Freigabe eines Vertex-Cache-Managers**Sinn und Zweck
des Managers**

Worauf unser Manager hinausläuft, dürfte mittlerweile recht offensichtlich sein. Zum einen wollen wir über ihn bequem statische Vertexlisten rendern können. Also muss der Manager, ebenso wie unser Skin-Manager, die erstellten Buffer in seinen eigenen Attributen speichern und dem Anwender der DLL lediglich eine ID für jedes erzeugte Objekt zurückgeben. Genau wie bei den Skins braucht der Anwender selbst keinerlei Daten zu speichern mit Ausnahme der ID, mit der er wieder an die Daten herankommt. Zum anderen soll der Manager aber auch dynamisch angegebene Vertexlisten *on-the-fly* rendern können. Nun ist das Erzeugen eines Vertex- oder Index-Buffers eine relativ zeitraubende Operation, die man im zeitkritischen Code vermeiden sollte. Daher erzeugt unser Manager zur Initialisierungszeit zwei Arrays von Vertex-Cache-Objekten, durch die die dynamischen Vertexlisten dann zur Grafikkarte zum Rendern gezogen werden. Der Konstruktor sieht entsprechend wie folgt aus:



```
ZFXD3DVCManager::ZFXD3DVCManager(
    ZFXD3DSkinManager *pSkinMan,
    LPDIRECT3DDEVICE9 pDevice,
    ZFXD3D *pZFXD3D, UINT nMaxVerts,
    UINT nMaxIndis, FILE *pLog) {

    DWORD dwID=1;
    int i=0;

    m_pSB = NULL;
    m_nNumSB = 0;

    m_pLog = pLog;
    m_pDevice = pDevice;
    m_pZFXD3D = pZFXD3D;
    m_pSkinMan = pSkinMan;
    m_dwActiveCache = MAX_ID;
    m_dwActiveSB = MAX_ID;

    for (i=0; i<NUM_CACHES; i++) {
        m_CacheUU[i] = new ZFXD3DVCache(nMaxVerts,
                                         nMaxIndis,
                                         sizeof(VERTEX),
                                         pSkinMan, pDevice,
```

```

        this, dwID++, pLog);

    m_CacheUL[i] = new ZFXD3DVCache(nMaxVerts,
                                    nMaxIndis,
                                    sizeof(LVERTEX),
                                    pSkinMan, pDevice,
                                    this, dwID++, pLog);

    } // for
} // constructor

```

Das Problem hierbei ist unter anderem, dass ein Vertex-Buffer zum Zeitpunkt seiner Erstellung wissen muss, wie groß ein Vertex ist, der mit dem Buffer verwendet werden soll. Weiter oben hatten wir das Thema Vertex-Strukturen ja schon angesprochen. Hier arbeiten wir mit den beiden Typen VERTEX und LVERTEX. Da die Strukturen unterschiedliche Größen haben können, brauchen wir auch zwei verschiedene Parameterangaben bei der Erzeugung der Buffer. Aus diesem Grund müssen wir für jeden unterschiedlichen Vertex-Typ, den wir als Struktur definieren, eine eigene Behandlungsroutine wie eben an dieser Stelle. Daher brauchen wir zwei verschiedene Arrays aus Vertex-Cache-Objekten. Die Bezeichnung UU steht dabei für *Untransformed&Unlit* und UL für *Untransformed&Lit*.

*Warum zwei
Arrays?*

Im Destruktor müssen wir dann logischerweise wieder die beiden Arrays für die Vertex-Cache-Objekte freigeben, ebenso wie alle zwischenzeitlich erzeugten statischen Buffer.

```

ZFXD3DVCManager::~ZFXD3DVCManager(void) {
    UINT n=0;
    int i=0;

    // Freigabe des Speichers in den statischen Buffern
    for (n=0; n<m_nNumSB; n++) {
        if (m_pSB[n].pVB) {
            m_pSB[n].pVB->Release();
            m_pSB[n].pVB = NULL;
        }
        if (m_pSB[n].pIB) {
            m_pSB[n].pIB->Release();
            m_pSB[n].pIB = NULL;
        }
    }

    // Freigabe des Arrays der statischen Buffer
    if (m_pSB) { free(m_pSB); m_pSB=NULL; }

    // Freigabe der Vertex-Cache-Objekte
    for (i=0; i<NUM_CACHES; i++) {
        if (m_CacheUU[i]) {
            delete m_CacheUU[i];

```



```

        m_CacheUU[i] = NULL;
    }
    if (m_CacheUL[i]) {
        delete m_CacheUL[i];
        m_CacheUL[i] = NULL;
    }
} // for
} // destructor

```

Management dynamischer Render-Listen

*Butter bei die
Fische*

Jetzt sind wir im Herzen des Managers gelandet. Alle anderen Funktionen dieser Klasse sind nur Peanuts, aber das Management der dynamischen Vertexlisten ist die wichtigste eingebaute Logik in dieser Klasse. Der Anwender unserer DLL soll eine Möglichkeit haben, seine Polygone vollkommen unsortiert an unsere Render-Funktion zu senden. Nehmen wir einmal an, wir haben ein Objekt mit 500 Polygonen und drei verschiedenen Texturen. Ein unbedarfter Anwender wird alle 500 Polygone einzeln zum Rendern schicken, sträflicherweise noch nicht einmal nach Texturen geordnet. Unser Manager soll nun die Logik besitzen, diese Polygone nicht sofort an Direct3D zum Rendern weiterzureichen, sondern die Polygone erst einmal zu batchen. Alle ankommenden Polygone werden also zunächst zwischengespeichert und ausdrücklich *nicht* gerendert.

Batching-Kriterien

Tatsächlich gerendert werden sollen die Polygone nur dann, wenn sich in einem Cache eine bestimmte Mindestanzahl an Polygonen angestaut hat. Dies hebt zum einen die Effizienz dadurch, dass nicht einzelne Polygone über den Bus wandern. Zum anderen, und das ist wohl wichtiger, sorgt es auch dafür, dass die Render-States nicht dauernd wild hin- und hergeschwitcht werden müssen. Das ständige Ändern der aktiven Textur, des Materials und aller weiteren States sollte man vermeiden. Dies führt uns zu der Erkenntnis, dass wir alle Polygone natürlich nicht in einem Topf sammeln können. Vielmehr müssten wir eigentlich für jede Kombination aus States einen eigenen Topf anlegen. Für eine kleine Demo-Engine wie diese hier ist das aber ein wenig oversized. Daher werden wir als einzige Sortierkriterien die Textur und das Material (kurz die Skin) heranziehen, die die zu rendernden Polygone verwenden. Im Konstruktor haben wir nun die entsprechenden Arrays von Vertex-Cache-Objekten initialisiert, wobei jedes dieser Objekte ein Topf ist. In einen Topf dürfen daher nur solche Polygone zusammengebatcht werden, die dieselbe Skin verwenden.

Probleme

Klingt simpel? Nun, das ist es eigentlich auch, wenn nur das kleine Wörtchen »eigentlich« nicht wäre. Natürlich wissen wir im Vorfeld nicht, wie viele verschiedene Skins im Spiel sein werden. Daher haben wir in unserem Array mit NUM_CACHE auch nur zehn Cache-Objekte erzeugt. Wir können also so lange die Sortierung korrekt aufrechterhalten, wie der Anwender in einem Frame nur zehn verschiedene Skins verwendet. So eine willkürliche

Festlegung macht natürlich keinen Sinn, daher brauchen wir ein wenig mehr Logik. Wenn der Anwender bereits Polygone mit zehn verschiedenen Skins zum Rendern geschickt hat, dann haben wir also in jedem Topf bereits etwas drin. Kommen nun noch Polygone einer neuen, elften Skin, dann haben wir keinen freien Topf mehr. In diesem Fall müssen wir uns einen Topf nehmen und diesen ausleeren. Wir suchen also nach einer bestimmten Heuristik einen der zehn teilweise gefüllten Töpfe aus und schicken seinen Inhalt zum Rendern an Direct3D. Dann setzen wir den nun leeren Topf auf die neue Skin und füllen dort die zugehörigen neuen Polygone hinein.

Das klingt schon nach einem guten Plan. Aber es gibt noch mehr Haken. Eine Liste zu rendernder Vertices unterhält noch vielfältige andere Abhängigkeiten. Die verwendete Skin (Textur und Material) ist dabei ein wichtiger Faktor, aber nicht der einzige. Sämtliche zur Renderzeit aktiven Renderstates gehören ebenso dazu. Wir können den Inhalt in einem Topf also nur so lange batchen, bis sich an den Randbedingungen etwas ändert. Zu diesen Randbedingungen gehören beispielsweise auch die Weltmatrix, die Projektionsmatrix und die View-Matrix. Bevor wir eine dieser Matrizen ändern, müssen wir den Inhalt sämtlicher Töpfe an Direct3D zum Rendern schicken. Anderenfalls werden die Polygone anders dargestellt, als der Anwender das eigentlich im Sinn hatte. Aber zur Erhaltung der Integrität haben wir weiter oben ja schon einiges gesehen. Wir erinnern uns dunkel: Immer wenn wir beispielsweise eine der vitalen Matrizen für das Direct3D-Device geändert haben, dann haben wir die Funktion `ZFXD3DVCManager::ForcedFlushAll` aufgerufen. Diese schauen wir uns gleich an, aber es ist nun kein Geheimnis mehr, dass diese Funktion einfach alle Töpfe durchläuft und sie entleert (sprich: über Direct3D rendert).

Vorsicht!

Jetzt haben wir alle Fallstricke erkannt. Das große Geheimnis ist nun, wie wir den richtigen Topf auswählen, in den wir neu ankommende Polygone einsortieren. Wenn man ein wenig darüber nachdenkt, ist das nicht weiter kompliziert. Schließlich sortieren wir die Polygone ausschließlich über die Skin, die sie verwenden. Im Pseudo-Code sieht eine sinnvolle Sortierungsheuristik wie folgt aus:

**Betankungs-
heuristik**

```
for i=0 to alle Töpfe do {
    if (Topf[i] verwendet dieselbe Skin)
    {
        Füge Polygone zu Topf[i] hinzu
        return;
    }
    if (Topf[i] ist leer)
    {
        merke dir Topf[i] als leerer Topf
    }
    if (Topf[i] ist voller als bisher vollster Topf)
    {
```

```

        merke dir Topf[i] als bisher vollsten Topf
    }
}
if (mind. ein leerer Topf gefunden)
{
    Setze richtige Skin für den leeren Topf
    Füge Polygone dem leeren Topf hinzu
    return;
}
else
{
    Leere bisher vollsten Topf
    Setze richtige Skin für diesen nun leeren Topf
    Füge Polygone zu diesem nun leeren Topf hinzu
    return;
}

```

Wir durchlaufen zuerst das Array mit unseren Vertex-Cache-Objekten. Sobald wir einen Cache finden, der dieselbe Skin verwendet wie die nun kommenden Polygone, fügen wir diesem einfach die neuen Polygone noch hinzu und beenden die Funktion. Das ist der einfachste Fall, den wir haben. Während wir das Array durchlaufen, speichern wir gleichzeitig in zwei dafür vorgesehenen Pointern noch die Adresse eines leeren Topfes, falls es einen solchen noch im Array gibt, sowie die Adresse des Topfes, der zurzeit am vollsten ist.

Haben wir keinen Topf mit derselben Skin gefunden, dann schauen wir nach der Schleife nach, ob es noch einen komplett leeren Topf gibt. Ist das der Fall, dann legen wir die neue Skin für diesen Topf fest und füllen die Polygone in diesen Topf. Wenn wir aber auch keinen leeren Topf gefunden haben, dann müssen wir einen Topf zwangsweise leeren, um die neuen Polygone irgendwo speichern zu können. In diesem Fall nehmen wir den bisher vollsten Topf, rendern seinen Inhalt, womit er geleert ist, setzen die neue Skin für ihn fest und füllen die neuen Polygone in diesen Topf.

Einfacher als Pfannkuchenessen, oder? Und hier ist die Funktion in richtigem Code:

```

HRESULT ZFXD3DVCManager::Render(ZFXVERTEXID VertexID,
                                UINT nVerts,
                                UINT nIndis,
                                const void *pVerts,
                                const WORD *pIndis,
                                UINT SkinID) {
    ZFXD3DVCache **pCache=NULL,
    *pCacheEmpty=NULL,
    *pCacheFullest=NULL;
    int nEmptyVC = -1;

```



```

int nFullestVC = 0;

bool bShaders = m_pZFXD3D->UsesShaders();

// welcher Vertex-Typ wird verwendet?
switch (VertexID) {
    case VID_UU: { pCache = m_CacheUU; } break;
    case VID_UL: { pCache = m_CacheUL; } break;
    default: return ZFX_INVALIDID;
} // switch

pCacheFullest = pCache[0];

// aktiver Buffer kann ungültig werden
m_dwActiveSB = MAX_ID;

// SUCHE DEN GEEIGNETSTEN TOPF

// gibt es schon einen Cache mit dieser Skin?
for (int i=0; i<NUM_CACHES; i++) {
    if (pCache[i]->UsesSkin(SkinID))
        return pCache[i]->Add(nVerts, nIndis, pVerts,
                               pIndis, bShaders);

    // merke dir irgendeinen leeren Cache
    if (pCache[i]->IsEmpty())
        pCacheEmpty = pCache[i];

    // merke dir den vollsten Cache
    if (pCache[i]->NumVerts() >
        pCacheFullest->NumVerts())
        pCacheFullest = pCache[i];
}

// Kein Glück bisher. Gibt es einen leeren Cache?
if (pCacheEmpty) {
    pCacheEmpty->SetSkin(SkinID, bShaders);
    return pCacheEmpty->Add(nVerts, nIndis, pVerts,
                           pIndis, bShaders);
}

// Auch kein Glück, also leere den vollsten Cache.
pCacheFullest->Flush(bShaders);
pCacheFullest->SetSkin(SkinID, bShaders);
return pCacheFullest->Add(nVerts, nIndis, pVerts,
                          pIndis, bShaders);
} // Render

```

Wenn das mal nicht komplizierter klang, als es wirklich ist. Durch diese Heuristik haben wir nun sichergestellt, dass wir absolut nur dann wirklich rendern müssen, wenn es sich nicht vermeiden lässt. So lange es geht, speichern wir die Polygone in den einzelnen Töpfen ab, so dass wir den beschwerlichen Weg über den Bus so selten wie möglich und mit so vielen Polygonen wie möglich gleichzeitig gehen können. Das *on-the-fly* Rendern von Vertexlisten mag damit immer noch eine Bremse in der Engine sein, aber wir haben nun eine Art ABS in die Bremse eingebaut.

Buffer-Flushing

Oft zitiert, lang erwartet und nun endlich implementiert: die Funktionen zum Flushen der Vertex-Cache-Objekte. Wann immer wir einen State der Engine ändern, müssen wir alle Cache-Objekte leeren, die von diesen States abhängig sind. Insbesondere trifft dies auf das Ändern der vitalen Matrizen zu. Dazu haben wir einmal eine Funktion, mit der wir alle Cache-Objekte eines bestimmten Vertex-Typs entleeren können, also nur eines der bisher zwei Arrays. Daneben haben wir aber noch eine Funktion, die alle Arrays von Cache-Objekten entleert.



```
HRESULT ZFXD3DVCManager::ForcedFlush(
                                ZFXVERTEXID VertexID) {
    ZFXD3DVCache **pCache=NULL;
    HRESULT hr = ZFX_OK;
    int i=0;

    switch (VertexID) {
        case VID_UU: { pCache = m_CacheUU; } break;

        case VID_UL: { pCache = m_CacheUL; } break;

        // unbekannter Vertex-Typ
        default: return ZFX_INVALIDID;
    } // switch

    for (i=0; i<NUM_CACHES; i++)
        if (FAILED( pCache[i]->Flush(
                        m_pZFXD3D->UsesShaders()) ))
            hr = ZFX_FAIL;
    return hr;
} // ForcedFlush

HRESULT ZFXD3DVCManager::ForcedFlushAll(void) {
    HRESULT hr = ZFX_OK;
    bool    bShaders = m_pZFXD3D->UsesShaders();
    int     i;

    for (i=0; i<NUM_CACHES; i++)
```

```
if (!m_CacheUU[i]->IsEmpty() )
    if (FAILED( m_CacheUU[i]->Flush(bShaders) ))
        hr = ZFX_FAIL;

for (i=0; i<NUM_CACHES; i++)
    if (!m_CacheUL[i]->IsEmpty() )
        if (FAILED( m_CacheUL[i]->Flush(bShaders) ))
            hr = ZFX_FAIL;

return hr;
} // ForcedFlushAll
```

Wie ihr seht, tut sich hier eigentlich gar nichts, außer dass die Funktion `ZFXD3DVCache::Flush` aufgerufen wird, die den Inhalt der jeweiligen Instanz an `Direct3D` zum Rendern sendet. Beachtenswert ist hier lediglich die Übergabe des `bool`-Parameters, der angibt, ob die Engine durch Shader betrieben wird oder nicht, da die `Render`-Funktion darüber informiert sein muss, um die entsprechenden Einstellungen vorzunehmen.

Statische Buffer erzeugen und rendern

Neben den dynamischen Buffern gibt es natürlich noch die statischen Buffer. Statisch heißen sie deshalb, weil sich der Inhalt der Buffer möglichst *nicht* ändern, also statisch bleiben sollte. Jede Änderung, die man an dem Inhalt eines solchen Buffers vornimmt, erzeugt unnötigen Traffic über den langsamen Bus. Die statischen Buffer werden in der Regel im schnellen VRAM auf der Grafikkarte erzeugt, so lange dort noch Platz ist. Um die korrekte Verwendung eines solchen Buffers zu gewährleisten, oder besser gesagt zu erzwingen, werden wir keine Methoden implementieren, um auf den Inhalt eines erzeugten statischen Buffers zugreifen zu können. Ein solcher Zugriff ist nur in einem der beiden folgenden Fälle (oder einer Kombination beider Fälle) notwendig:

- ➔ nur lesender Zugriff
- ➔ nur schreibender Zugriff

Ein nur lesender Zugriff auf einen statischen Buffer ist aufgrund des Performance-Abfalls wenig sinnvoll. Wenn man die Trianglelisten, die in dem Buffer abgelegt sind, oft verwenden muss (beispielsweise für Kollisionsdetektion), empfiehlt es sich, eine Kopie der Daten in eigenen Strukturen zu speichern und den Buffer wirklich und ausschließlich nur zum Rendern zu verwenden. Die lesenden Zugriffe führt man dann über die kopierte Liste im System-RAM aus, was wesentlich schneller geht.

Ein schreibender Zugriff auf einen Buffer ist nur nötig, wenn sich die Form der im Buffer gespeicherten Polygone ändert. Das ist nur der Fall, wenn man die Geometrie wirklich nachhaltig ändert. Ausdrücklich nicht notwen-

dig ist das bei der Animation von Figuren, wie wir in einem späteren Kapitel sehen werden. Daraus lässt sich der Schluss herleiten, dass schreibende Zugriffe auf einen Buffer eher selten sein werden. Hier können wir mit unserem Interface den Umweg nutzen, den entsprechenden statischen Buffer zu löschen und einen neuen für den geänderten Inhalt zu erstellen. Sind öfter schreibende Zugriffe auf einen Buffer notwendig, so sollte man auf die dynamischen Buffer zurückkommen, denn hier ist der schreibende Zugriff schneller zu lösen als bei einem statischen Buffer.

Erstellung eines statischen Buffers

Langer Rede kurzer Sinn: Einen statischen Vertex- oder Index-Buffer erzeugen wir unter Direct3D ganz genauso wie einen dynamischen. Nur verwenden wir eben das Flag `D3DUSAGE_DYNAMIC` nicht. Damit weiß Direct3D, wie wir diesen Buffer verwenden wollen, und legt ihn entsprechend an. Die folgende Funktion dient dazu, über die `ZFXD3DVCManager`-Klasse einen statischen Buffer vom Typ `ZFXSTATICBUFFER` anzulegen und zeitgleich mit Vertex- und Indexlisten zu betanken sowie die verwendete Skin über ihre ID für den Buffer zu speichern. Gespeichert wird der Buffer in dem Attribut `m_pSB` der Klasse. Ganz analog wie bei unserem Skin-Manager allokalieren wir hier dynamisch Speicher für jeweils 50 neue `ZFXSTATICBUFFER`-Elemente in diesem Pointer, falls es in diesem nicht mehr genug Platz geben sollte. Danach verwenden wir diesen Pointer wie ein Array und das Attribut `m_nNumSB` als Zähler für das Array.



```
HRESULT ZFXD3DVCManager::CreateStaticBuffer(
    ZFXVERTEXID VertexID,
    UINT nSkinID, UINT nVerts,
    UINT nIndis, const void *pVerts,
    const WORD *pIndis, UINT *pnID) {

    HRESULT hr;
    DWORD   dwActualFVF;
    void    *pData;

    if (m_nNumSB >= (MAX_ID-1)) return ZFX_OUTOFMEMORY;

    // mehr Speicher allokalieren, falls nötig
    if ( (m_nNumSB % 50) == 0 ) {
        int n = (m_nNumSB+50)*sizeof(ZFXSTATICBUFFER);
        m_pSB = (ZFXSTATICBUFFER*)realloc(m_pSB, n);
        if (!m_pSB) return ZFX_OUTOFMEMORY;
    }

    m_pSB[m_nNumSB].nNumVerts = nVerts;
    m_pSB[m_nNumSB].nNumIndis = nIndis;
    m_pSB[m_nNumSB].nSkinID   = nSkinID;

    // Größe und Format eines Vertex
    switch (VertexID) {
        case VID_UU: {
```

```

        m_pSB[m_nNumSB].nStride = sizeof(VERTEX);
        m_pSB[m_nNumSB].dwFVF = FVF_VERTEX;
    } break;
case VID_UL: {
    m_pSB[m_nNumSB].nStride = sizeof(LVERTEX);
    m_pSB[m_nNumSB].dwFVF = FVF_LVERTEX;
    } break;
default: return ZFX_INVALIDID;
} // switch

// Index-Buffer erzeugen, falls nötig
if (nIndis > 0) {
    m_pSB[m_nNumSB].bIndis = true;
    m_pSB[m_nNumSB].nNumTris = int(nIndis / 3.0f);

    hr = m_pDevice->CreateIndexBuffer(
        nIndis * sizeof(WORD),
        D3DUSAGE_WRITEONLY,
        D3DFMT_INDEX16,
        D3DPPOOL_DEFAULT,
        &m_pSB[m_nNumSB].pIB,
        NULL);
    if (FAILED(hr)) return ZFX_CREATEBUFFER;

    // Betankung des Index-Buffers
    if (SUCCEEDED(m_pSB[m_nNumSB].pIB->Lock(
        0, 0, (void**)
        (&pData), 0))) {
        memcpy(pData, pIndis, nIndis*sizeof(WORD));
        m_pSB[m_nNumSB].pIB->Unlock();
    }
    else return ZFX_BUFFERLOCK;
}
else {
    m_pSB[m_nNumSB].bIndis = false;
    m_pSB[m_nNumSB].nNumTris = int(nVerts / 3.0f);
    m_pSB[m_nNumSB].pIB = NULL;
}

// Kein Bedarf für FVF, falls Shader verwendet
if (m_pZFXD3D->UsesShaders()) dwActualFVF = 0;
else dwActualFVF = m_pSB[m_nNumSB].dwFVF;

// Vertex-Buffer erstellen
hr = m_pDevice->CreateVertexBuffer(
    nVerts*m_pSB[m_nNumSB].nStride,
    D3DUSAGE_WRITEONLY, dwActualFVF,
    D3DPPOOL_DEFAULT,
    &m_pSB[m_nNumSB].pVB, NULL);
if (FAILED(hr)) return ZFX_CREATEBUFFER;

```

```

// Betankung des Vertex-Buffers
if (SUCCEEDED(m_pSB[m_nNumSB].pVB->Lock(
    0, 0, (void**)
    (&pData), 0))) {
    memcpy(pData, pVerts, nVerts*m_pSB[
        m_nNumSB].nStride);
    m_pSB[m_nNumSB].pVB->Unlock();
}
else return ZFX_BUFFERLOCK;

(*pnID) = m_nNumSB;
m_nNumSB++;
return ZFX_OK;
} // CreateStaticBuffer

```

Vertex-Typ und Shader

Dass die Funktion ein wenig lang erscheint, liegt daran, dass wir zunächst unterscheiden müssen, welchen Vertex-Typ wir verwenden. Dies benötigen wir, um die Größe des Vertex-Buffers korrekt bestimmen zu können. Eine weitere Unterscheidung ist nötig. Wenn wir nicht mit Vertex-Shadern arbeiten, dann müssen wir das FVF (Flexible-Vertex-Format) bei der Erstellung des Vertex-Buffers mit angeben.

Die Funktion `ZFXD3DVCManger::CreateStaticBuffer` liefert übrigens als Call-by-Reference-Parameter `pnID` die Inventarnummer des erzeugten Buffers an den Aufrufer zurück. Diese ID muss der Aufrufer speichern, denn anders kommt er nicht mehr an den erzeugten Buffer heran.

Rendern statischer Buffer

Um nun einen statischen Buffer zu rendern, verwendet der Aufrufer die ID, die er von der `ZFXD3DVCManger::CreateStaticBuffer`-Funktion erhalten hat, und ruft dann ganz einfach die Methode `ZFXD3DVCManger::Render` mit dieser ID als Parameter auf. Diese Funktion prüft zunächst, ob der entsprechende Buffer nicht schon aktiv ist, also als Buffer für Direct3D eingestellt. Ist das nicht der Fall, dann erledigt die Funktion dies noch schnell über die Direct3D-Methoden `IDirect3DDevice9::SetIndices` für den Index-Buffer und `IDirect3DDevice9::SetStreamSource` für den Vertex-Buffer. Daran schließt sich eine Abfrage der zurzeit aktiven Skin an, bei der gegebenenfalls noch die Skin, die der statische Buffer verwendet, aktiviert wird. Werden keine Vertex-Shader verwendet, dann muss noch das FVF dem Direct3D-Device bekannt gemacht werden. Anderenfalls wurde bei der Aktivierung eines Vertex-Shaders ja bereits die entsprechende Vertex-Deklaration aktiviert.

Dann kann der Buffer bereits gerendert werden. Natürlich verwenden wir hier dieselben Einstellungen bezüglich der Textur und der Farbe für das Material, falls wir uns im Wireframe-Modus befinden.



```

HRESULT ZFXD3DVCManager::Render(UINT nID) {
    HRESULT hr=ZFX_OK;

    ZFXRENDERSTATE sm = m_pZFXD3D->GetShadeMode();

    // Aktiver Cache ist jetzt ungültig
    m_dwActiveCache = MAX_ID;

    // Aktiviere statischen Buffer, falls inaktiv
    if (m_dwActiveSB != nID) {
        // Indices verwendet?
        if (m_pSB[nID].bIndis)
            m_pDevice->SetIndices(m_pSB[nID].pIB);

        m_pDevice->SetStreamSource(0, m_pSB[nID].pVB,
                                   0, m_pSB[nID].nStride);
        m_dwActiveSB = nID;
    }

    // Verwendet das Device bereits diese Skin?
    if (m_pZFXD3D->GetActiveSkinID() !=
        m_pSB[nID].nSkinID) {
        // Material als aktiv markieren
        ZFXSKIN *pSkin = &m_pSkinMan->m_pSkins[
            m_pSB[nID].nSkinID];

        // SPEZIALFALL WIREFRAME-MODUS
        if (sm == RS_SHADE_SOLID) {
            // setze das Material für das Device
            ZFXMATERIAL *pMat = &m_pSkinMan->
                m_pMaterials[pSkin->nMaterial];
            D3DMATERIAL9 mat = {
                pMat->cDiffuse.fR, pMat->cDiffuse.fG,
                pMat->cDiffuse.fB, pMat->cDiffuse.fA,
                pMat->cAmbient.fR, pMat->cAmbient.fG,
                pMat->cAmbient.fB, pMat->cAmbient.fA,
                pMat->cSpecular.fR, pMat->cSpecular.fG,
                pMat->cSpecular.fB, pMat->cSpecular.fA,
                pMat->cEmissive.fR, pMat->cEmissive.fG,
                pMat->cEmissive.fB, pMat->cEmissive.fA,
                pMat->fPower };
            m_pDevice->SetMaterial(&mat);

            // setze die Textur für das Device
            for (int i=0; i<8; i++) {
                if (pSkin->nTexture[i] != MAX_ID)
                    m_pDevice->SetTexture(i,
                        (LPDIRECT3DTEXTURE9)
                        m_pSkinMan->m_pTextures[
                            pSkin->nTexture[i]].pData);
            }
        }
    }
}

```

```

    }
}
else {
    ZFXCOLOR clrWire = m_pZFXD3D->GetWireColor();
    // setze das Material für das Device
    D3DMATERIAL9 matW = {
        clrWire.fR,clrWire.fG,clrWire.fB,clrWire.fA,
        clrWire.fR,clrWire.fG,clrWire.fB,clrWire.fA,
        0.0f,          0.0f,          0.0f,          1.0f,
        0.0f,          0.0f,          0.0f,          1.0f,
        1.0f );
    m_pDevice->SetMaterial(&matW);

    // keine Textur für das Device
    m_pDevice->SetTexture(0, NULL);
}

// Alpha-States festlegen
if (pSkin->bAlpha) {
    m_pDevice->SetRenderState(D3DRS_ALPHAREF, 50);
    m_pDevice->SetRenderState(D3DRS_ALPHAFUNC,
        D3DCMP_GREATEREQUAL);
    m_pDevice->SetRenderState(D3DRS_SRCBLEND,
        D3DBLEND_SRCALPHA);
    m_pDevice->SetRenderState(D3DRS_DESTBLEND,
        D3DBLEND_INVSRCALPHA);
    m_pDevice->SetRenderState(
        D3DRS_ALPHATESTENABLE, TRUE);
    m_pDevice->SetRenderState(
        D3DRS_ALPHABLENDENABLE, TRUE);
}
else {
    m_pDevice->SetRenderState(
        D3DRS_ALPHATESTENABLE, FALSE);
    m_pDevice->SetRenderState(
        D3DRS_ALPHABLENDENABLE, FALSE);
}

// Die aktive Skin wurde nun geändert
m_pZFXD3D->SetActiveSkinID(m_pSB[nID].nSkinID);
} // [device->skin]

// Falls ein Shader verwendet werden, dann muss er
// über ActivateVShader() aktiviert werden. Sonst FVF
if (!m_pZFXD3D->UsesShaders())
    m_pDevice->SetFVF(FVF_VERTEX);

// indizierte Primitive
if (m_pSB[nID].bIndis) {
    if ( sm == RS_SHADE_POINTS ) {
        hr = m_pDevice->DrawPrimitive(

```

```

        D3DPT_POINTLIST, 0,
        m_pSB[nID].nNumVerts);
    }
    else if ( sm == RS_SHADE_HULLWIRE ) {
        hr = m_pDevice->DrawIndexedPrimitive(
            D3DPT_LINESTRIP, 0,
            0, m_pSB[nID].nNumVerts,
            0, m_pSB[nID].nNumVerts);
    }
    else { // RS_SHADE_SOLID || RS_SHADE_TRIWIRE
        hr = m_pDevice->DrawIndexedPrimitive(
            D3DPT_TRIANGLELIST, 0,
            0, m_pSB[nID].nNumVerts,
            0, m_pSB[nID].nNumTris);
    }
}
// nicht indizierte Primitive
else {
    if ( sm == RS_SHADE_POINTS ) {
        hr = m_pDevice->DrawPrimitive(
            D3DPT_POINTLIST, 0,
            m_pSB[nID].nNumVerts);
    }
    else if ( sm == RS_SHADE_HULLWIRE ) {
        hr = m_pDevice->DrawPrimitive(
            D3DPT_LINESTRIP,
            m_pSB[nID].nNumVerts,
            m_pSB[nID].nNumVerts);
    }
    else { // RS_SHADE_SOLID || RS_SHADE_TRIWIRE
        hr = m_pDevice->DrawPrimitive(
            D3DPT_TRIANGLELIST,
            m_pSB[nID].nNumVerts,
            m_pSB[nID].nNumTris);
    }
}
return hr;
} // Render

```

Man kann hier auch schön erkennen, dass die Angabe einer Indexliste bei unseren statischen Buffern optional möglich, aber nicht erforderlich ist. Gibt man bei der Erstellung eines statischen Buffers eine Indexliste an, so wird ein Index-Buffer miterzeugt und entsprechend auch beim Rendern verwendet. Gibt man keine Indices an, so wird lediglich ein Vertex-Buffer erstellt und beim Rendern verwendet. Der gerenderte Primitiventyp ist dabei ebenso wie bei dem Rendern aus einem Cache-Objekt von der Shade-Mode-Einstellung des ZFXD3D-Device-Objekts abhängig.

*Indiziert versus
nicht indiziert*

**Zwischen-
Resümee**

Damit ist unser kleiner Vertex-Manager auch schon fertig. Über das Interface `ZFXVertexCacheManager` können wir nun performante statische Buffer erstellen und rendern lassen. Auch das *On-the-fly*-Rendering ist so effizient wie nur möglich integriert, wenn man das Rendern möglichst allgemein halten möchte. Der Manager ist als Attribut des Interfaces `ZFXRenderDevice` implementiert und wird von diesem initialisiert und auch wieder freigegeben. Der Anwender unserer DLL kann also über sein Interface-Objekt einen Pointer auf den Manager erhalten (genauso wie eingangs in diesem Kapitel für den Skin-Manager gezeigt) und dann dessen Interface-Funktionen verwenden.

```
// pZFXDevice ist ein valide initialisiertes Objekt
// vom Typ ZFXRenderDevice, myID ist die ID eines
// zuvor erzeugten statischen Buffers
pZFXDevice->GetRenderManager()->Render(myID);
```

6.8 Rendern von Text, Punkten und Linien

Wir haben zwar schon viel Arbeit investiert, aber gewisse Funktionalitäten fehlen unserem Render-Device immer noch. An dieser Stelle bereichern wir die Render-Funktionalität unserer Grafik-Engine noch um das Rendern von Text, Punkten und Linien. Ersteres wird man recht häufig benötigen, Letzteres eher selten. Aber schließlich wollen wir allen etwas bieten können.

Die in diesem Abschnitt entwickelten Funktionen werden allesamt nicht unbedingt auf Geschwindigkeit optimiert. Der Komfort in der Anwendung bleibt aber gewohnt hoch durch die einfache Bedienung und Flexibilität. Die hier gezeigten Implementierungen sollte man aber später durch eigene, geschwindigkeitsoptimierte Varianten ersetzen.

Fonts anlegen und Text rendern

Font-Engines

Was wäre eine 3D-Engine, die keinen Text rendern könnte? Richtig, sie wäre eine verkrüppelte 3D-Engine. Diese Funktionalität braucht nahezu jedes Programm, und sei es lediglich zum Ausgeben der Frame-Rate. Nun gibt es fast unendlich viele verschiedene Möglichkeiten, Text zu rendern. Einige sind effizienter als andere. Am sinnvollsten wäre es natürlich, man würde sich seine eigene Font-Klasse implementieren oder wenigstens entsprechende Funktionen.

3D-Fonts

Im DirectX SDK findet sich auch eine Klasse namens `CD3DFont`, welche man sich auf alle Fälle einmal anschauen sollte. In dieser Klasse wird gezeigt, wie man vollautomatisch einen Font nach den Wünschen des Anwenders erzeugen (Schriftart, -größe, -eigenschaften usw.) und eine Textur daraus erstellen kann. Beim Rendern von Text werden die entsprechenden Buchstaben

dann aus der Textur ausgelesen und auf echte 3D-Rechtecke gelegt. Dadurch erhält man nicht nur einen komfortablen Font für die Ausgabe von Text, man kann diesen Text auch als vollwertiges 3D-Objekt behandeln und ihn frei im Raum verschieben und rotieren. Dieses Beispiel empfehle ich deswegen so, weil wir hier nicht den Platz haben, ausführlich auf Text einzugehen.

Wir bedienen uns hier der Klasse `D3DXFont`. Diese arbeitet intern mit dem Windows API-GDI und ist daher nicht wirklich zu empfehlen, um viel Text auszugeben. Aber wir können hier mit wenigen Zeilen Code einen vollwertigen 2D-Font erzeugen und verwenden. Daher wählen wir dies als bessere Alternative. Aber ich empfehle dennoch, den Font später durch eine eigene Implementierung zu ersetzen. Dazu fügen wir nun in der Klassen-Definition `ZFXD3D` die folgenden beiden Member und Methoden ein:

D3DXFont

```
LPD3DXFONT *m_pFont;           // Font-Objekte
UINT m_nNumFonts;              // Anzahl Fonts

HRESULT CreateFont(const char*, int, bool, bool,
                  bool, DWORD, UINT*);

HRESULT DrawText(UINT, int, int, UCHAR, UCHAR,
                UCHAR, char*,...);
```

Natürlich beleidige ich eure Intelligenz jetzt nicht, indem ich erwähne, dass die beiden Methoden natürlich auch als rein virtuelle Methoden im Interface `ZFXRenderDevice` deklariert werden müssen. Das haben wir ja inzwischen oft genug gesehen. Die Parameter der beiden Methoden schauen wir uns gleich an. Hier sei noch erwähnt, dass ein Anwender natürlich beliebig viele verschiedene Fonts in unserer Engine anlegen kann. Daher haben wir hier einen Pointer auf die Font-Objekte und einen Zähler für die bereits angelegten Fonts. Aber schauen wir uns das Anlegen einmal genau an.

Als Parameter müssen wir der Methode zuerst einmal angeben, welchen Font-Typ wir erzeugen möchten. Das kann beispielsweise der Sting »Arial« sein. Als Nächstes folgt dann die Angabe der Dicke des Fonts (0 entspricht Default, 700 ist fette Formatierung) sowie drei `bool`-Parameter für kursive, unterstrichene oder durchgestrichene Formatierung. Abschließend gibt man noch die Höhe des Fonts in logischen Koordinaten an sowie einen Zeiger auf die Adresse, an der man die ID des erzeugten Fonts für die spätere Anwendung speichert.

Parameter

Aus all diesen Angaben erstellt unsere Funktion dann einen Font und speichert ihn im Attribut `m_pFont` ab, damit der Anwender ihn später verwenden kann.



```

HRESULT ZFXD3D::CreateFont(const char *chType,
                           int nWeight, bool bItalic,
                           bool bUnderline, bool
                           bStrike, DWORD dwSize,
                           UINT *pID) {

    HRESULT hr;
    HFONT    hFont;
    HDC      hDC;
    int      nHeight;

    if (!pID) return ZFX_INVALIDPARAM;

    hDC = GetDC( NULL );
    nHeight = -MulDiv(dwSize, GetDeviceCaps(hDC,
        LOGPIXELSY), 72);
    ReleaseDC(NULL, hDC);

    hFont = ::CreateFont(nHeight, // log. Höhe
        0, 0, 0, // Breite durchschn.
        nWeight, // Dicke
        bItalic, bUnderline, bStrike,
        DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
        DEFAULT_PITCH | FF_DONTCARE, chType);
    if (hFont == NULL) return ZFX_FAIL;

    m_pFont = (LPD3DFONT*)realloc(m_pFont,
        sizeof(LPD3DFONT)*(m_nNumFonts+1));

    hr = D3DXCreateFont(m_pDevice, hFont,
        &m_pFont[m_nNumFonts]);
    DeleteObject(hFont);

    if (SUCCEEDED(hr)) {
        (*pID) = m_nNumFonts;
        m_nNumFonts++;
        return ZFX_OK;
    }
    else return ZFX_FAIL;
} // CreateFont

```

Wie ihr seht, gibt es auch hier keine großen Mysterien. Den Font erstellen wir mit der WinAPI-Funktion `CreateFont()`. Da unsere Funktion lustigerweise genauso heißt, benötigen wir vor dem Aufruf noch den Scope-Operator, um die richtige Funktion zu erwischen. Wer bei der Anwendung von Fonts der WinAPI-Funktion noch ein wenig Nachholbedarf hat, der sollte dazu ein wenig in der MSDN stöbern. Ist das alles glatt gegangen, dann erstellen wir ein `D3DXFont`-Objekt durch Aufruf der `D3DX`-Funktion `D3DXCreateFont()`. Und das war's auch schon.

Über die ID, die unsere Funktion zurückgegeben hat, können wir mit unserem Render-Device nun auch Text über diesen Font rendern lassen.² Nun lüften wir auch das Geheimnis der ellenlangen Parameterliste der ZFXD3D::DrawText-Methode: Der erste Parameter ist die ID eines vorher erzeugten Font-Objekts. Als Nächstes folgen die x- und y-Bildschirm-Koordinaten der Position, an der der Text erscheinen soll. Die nächsten drei Parameter geben die RGB-Werte der Farbe an, mit der gerendert werden soll. Abschließend folgen dann endlich der Formatierungsstring für den auszugebenden Text und eine beliebige Anzahl an auszugebenden Variablen. Die letzten beiden Parameter werden genauso verwendet wie in der ANSI-C-Funktion `fprintf()`.

```
HRESULT ZFXD3D::DrawText(UINT nID, int x, int y,
                        UCHAR r, UCHAR g, UCHAR b,
                        char *ch, ...) {
    RECT rc = { x, y, 0, 0 };
    char cch[1024];
    char *pArgs;

    // Variablen in den String packen
    pArgs = (char*) &ch + sizeof(ch);
    vsprintf(cch, ch, pArgs);

    if (nID >= m_nNumFonts) return ZFX_INVALIDPARAM;

    m_pFont[nID]->Begin();
    // berechne die Größe des Textes
    m_pFont[nID]->DrawText(cch, -1, &rc, DT_SINGLELINE
                          | DT_CALCRECT, 0);
    // jetzt rendere den Text
    m_pFont[nID]->DrawText(cch, -1, &rc,
                          DT_SINGLELINE,
                          D3DCOLOR_ARGB(255,r,g,b));
    m_pFont[nID]->End();
    return ZFX_OK;
} // DrawText
```

*Ausgabe von Text
über einen Font*



Zum eigentlichen Rendern des Textes verwenden wir die Methode `ID3DXFont::DrawText`. Hierbei ist ein Rechteck anzugeben, in dem der Text auszugeben ist. Je nach Font kann das Rechteck aber unterschiedliche Größen haben, daher bietet diese Methode die Möglichkeit, zuerst die Größe des Dreiecks zu bestimmen. Gibt man das Flag `DT_CALCRECT` an, dann rendert die Funktion nicht, sondern gibt im dritten Parameter das entsprechend bemessene Rechteck zurück. Daher rufen wir die Methode noch ein zweites Mal auf, diesmal mit dem korrekt berechneten Rechteck zum Rendern des Textes.

Rendern

² Sonst wäre der Font ja auch relativ witzlos.

Das war es auch schon für den Text. Der Anwender unserer DLL kann nun beliebig viele verschiedene Fonts erzeugen, ob dick oder dünn, kursiv oder durchgestrichen, groß oder klein – alles kein Problem. Dafür erhält er zu jedem Font eine ID zurück und kann diese ID dann zusammen mit Koordinaten und der gewünschten Farbe und dem Text zum Rendern verwenden.

6.9 Punktlisten rendern

*Viele bunte
Punkte*

Eventuell kommt ein Entwickler einmal in die Situation, in der er wirklich einzelne Punkte am Bildschirm rendern muss. In der Regel wird man heutzutage für Dinge wie beispielsweise Hintergrundsterne, Partikel und Ähnliches keine Punkte mehr verwenden, sondern Billboards oder Point-Sprites mit schönen Texturen. Wenn man aber doch einmal eine Liste von pixelgroßen Punkten ohne Texturen in einer bestimmten Farbe rendern möchte, was höchst selten sein wird, dann sollte man das auch mit unserer Engine tun können. Im Interface des Vertex-Cache-Managers fügen wir daher die folgende Methode ein:

```
virtual HRESULT RenderPoints(
    ZFXVERTEXID    VertexID,
    UINT           nVerts,
    const void      *pVerts,
    const ZFXCOLOR *pClr1)=0;
```

Parameter

Übergeben muss man dieser Methode lediglich den Typ der verwendeten Vertices, die Anzahl der Vertices sowie einen Pointer auf eine Liste von Vertices und einen Pointer auf ein ZFXCOLOR-Objekt. Ein Vertex hält dabei die Koordinaten eines Punktes, den man rendern möchte.

Implementierung

Im Folgenden zeige ich gleich, wie man diese Funktion implementiert. Vorweg schicke ich nochmals den Hinweis, dass diese Funktion selten verwendet werden wird und daher vollkommen unoptimiert bleibt. Das bedeutet, dass wir an dieser Stelle schlicht und einfach die `IDirect3DDevice9::DrawPrimitiveUP`-Funktion verwenden. So weit ist die Umsetzung der Methode also keine Kunst. Eigentlich leiten wir den Aufruf nur an Direct3D weiter. Wenn da nicht ein großes ABER wäre. Wir müssen bei der Umsetzung der Methode insbesondere darauf achten, dass wir nicht das sorgsam gespannte Netz aus gespeicherten Zuständen unserer Engine torpedieren. Die Vertex-Cache-Objekte sorgen ja in der Regel dafür, dass sie selbst korrekt ihre Texturen und Buffer aktivieren wenn dies nötig sein sollte. Wenn wir aber zwischendurch eventuell Linien und Punkte quer reinrendern, dann müssen wir darauf achten, dem Vertex-Cache-Manager auch mitzuteilen, dass alle seine Zustände nun hinfällig sind und jeder Cache sich auf alle Fälle vor einem erneuten Rendern korrekt aktivieren muss. Um dies sicherzustellen, verwenden wir nun die Methode `ZFXD3DVCManager::InvalidateStates`, die die entsprechenden Attribute als invalide markiert.


```
void ZFXD3DVCManager::InvalidateStates(void) {
    m_pZFXD3D->SetActiveSkinID(MAX_ID);
    m_dwActiveSB    = MAX_ID;
    m_dwActiveCache = MAX_ID;
}
```



Danach bleibt noch die Frage, ob die Engine zurzeit im Modus mit Shadern läuft. Entsprechend müssen wir dann entweder die Default-Shader mit normaler Transformation aktivieren oder eben das flexible Vertex-Format. Aber dann können wir auch schon rendern.

```
HRESULT ZFXD3DVCManager::RenderPoints(
                                ZFXVERTEXID  VID,
                                UINT           nVerts,
                                const void     *pVerts,
                                const ZFXCOLOR *pClr) {

    D3DMATERIAL9 mtrl;
    DWORD        dwFVF;
    int          nStride;

    // zurzeit aktive Zustände werden ungültig
    InvalidateStates();

    memset(&mtrl, 0, sizeof(D3DMATERIAL9));
    mtrl.Diffuse.r = mtrl.Ambient.r = pClr->r;
    mtrl.Diffuse.g = mtrl.Ambient.g = pClr->g;
    mtrl.Diffuse.b = mtrl.Ambient.b = pClr->b;
    mtrl.Diffuse.a = mtrl.Ambient.a = pClr->a;

    m_pDevice->SetMaterial(&mtrl);
    m_pDevice->SetTexture(0, NULL);

    switch (VID) {
        case VID_UU: {
            nStride = sizeof(VERTEX);
            dwFVF   = FVF_VERTEX;
        } break;
        case VID_UL: {
            nStride = sizeof(LVERTEX);
            dwFVF   = FVF_LVERTEX;
        } break;
        default: return ZFX_INVALIDID;
    } // switch

    // Shader oder FVF
    if ( m_pZFXD3D->UsesShaders() ) {
        m_pZFXD3D->ActivateVShader(0, VID);
        m_pZFXD3D->ActivatePShader(0);
    }
}
```



```

else m_pDevice->SetFVF(dwFVF);

// rendere Punktliste
if (FAILED(m_pDevice->DrawPrimitiveUP(
    D3DPT_POINTLIST, nVerts,
    pVerts, nStride)))
    return ZFX_FAIL;
return ZFX_OK;
} // RenderPoints

```

Man beachte hierbei, dass der Anwender immer noch die Aufgabe zu erfüllen hat, die entsprechende Transformationsmatrix zu aktivieren oder zu deaktivieren – abhängig davon, wo die Linie gerendert werden soll.

Linienlisten rendern

*Aus Punkten
werden Linien*

Analog wie beim Rendern von Linien kann es einem Anwender unseres Render-Device auch einmal in den Sinn kommen, Linien zu rendern. Auch das ist ein Feature, das man bei heutigen 3D-Engines eher selten brauchen wird. Allerdings bekommt dieses Feature wieder mehr Bedeutung, wenn man beispielsweise an Editoren denkt. Hier hat man gern ein Gitter zur Markierung der Koordinaten in der 2D-Ansicht so wie Kästchen- oder Millimeterpapier beim Real-Life-Zeichnen. Die neue Methode für das Vertex-Cache-Manager-Interface sieht so aus:

```

virtual HRESULT RenderLines(
    ZFXVERTEXID    VertexID,
    UINT           nVerts,
    const void      *pVerts,
    const ZFXCOLOR *pClr,
    bool           bStrip)=0;

```

Implementierung

Die folgende Implementierung ist zu 99,9% identisch mit der Methode für das Rendern von Punkten. Einzig und allein zwei Parameter der eigentlichen Render-Methode DrawPrimitiveUP() muss man hier ändern. Daher spare ich mir jeden weiteren Kommentar, bis auf die Anmerkung, dass wir hier nicht nur Linienlisten ermöglichen, sondern auch Linien-Strips. Bei einer Liste bilden immer die zwei folgenden Vertices der Vertexliste zusammen eine unabhängige Linie. Bei einem Strip verläuft eine Linie immer von einem Vertex der Vertexliste zum nachfolgenden, so dass auf alle Fälle eine zusammenhängende Polylinie entsteht.



```

HRESULT ZFXD3DVCManager::RenderLines(
    ZFXVERTEXID    VID,
    UINT           nVerts,
    const void      *pVerts,
    const ZFXCOLOR *pClr,
    bool           bStrip) {

```

```

D3DMATERIAL9 mtrl;
DWORD        dwFVF;
int          nStride;

// zurzeit aktive Zustände werden ungültig
InvalidateStates();

if (pClr) {
    memset(&mtrl, 0, sizeof(D3DMATERIAL9));
    mtrl.Diffuse.r = mtrl.Ambient.r = pClr->fR;
    mtrl.Diffuse.g = mtrl.Ambient.g = pClr->fG;
    mtrl.Diffuse.b = mtrl.Ambient.b = pClr->fB;
    mtrl.Diffuse.a = mtrl.Ambient.a = pClr->fA;
    m_pDevice->SetMaterial(&mtrl);
}
m_pDevice->SetTexture(0, NULL);

switch (VID) {
    case VID_UU: {
        nStride = sizeof(VERTEX);
        dwFVF   = FVF_VERTEX;
    } break;
    case VID_UL: {
        nStride = sizeof(LVERTEX);
        dwFVF   = FVF_LVERTEX;
    } break;
    default: return ZFX_INVALIDID;
} // switch

// Shader oder FVF
if ( m_pZFXD3D->UsesShaders() ) {
    m_pZFXD3D->ActivateVShader(0, VID);
    m_pZFXD3D->ActivatePShader(0);
}
else m_pDevice->SetFVF(dwFVF);

// rendere Punktliste
if (!bStrip) {
    if (FAILED(m_pDevice->DrawPrimitiveUP(
        D3DPT_LINELIST, nVerts/2,
        pVerts, nStride)))
        return ZFX_FAIL;
}
else {
    if (FAILED(m_pDevice->DrawPrimitiveUP(
        D3DPT_LINESTRIP, nVerts-1,
        pVerts, nStride)))
        return ZFX_FAIL;
}

```

```
return ZFX_OK;
} // RenderLines
```

Damit haben wir nun eigentlich alle elementaren Render-Funktionen mit unserem Device abgedeckt. Daher können wir uns nun mit großen Schritten auf das Ende des Kapitels zubewegen.



Die beiden Methoden `ZFXVertexCacheManager::RenderLines` und `ZFXVertexCacheManager::RenderPoints` setzen den aktiven Shader auf den Default-Shader (falls die Grafikkarte überhaupt Shader unterstützt). Wenn die Engine also über eigene Shader des Anwenders betrieben wird, dann muss der Anwender nach der Verwendung dieser Funktion vor dem Rendern anderer Objekte den entsprechend gewünschten Shader wieder aktivieren.

6.10 Darstellung einer Szene

*Änderungen am
Code aus Kapitel 3*

Nachdem wir nun unsere DLL um so viel Funktionalität bereichert haben, müssen wir natürlich noch einige Dinge am bisherigen Code ändern. Im Konstruktor müssen wir die hinzugekommenen Attribute mit 0 bzw. NULL initialisieren. In der Funktion `ZFXD3D::Release` müssen wir auch alle entsprechenden Pointer wieder freigeben. Insbesondere sind hier die Vertex- und Pixel-Shader-Objekte und der Skin-Manager sowie der Vertex-Manager zu nennen. Da das aber recht rudimentäre Änderungen sind, verweise ich an dieser Stelle einfach auf den kompletten Quelltext auf der CD-ROM zu diesem Buch. Auch das Loggen aufgetretener Fehler ist im Original-Quelltext ausführlicher vorhanden. Ein Blick darauf lohnt sich also allein schon deshalb.

Bei der Funktion `ZFXD3D::Go`, die beim erfolgreichen Beenden des Auswahldialogs für den Bildschirmmodus aufgerufen wird, ändern wir nur den letzten Abschnitt. Anstelle des bisherigen Rückgabewertes `ZFX_OK` rufen wir jetzt eine weitere neue Funktion unserer Klasse auf, nämlich `ZFXD3D::OneTimeInit`.



```
HRESULT ZFXD3D::Go(void) {
    [...]

    m_dwWidth  = m_d3dpp.BackBufferWidth;
    m_dwHeight = m_d3dpp.BackBufferHeight;
    return OneTimeInit();
} // ZFXD3D::Go
```

*Hochfahren der
Engine*

Die Funktion `ZFXD3D::OneTimeInit` dient genau zu dem Zweck, den ihr Name bereits suggeriert. Unsere DLL ist nun so komplex geworden, dass wir einige grundlegende Dinge auf alle Fälle beim Hochfahren der Engine einstellen müssen, um einen korrekten Ablauf gewährleisten zu können. Dazu gehören unter anderem die Evaluation der Unterstützung von Shadern und

das Einstellen der Viewports und der vitalen Matrizen. Damit der Anwender die Engine auch verwenden kann, ohne alle diese Schritte auszuführen, starten wir durch diese Funktion mit Default-Werten, und zwar wie folgt:

```
HRESULT ZFXD3D::OneTimeInit(void) {
    ZFX3DInitCPU();

    // per Default werden Shader verwendet
    m_bUseShaders = true;

    // Skin- und Vertex-Manager initialisieren
    m_pSkinMan = new ZFXD3DSkinManager(m_pDevice,
                                        m_pLog);

    m_pVertexMan = new ZFXD3DVCManager(
        (ZFXD3DSkinManager*)m_pSkinMan,
        m_pDevice, this, 3000, 4500,
        m_pLog);

    // Renderstates aktivieren
    m_pDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
    m_pDevice->SetRenderState(D3DRS_CULLMODE,
                            D3DCULL_CCW);
    m_pDevice->SetRenderState(D3DRS_ZENABLE,
                            D3DZB_TRUE);

    // Erstelle Standard-Material
    memset(&m_StdMtrl, 0, sizeof(D3DMATERIAL9));
    m_StdMtrl.Ambient.r = 1.0f;
    m_StdMtrl.Ambient.g = 1.0f;
    m_StdMtrl.Ambient.b = 1.0f;
    m_StdMtrl.Ambient.a = 1.0f;

    if (FAILED(m_pDevice->SetMaterial(&m_StdMtrl))) {
        Log("error: set material (OneTimeInit)");
        return ZFX_FAIL;
    }

    // Textur-Filtering aktivieren
    m_pDevice->SetSamplerState(0, D3DSAMP_MAGFILTER,
                            D3DTEXF_LINEAR);
    m_pDevice->SetSamplerState(0, D3DSAMP_MINFILTER,
                            D3DTEXF_LINEAR);
    m_pDevice->SetSamplerState(0, D3DSAMP_MIPFILTER,
                            D3DTEXF_LINEAR);

    // aktiviere persp. Projektion auf 0. Stufe
    ZFXVIEWPORT vpView = { 0, 0, m_dwWidth, m_dwHeight };
    m_Mode = EMD_PERSPECTIVE;
    m_nStage = -1;
}
```



```
SetActiveSkinID(MAX_ID);

// Einheitsmatrix für die Viewmatrix
IDENTITY(m_mView3D);

// Clipping-Plane-Werte
SetClippingPlanes(0.1f, 1000.0f);

// Initialisiere Shader Stuff
PrepareShaderStuff();

// Erzeuge Default-Shader => Seine ID ist 0
if (m_bUseShaders) {
    const char BaseShader[] =
        "vs.1.1          \n" \
        "dcl_position0 v0   \n" \
        "dcl_normal0  v3     \n" \
        "dcl_texcoord0 v6     \n" \
        "dp4 oPos.x, v0, c0   \n" \
        "dp4 oPos.y, v0, c1   \n" \
        "dp4 oPos.z, v0, c2   \n" \
        "dp4 oPos.w, v0, c3   \n" \
        "mov oD0, c4         \n" \
        "mov oT0, v6         \n";

    if (FAILED(CreateVShader((void*)BaseShader,
                            sizeof(BaseShader),
                            false, false, NULL)))
        return ZFX_FAIL;

    if (FAILED(ActivateVShader(0, VID_UU)))
        return ZFX_FAIL;
} // default shader

// Ambientes Licht einstellen
SetAmbientLight(1.0f, 1.0f, 1.0f);

// Perspektivische Projektion Stufe 0 einstellen
if (FAILED(InitStage(0.8f, &vpView, 0)))
    return ZFX_FAIL;

// Perspektivische Projektion Stufe 0 aktivieren
if (FAILED(SetMode(EMD_PERSPECTIVE, 0)))
    return ZFX_FAIL;

return ZFX_OK;
} // OneTimeInit
```

Ich möchte hier nicht zu weit auf die Materie der Vertex-Shader vorgeifen. In dieser Funktion sieht man jedoch sehr schön den ersten Shader, der von unserer Engine per Default erstellt wird, falls die Grafikkarte Vertex-Shader unterstützt. Dieser Shader multipliziert die Vertices nur mit der Combomatrix aus Welt-, View- und Projektionsmatrix, speichert das Ergebnis und schiebt dann lediglich die Texturkoordinaten und den Wert des ambienten Lichts (aus dem konstanten Register c4) als Vertex-Farbe weiter durch die GPU. Dieser Default-Shader stellt also sicher, dass alle Objekte korrekt projiziert und mit grundlegender Beleuchtung und korrekten Texturen dargestellt werden können. Sollte der Anwender also keinen eigenen Shader anlegen, so ist wenigstens die Grundfunktionalität einer 3D-Pipeline hier gegeben. Da wir diesen Shader auch wirklich über die Interface-Methode und nicht durch unschönes Hardcoding anlegen, ist er ein offiziell verfügbarer Shader in unserer Engine, den der Anwender auch von außerhalb über die ID 0 jederzeit wieder aktivieren kann.

Shader

6.11 Demo-Applikation zur Anwendung der DLL

Erfreulicherweise ändert sich am Quelltext der Demo-Applikation aus dem dritten Kapitel recht wenig. Die meisten neu eingeführten Dinge verwendet die DLL, insbesondere durch die Funktion `ZFXD3D::OneTimeInit`, bereits automatisch bzw. initialisiert sie sich selbst korrekt mit Default-Einstellungen. Den Quelltext der Demo-Applikation werde ich hier also nicht noch einmal abdrucken, da sich zu wenig ändert.

Auf die wichtigsten dieser Änderungen möchte ich jedoch näher eingehen.

Multiple 3D-Child-Windows mit multiplen Viewports

Um die Fähigkeiten unserer Engine richtig zu demonstrieren, werden wir die Demo-Applikation so abändern, dass sie nicht nur die DLL hochfährt, sondern dass sie auch gleich die Funktionalität schön präsentiert. Dazu erstellen wir im Hauptfenster unserer Anwendung vier Child-Windows, die als 3D-Fenster an die DLL gemeldet werden, so wie wir es im Kapitel 3 bereits gemacht haben. Dazu erzeugen wir auf zwei Stages unserer Engine entsprechende Viewports:

```
ZFXVIEWPORT rc = { 750, 50, 480, 360 };  
g_pDevice->InitStage(0.8f, NULL, 0);  
g_pDevice->InitStage(0.8f, &rc, 1);
```

Der Viewport der ersten Stage geht also über den gesamten Back-Buffer des jeweiligen Child-Windows. Der Viewport der zweiten Stage ist hingegen in der oberen, linken Ecke angeordnet und um einiges kleiner. Man beachte, dass die Größe des Back-Buffers im Fenster-Modus immer der Größe des

kompletten Desktops entspricht, auch wenn die Anzeige nur im Client-Bereich eines Child-Windows stattfindet und entsprechend automatisch skaliert wird.

Zum Rendern habe ich in die Demo-Applikation noch die Funktion `ProgramTick()` eingebaut, die wie folgt aussieht:



```
HRESULT ProgramTick(void) {
    ZFXMatrix mWorld;
    mWorld.Identity();

    // ersten Viewport aktivieren
    g_pDevice->SetMode(EMD_PERSPECTIVE, 0);
    g_pDevice->SetClearColor(0.7f,0.7f,1.0f);

    // Buffer löschen und Szene starten
    g_pDevice->BeginRendering(true,true,true);

    // RENDER-AUFRUFE

    // zweiten Viewport aktivieren
    g_pDevice->SetMode(EMD_PERSPECTIVE, 1);
    g_pDevice->SetClearColor(1.0f,0.2f,0.2f);
    g_pDevice->Clear(true,true,true);

    // RENDER-AUFRUFE

    g_pDevice->EndRendering();
    return ZFX_OK;
} // Tick
```

*Rendern in
multiple
Viewports*

Zuerst stellt die Funktion die erste Stage ein, die den Viewport über den gesamten Client-Bereich verwendet. Über den Aufruf von `BeginRendering()` löschen wir den Inhalt des Back-Buffers und starten die Szene. Dann folgen Render-Aufrufe, die Geometrie über den ersten Viewport rendern. Dann aktivieren wir den Viewport der zweiten Stage, also den kleineren oben links im Client-Bereich. Dieser enthält jedoch unter Umständen noch Datenmüll von vorher, daher müssen wir ihn auch explizit löschen. Ein Unterbrechen der Szene ist dazu nicht notwendig. Jetzt folgen beliebige Render-Aufrufe, die Geometrie in den zweiten Viewport rendern. Danach beenden wir die Szene.

*Rendern in
multiple Child-
Windows*

Diese Funktion dient also nur dazu zu demonstrieren, wie man in zwei verschiedene Viewports rendern kann. Wenn wir unser Programm im Fullscreen-Modus starten, dann haben wir ja beliebig viele Viewports, aber nur ein Fenster, in das wir rendern können. Starten wir das Programm jedoch im Fenster-Modus, dann haben wir auch beliebig viele Child-Windows, in denen wir jeweils beliebig viele verschiedene Viewports verwenden können.

Jetzt zeige ich noch schnell, wie wir zwischen den einzelnen Child-Windows umschalten können:

```
int WINAPI WinMain(HINSTANCE hInst,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    [...]

    ZFXVector vR(1,0,0), vU(0,1,0), vD(0,0,1), vP(0,0,0);

    // Hauptschleife
    while (!g_bDone) {

        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        // berechne einen Frame
        if (g_bIsActive) {

            g_pDevice->UseWindow(0);
            g_pDevice->SetView3D(vR,vU,vD,vP);
            ProgramTick();

            g_pDevice->UseWindow(1);
            g_pDevice->SetView3D(vU*-1.0f,vR,vD,vP);
            ProgramTick();

            g_pDevice->UseWindow(2);
            g_pDevice->SetView3D(vR*-1.0f,vU*-1,vD,vP);
            ProgramTick();

            g_pDevice->UseWindow(3);
            g_pDevice->SetView3D(vU,vR*-1,vD,vP);
            ProgramTick();
        } // if
    } // while

    [...]
} // WinMain
```



In der Hauptschleife schalten wir nacheinander alle vier Child-Windows durch. Der Übersichtlichkeit halber rendern wir in jedes Child-Window genau dasselbe, nämlich das, was in der Funktion `ProgramTick()` gerendert wird. Natürlich könnte man in jedes Child-Window auch komplett andere Dinge rendern. Damit das aber nicht so langweilig aussieht, habe ich noch eine kleine Verspieltheit für die Kamera-Matrix eingebaut. In jedem Fenster

*Kamera mit Salto
Mortale*

ist die Ansicht der Kamera um jeweils 90 Grad im Uhrzeigersinn gekippt; dazu vertauschen wir einfach die Basisvektoren der Kamera entsprechend. Dadurch wird die Kamera sozusagen jeweils um 90 Grad weitergerollt, und das Bild im Child-Window erscheint entsprechend um 90 Grad in die andere Richtung gekippt.

Einfacher Geometrie-Loader

Damit haben wir durch ein paar wenige Änderungen an unserer bisherigen Demo-Applikation die volle Kraft unserer DLL demonstriert. Das allein wäre jedoch ziemlich langweilig, so lange wir nicht wirklich etwas rendern. Daher habe ich einen sehr, sehr einfachen Geometrie-Loader und Renderer in der Demo-Applikation implementiert, die in der folgenden Klasse stecken:



```
class ZFXModell {
protected:
    ZFXRenderDevice *m_pDevice;

    UINT    m_nNumSkins;
    UINT    *m_pSkins;
    UINT    m_nNumVertices;
    VERTEX  *m_pVertices;
    UINT    m_nNumIndices;
    WORD    *m_pIndices;
    UINT    *m_pCount;    // Indices pro Material
    UINT    *m_pBufferID; // statische Buffer
    FILE    *m_pFile;
    FILE    *m_pLog;
    bool    m_bReady;

    void ReadFile(void);

public:
    ZFXModell(const char *chFile,
               ZFXRenderDevice *pDevice,
               FILE *pLog);
    ~ZFXModell(void);

    HRESULT Render(bool bStatic);
};
```

**Quick'n'Dirty
forever**

Außer einem Konstruktor zur Initialisierung und einer Render-Funktion zum (*wer hätte es geahnt!*) Rendern beinhaltet die Klasse keinerlei Funktionalität. Sie liest einfach die Daten in einem bestimmten, einfachen Format ein und erstellt daraus eine Vertex- und Indexliste. Die Klasse ist das Resultat weniger Stunden Arbeit und daher mindestens ebenso hässlich implementiert, wie sie unfunktionell ist. Wir werden sie auch nicht weiter verwenden, und man sollte sie sich am besten gar nicht erst anschauen, da

sie wirklich nur zum Testen der DLL erstellt wurde und auch nicht garantiert bugfrei ist. Allerdings hat die Klasse ein nettes Feature. Sie legt sowohl statische Buffer in der DLL an als auch eigene Vertex- und Indexlisten, über die das dynamische Rendern mit der DLL getestet werden kann. Daher rührt der `bool`-Parameter der Render-Funktion.

Wer sich dennoch für die Klasse interessiert, der möge sich den Quelltext der Demo-Applikation auf der CD-ROM anschauen, ebenso wie die entsprechenden Dateien mit Beispielmolellen, die dort zu finden sind.

6.12 Ein Blick zurück, zwei Schritt nach vorn

Ich hatte das Gefühl, dieses Kapitel dauerte nahezu eine Ewigkeit. Ein wenig demotivierend kommt hier vielleicht hinzu, dass man auf den ersten Blick kaum etwas erreicht hat. Noch immer haben wir keine fluffigen 3D-Objekte auf dem Bildschirm. Nichts bewegt sich, nichts feuert mit Lasern wild um sich, und keine Effekte ziehen das Auge des Betrachters in seinen Bann. Trotzdem haben wir über 90 Seiten in diesem Buch damit zugebracht, am Design der DLL herumzufeilen. Und genau darum ging es auch hier: das Design einer Render-DLL zur Kapselung einer Grafik-API. Daneben haben wir außerdem noch eine Menge Wissen im Bereich der Performance-Optimierung aufgebaut. Ab jetzt sollten wir uns bei jedem Polygon, das wir an die DLL senden, fragen, ob wir es wirklich auf die optimalste Art und Weise zum Rendern geschickt haben. Genug Grundwissen, um diese Frage zu beantworten oder entsprechende Vergleichstests durchzuführen, haben wir nun auf alle Fälle.

Lernerfolg

Dieses Kapitel ist auch mehr oder weniger das letzte gewesen, das sich aktiv mit einer Grafik-API (hier Direct3D) auseinander zu setzen hatte. Wir haben nun alles das, was wir zum Rendern und zum Setzen der Render-States brauchen, in der DLL gekapselt; und zwar nicht irgendwie, sondern recht komfortabel. An dem kaum gewachsenen Umfang der Demo-Applikation kann man ersehen, wie einfach unsere DLL nach außen hin anzuwenden ist. Durch wenige Funktionsaufrufe ist die DLL initialisiert, betriebsbereit und flexibel umschaltbar. Auch wenn wir nicht wirklich bezaubernde Dinge auf den Bildschirm rendern, so haben wir doch nun ein mächtiges Tool, das uns bei der beschwerlichen Reise durch den Rest dieses Buches gute Dienste leisten wird.

Direct3D, ade!

Wir haben in den folgenden Kapiteln zwar noch zwei kleine Intermezzos auf unserem Programm, namentlich DirectSound und DirectInput, doch diese beiden recht knapp gehaltenen Einführungen sind dann wirklich die letzten Kapitel, die nicht aktiv zur Bereicherung unseres Repertoires an (grafischen) Effekten beitragen werden. Der Rest des Buches widmet sich nun ausführlich dem, was wir eigentlich tun wollen: der Programmierung von 3D-Computerspielen.

Effekte backbord voraus

Standby to initiate release sequencers on express elevator to hell ...

7 3D-Pipeline und Shader

*»Ungerechtigkeit wird durch Gleichgültigkeit unterstützt.«
(H. M. Murdock, Golfball-Befreiungsarmee)*

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Funktionsweise der 3D-Pipeline vom Programm zum Pixel
- ➔ CPU- versus GPU-lastiger Code
- ➔ Prinzip und Aufbau von Vertex- und Pixel-Shadern
- ➔ diverse Shader-Beispiel-Applikationen
- ➔ Point Light Attenuation per Pixel
- ➔ Tangent-Space und Bump-Mapping

7.1 Grundlagen von Shadern

In diesem Kapitel möchte ich mit euch darüber plaudern, was Vertex-Shader und Pixel-Shader eigentlich genau sind. Dabei ist der Name Shader an sich ein wenig irreführend, denn ein Shader hat zunächst einmal gar nichts mit Shading (engl. Schattierung) zu tun. Vereinfacht gesagt, dienen Vertex-Shader dazu, die Daten eines Vertex zu manipulieren, den ein Programm auf die Fahrt durch die 3D-Pipeline schickt. Ein Pixel-Shader dient ganz analog dazu, einen Pixel zu manipulieren, den die Grafikkarte dann aus den Vertex-Daten für ein Dreieck generiert hat. Dort kommt dann also doch ein bisschen Shading ins Spiel.

*Shader shaden
nichts*

In der OpenGL-API hat man für die Shader daher auch eine Bezeichnung gewählt, die ein wenig treffender ist. Dort nennt man sie Vertex-Programms und Pixel-Programms.



Mini-Programme

Vertex-Shader und Pixel-Shader sind an sich schnell erklärt. Bei Shadern handelt es sich um kleine Programme in Form von speziellen Assembler-Instruktionen, die noch kompiliert werden müssen, oder die Shader liegen bereits in kompilierter Form vor. Beispielsweise bietet das DirectX SDK einen Shader-Compiler an, mit dem man seine Shader kompilieren kann,

um sie dann als kompilierte Datei in sein Programm zu laden. Im letzten Kapitel haben wir aber schon eine Methode unserer Render-Devices geschrieben, um den Compiler der D3DX-Bibliothek zu benutzen.

*Steuerung der
GPU*

Doch was macht ein Shader nun genau? Wir müssen uns das etwa wie mit einem normalen Assembler-Programm vorstellen. Dort schreiben wir eine Reihe von Instruktionen, die die CPU veranlassen sollen, etwas Bestimmtes zu tun. Eine spezielle Form davon haben wir auch schon mit dem SSE-Assembler gesehen und selbst verwendet. Bei den Shadern ist es nun so, dass die Instruktionen dieser speziellen Programme nicht von der CPU ausgeführt werden, sondern von der GPU, also dem Prozessor auf der Grafikkarte. Etwas legerer drückt man das auch so aus, dass man mit Assembler die CPU programmieren und analog mit Shadern die GPU programmieren kann.

Was nicht geht

Das ist natürlich nur zum Teil richtig, denn die aktuelle Generation von Shadern erlaubt einem nur einen Teil der Grafikkarte wirklich vollkommen selbst zu steuern. Es gibt auch weiterhin Operationen, die eine Grafikkarte vollkommen autonom durchführt und die wir nicht beeinflussen, sondern nur durch API-Funktionen nutzen können. Ein Beispiel ist hierbei das Alpha-Blending. Dieses muss man, auch wenn man Shader verwendet, wie bisher über Renderstates von Direct3D oder entsprechende Anweisungen und Einstellungen in OpenGL steuern.

3D-Pipeline

Power-Vertex

Schauen wir uns nun mal an, was wirklich alles passiert, wenn wir unsere Geometrie in Form von Vertices und Indexlisten zum Rendern an die API übergeben. Abbildung 7.1 zeigt eine Darstellung dessen, was man als 3D-Pipeline bezeichnen kann. Als Input haben wir am oberen Ende der Pipeline die Geometrie. Diese wird dann entweder an die so genannte *Transformation and Lighting-Engine* (kurz TnL) oder an einen Vertex-Shader weitergeleitet, falls einer aktiv ist. Die TnL-Engine ist die eingebaute Funktionalität der Grafikkarte, Vertexdaten zu transformieren und zweidimensional zu projizieren und dabei die Vertices zu beleuchten. Dieses Feature einer Grafikkarte war vor einigen Jahren ein sehr großer Sprung für die Performance von 3D-Beschleunigern, denn zuvor musste diese Aufgabe in der Software erledigt werden. Das war natürlich entsprechend langsam, zumal es auch noch keine so fortschrittliche Technologie bei den CPUs gab wie beispielsweise SSE. Natürlich übernehmen APIs wie Direct3D und OpenGL diesen Job auch, falls eine Grafikkarte wirklich kein TnL in der Hardware unterstützen sollte. Die TnL-Engine bildet einen Teil der so genannten *Fixed-Function-Pipeline*. Sie heißt deshalb so, weil der Programmierer das Verhalten dieser Pipeline nur insofern steuern kann, als dass er bestimmte Input-Parameter liefert. Die Vertex-Shader hingegen bilden einen Teil der so genannten *Flexible-Pipeline*, weil der Programmierer hier die Transformation und Beleuchtung vollkommen frei selbst programmieren kann.

Nun gelangen die Daten als Nächstes zur Viewport-Transformation und zum Clipping. Dort werden sie auf den Bereich 0.0f bis 1.0f umgerechnet, und dann werden die Pixel geclippt, die außerhalb des Viewports liegen. Danach wird der Pixel wiederum entweder in den zweiten Teil der Fixed-Function-Pipeline gesteckt, der durch die Multitexturing-Unit auf der Grafikkarte gebildet wird. Oder aber die Daten gehen in einen Pixel-Shader, falls einer aktiv ist. Dort wird die Farbe des Pixels berechnet, die er vorerst erhält. Doch der arme Pixel hat seinen Spießrutenlauf immer noch nicht ganz hinter sich. Nun muss er noch durch den Nebel des Grauens (die Fog-Berechnungen), und falls er vom Betrachter aus gesehen so weit weg ist, dass er im eventuell aktivierten Nebel liegt, so wird er entsprechend mit der Nebelfarbe geblendet. Nun folgt das Alphablending und der Test gegen den Stencil-Buffer, sofern dieser aktiv ist. Natürlich darf auch der Test gegen den Depth-Buffer nicht fehlen, falls dieser aktiv ist, damit es nicht zu fehlerhaften Überdeckungen von Pixeln kommt. Hat der Pixel dies nun alles über sich ergehen lassen und er wurde immer noch nicht aus der Pipeline geschmissen, dann darf er endlich in den Frame-Buffer. Dies wird in der Regel der Back-Buffer sein. Es kann sich dabei aber auch um eine Textur handeln, die als Render-Target verwendet wird.

Power-Pixel

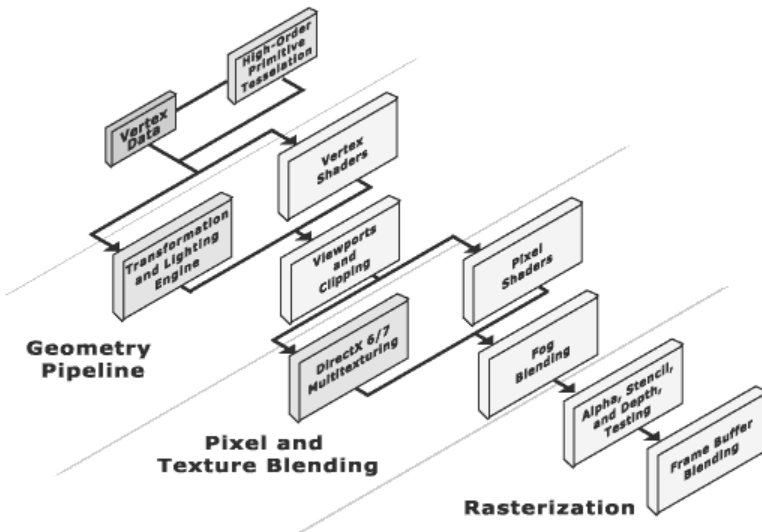


Abbildung 7.1:
Die 3D-Pipeline von Direct3D (Die Abbildung ist Eigentum von Microsoft.)

Die Fixed-Function-Pipeline besteht also aus den zwei Elementen TnL-Engine und Multitexturing-Unit. Die Flexible-Pipeline besteht aus den beiden jeweiligen Gegenparts, den Vertex-Shadern und den Pixel-Shadern. Man kann aus der Grafik auch gut ersehen, dass die beiden Shader-Typen ihren jeweiligen Gegenpart vollkommen ersetzen. Wenn man einen Vertex-Shader verwenden möchte, dann muss man auch sämtliche Aufgaben übernehmen, die sonst die TnL-Engine ausgeführt hätte. Man muss den Vertex transformieren, die Beleuchtung des Vertex berechnen, falls man eine Per-

Fixiert oder flexibel?

Vertex-Beleuchtung verwendet, usw. Gleiches gilt für die Pixel-Shader. Wenn man diese anstelle der Multitexturing-Unit verwendet, muss man auch sämtliche Blending-Operationen multipler Texturen in seinem Pixel-Shader implementieren.

*Immer nur einer,
bitte*

Aus der Grafik ergibt sich auch, dass man immer nur einen Vertex-Shader und einen Pixel-Shader aktiv haben kann. Vor dem Aufruf einer Render-Funktion setzt man maximal einen Vertex-Shader und maximal einen Pixel-Shader für die Grafikkarte. Es gibt also keine Interaktionsmöglichkeit zwischen verschiedenen Vertex-Shadern und auch nicht zwischen verschiedenen Pixel-Shadern. Und es ist auch nicht direkt möglich, Werte aus einem Shader zurück an die Applikation zu geben. Die 3D-Pipeline ist eine Einbahnstraße für die Vertex-Daten, die entweder auf dem Frame-Buffer oder im Nirgendwo endet. Es gibt an keiner Stelle ein Zurück in den Schutz der Applikation.

*Wenn, dann
richtig!*

Des Weiteren sei darauf verwiesen, dass man idealerweise immer nur einen Pixel-Shader verwendet, wenn man auch einen eigenen Vertex-Shader benutzt und umgekehrt. Einige Grafikkarten haben Probleme damit, wenn man nur einen der beiden möglichen Shader verwendet und für den anderen Teil weiterhin die Fixed-Function-Pipeline nutzen möchte. Es ist auch so, dass moderne Grafikkarten eigentlich gar keine Fixed-Function-Pipeline mehr haben. Diese wird dort intern über eigene Shader emuliert, auch wenn das für den Anwender nicht sichtbar ist.

CPU-lastig versus GPU-lastig

Are you limited?

Bevor wir uns nun gleich in die Shader-Geschichte stürzen, möchte ich noch eine Sache klären. Wenn man eine sehr geschmeidige Applikation programmiert hat und diese dann unerwartet langsam läuft, dann kann das sehr viele verschiedene Ursachen haben, die man aber in zwei Kategorien aufteilt: CPU-limited und GPU-limited. Damit ist gemeint, ob die Anwendung deswegen so langsam ist, weil die CPU im weitesten Sinne überlastet ist (dabei bezieht man i.d.R. auch Bus-Traffic usw. mit ein) oder weil die Grafikkarte viel zu viel zu tun hat. Man kann eine Grafik-Applikation natürlich auch dann wunderbar ausbremsen, wenn man meint, man hätte eine GeForce FX oder Radeon 9800 Pro und lässt sie einfach alles brute-force ohne Frustum-Culling usw. rendern. Auf Sseiten der Grafikkarte gibt es dann auch weitere Unterscheidungen, wo es nun genau hakt. Der Bottleneck kann das TnL bzw. ein zu schwerer Vertex-Shader bei zu vielen Daten sein. Oder aber der Bottleneck ist die Fillrate, also das Setzen einzelner Pixel, von denen man zu viele aufgrund von zu viel Overdraw und einer zu hohen Auflösung hat und auf die man dann womöglich noch viele Multitexturing-Operationen oder einen zu schweren Shader hetzt.

Mit dem Auftauchen der Shader auf der Bühne der 3D-Grafik versuchte man natürlich, so viel wie möglich in Shader zu packen, damit die schnelle Grafikkarte endlich alles machen konnte was mit Grafik im weitesten Sinne zu tun hatte. Einige Beispiele sind Character-Animation, die wir uns im nächsten Kapitel ansehen werden, und Shadow-Volumes, die später im Buch behandelt werden. Diese beiden Dinge, bei denen man in jedem Frame viele Vertex-Daten manipulieren muss, kann man natürlich über Shader umsetzen. Und sie sind auch sehr schnell mit entsprechenden Shadern. Doch wenn man das Ganze im Kontext eines umfassenden Spiels sieht, dann hat die Grafikkarte auf einmal nicht nur die Character-Transformation am Hals, sondern auch das Rendern und Beleuchten ganzer Level. Man tendiert also schnell dazu, alles zur Grafikkarte zu schieben und nichts auf der CPU zu machen. Zudem kommt in echten Anwendungen noch hinzu, dass man beispielsweise die transformierten Daten eines Characters für die Kollisionsabfrage benötigt und dann, zumindest für potenziell sehr wahrscheinlich kollidierende Characters, die ganze Berechnung nochmals auf der CPU durchführen muss.

*Shader forcieren
GPU-limited Code*

Bei den Shadow-Volumes kommt noch ein anderes Problem hinzu. Vertex-Shader können keine Geometrie erzeugen. Sie kennen immer nur genau einen Vertex, den sie transformieren und dann in der Pipeline weiterleiten. Sie können aber keine Geometrie erzeugen, da sie ja noch nicht einmal eine Ahnung vom Kontext eines Triangles haben, zu dem »ihr« aktueller Vertex gehört. Es sei nur so viel gesagt, dass man für ein Shadow-Volume zu einem existierenden Modell zusätzliche Polygone in Abhängigkeit zum Lichteinfall berechnen muss. Da dies in einem Shader nicht möglich ist, muss man dort alle potenziell irgendwann gebrauchten Polygone schon vorher in das Modell zusätzlich einfügen und mit auf die Grafikkarte bringen. Man braucht dabei für jede Kante des Modells, an der sich zwei Polygone berühren, ein zusätzliches Rechteck, also zwei Dreiecke. Und das ist je nach Anzahl und Detailgrad der Modelle nicht gerade wenig.

*Shadow-Volumes
mit Shadern*

So etwas wie eine beste Strategie, was man nun über Shader auf der GPU löst oder was man die CPU berechnen lässt, gibt es natürlich nicht. Das ist immer abhängig von der Zielpattform und dem Programm an sich. Im Zweifelsfall ist es aber immer gut, wenn man beide Möglichkeiten implementiert und auf verschiedenen Systemen vergleicht. Und da wir gerade bei Performance sind, schauen wir noch mal etwas genauer hin.

Beste Strategie

Eine der teuersten Operationen auf Grafikkarten ist es, den aktuellen Shader zu wechseln. Wo es möglich ist, sollte man auf alle Fälle seine Geometrie sortiert nach Shadern rendern, um so die Switches zwischen verschiedenen Shadern zu minimieren. Dasselbe gilt für die Fixed-Function-Pipeline, die ja auf modernen Karten durch sehr schwere Shader emuliert wird. Anstatt von einem Shader auf die Fixed-Function-Pipeline zu wechseln, ist es immer besser, einen eigenen Shader zu schreiben, der genau das

*Switches
vermeiden*

tut, was man von der Fixed-Function-Pipeline gerade möchte. Ein Wechsel von einem anderen Shader zu einem solchen eigenen Shader ist i.d.R. billiger, also schneller als der Wechsel auf die Fixed-Function-Pipeline.

Welchem Bottleneck man nun genau unterliegt wird man normalerweise nicht so ganz einfach bestimmen können, da oftmals verschiedene Dinge gleichzeitig die Performance herabsetzen. Wenn man beispielsweise ein Problem mit der Fillrate hat, dann wird man einen Performance-Gewinn feststellen, wenn man die Bildschirmauflösung kleiner stellt oder einen einfacheren Pixel-Shader einsetzt. Wenn die CPU der Bottleneck ist, dann wird man keine Performanceänderung sehen, wenn man einfach weniger Geometrie rendern lässt oder auf Texturen verzichtet usw. Es gibt also einige Möglichkeiten, um sich an die Problemzonen heranzutasten. Zu wissen, wo die Performance-Killer in einer Applikation stecken, ist meistens gleichbedeutend damit, dass man einem Verbesserungsansatz schon sehr nah ist. Durch Shader lassen sich etliche Dinge von der CPU auf die GPU verlagern.

Vertex-Manipulation über Vertex-Shader

Vertex-ALU

Der Kern des Vertex-Shader-Geschäfts ist die so genannte Vertex-ALU, die Arithmetic Logical Unit. In der ALU finden alle Berechnungen statt, die wir mit dem Vertex-Shader umgesetzt haben möchten; sie ist also das Herz und die Seele des Vertex-Processings auf der Grafikkarte. Damit die ALU aber irgendetwas berechnen kann, braucht sie auch Input von Daten und natürlich die Möglichkeit, ihre Ergebnisse zu publizieren. Der Input für die ALU kommen auf der Grafikkarte aus zwei verschiedenen Richtungen. Zum einen gibt es 16 so genannte Inputvektoren namens `v0` bis `v15`. Verwendet man die Fixed-Function-Pipeline, so legt man in Direct3D über das Flexible-VertexFormat (FVF) fest, welche Vertex-Komponenten (Position, Normalen-Vektor, Texturkoordinaten usw.) im Vertex-Format vorhanden sind, und Direct3D weist diese einzelnen Komponenten dann entsprechenden Input-Registern zu. Diese Arbeit erledigen wir bei der Implementierung eines Vertex-Shaders nun selbst und können dabei alle 16 Register frei verwenden. Alle Register der GPU sind im Folgenden immer 128 Bit groß, also nicht ganz zufällig groß genug für vier `float`-Werte mit je 32 Bit – oder anders ausgedrückt: für einen vierdimensionalen Vektor. Das ist auch einer der Gründe, warum die GPU viele Aufgaben im Bereich der Computergrafik und damit der Vektormathematik schneller erledigen kann als die CPU. Die GPU ist auf Vektorrechnung hin optimiert und kann viele Berechnungen in einem Zyklus erledigen, für die die CPU mehrere Zyklen benötigen würde. Daher ist die GPU trotz langsamerer Taktfrequenz in diesem Bereich viel schneller als die CPU.

Konstanten-Register

Neben den Input-Registern gibt es die so genannten Konstanten-Register `c0`, `c1`, ..., `cn`. Sie heißen so, weil sie lediglich Konstanten speichern können, die der Shader abfragt. Aber ebenso wie bei den Inputregistern ist es nicht mög-

lich, dass der Shader Daten in die Konstanten-Register hineinschreibt. Diese dienen der Applikation, also dem Programm, das einen Shader einsetzt, dazu, bestimmte Werte an den Shader zu übermitteln wie beispielsweise eine Transformationsmatrix, die Vektor des einfallenden Lichts oder die Hausnummer der Oma von gegenüber. Wie viele Konstanten-Register eine Grafikkarte anbietet, ist unterschiedlich und kann zur Laufzeit ebenfalls über die D3DCAPS9-Struktur und deren Feld `MaxVertexShaderConst` erfragt werden. Aber man kann davon ausgehen, mindestens 96 Stück auf jeder Grafikkarte zu finden, die auch Shader unterstützt. Das klingt zunächst recht viel, aber man sollte bedenken, dass eine Matrix bereits vier Konstanten-Register belegt. Wenn man nun eine Character-Animation über Shader umsetzen möchte, benötigt man vergleichsweise viele Matrizen für all die Bones des Modells. Aber in den meisten anderen Fällen reichen diese Register vollkommen aus.

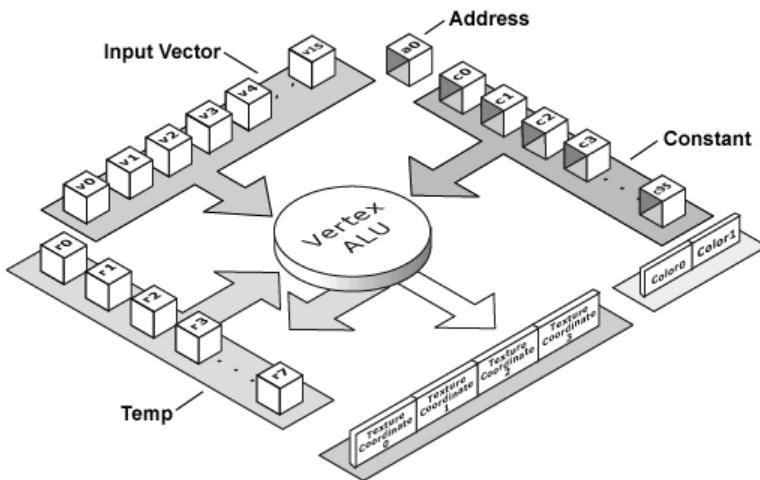


Abbildung 7.2:
Aufbau der Vertex-ALU mit ihren Registern (Die Abbildung ist Eigentum von Microsoft.)

Wenn ein Mensch eine komplexere Berechnung durchführt, braucht er meistens einen Zettel, um sich wenigstens ein paar Zwischenergebnisse zu notieren. Ebenso braucht man in einem Vertex-Shader ein paar Register, in denen man ein paar Werte zwischenspeichern kann. Dazu dienen die temporären Register in einem Shader, von denen der ALU zwölf Stück namens `r0`, ..., `r11` zur Verfügung stehen. In diese Register kann der Shader jeweils 128 Bit an Daten schreiben und auch wieder aus ihnen lesen. Wurde ein Vertex abgearbeitet, so geht der Inhalt der Register natürlich wieder verloren. Daher bezeichnet man sie auch als *temporäre* Register.

Temporäre Register

Wenn wir mit der Berechnung fertig sind so nehmen wir unser Lineal zur Hand und unterstreichen das Ergebnis zweimal fein säuberlich. Die ALU hingegen schiebt die Ergebnisse in die so genannten Output-Register, von wo aus sie weiter den Weg der 3D-Pipeline gehen, wie oben beschrieben

Output-Register

wurde. Als Output-Register stehen `oD0` für den diffusen Farbwert und `oD1` für den spekulären Farbwert zur Verfügung. Dies gilt natürlich nur, wenn man die Fixed-Function-Pipeline anstelle eines Pixel-Shaders verwendet, was man ja nicht tun sollte. Ansonsten kann man die Werte im Pixel-Shader ja als Input übernehmen und interpretieren, wie man möchte. Das Output-Register `oPos` dient dazu, die transformierte und projizierte Position des Vertex aufzunehmen. Außerdem gibt es noch die Register für die Texturkoordinaten `oT0`, ..., `oT7`, wobei je nach Grafikkarte auch weniger verfügbar sein können. Verbleibt noch das `oFog`-Register für einen Nebelwert an dem Vertex und zu guter Letzt das Register `oPts` für die Größe eines Point-Sprites, wobei die beiden letztgenannten jeweils nur einen `float`-Wert in den ersten 32 Bit aufnehmen können.

Kurz gesagt ...

Fassen wir also noch mal einige Dinge explizit zusammen: Ein Vertex-Shader ist ein kleines Assembler-Programm, das man der Grafikkarte schickt, damit diese das Programm dann für jeden Vertex ausführt, den wir zum Rendern schicken. Wenn man im Vertex-Shader einen Wert berechnen muss, der für alle Vertices eines Modells gleich ist, dann sollte man das entsprechend in der Applikation tun und das Ergebnis in einem Konstanten-Register speichern, das dem Shader zur Verfügung steht. Der Shader ersetzt die `TnL`-Funktionen der Hardware; das bedeutet, wir müssen den Vertex in einen eigenen Shader wenigstens transformieren. Je länger ein Shader ist, desto länger dauern die Berechnungen pro Vertex. Der Vertex-Shader erzeugt bestimmte Output-Werte, die dann weiter durch die 3D-Pipeline gehen, und letzten Endes werden aus drei Vertices einzelne Pixel für die Anzeige auf dem Bildschirm berechnet – natürlich nur, falls man keinen Wireframe- oder Linienmodus aktiviert hat.

Pixel-Manipulation über Pixel-Shader

Pixel-ALU

Da wo es eine Vertex-ALU gibt, ist auch eine Arithmetic Logical Unit für Pixel nicht fern, die in Abbildung 7.3 zu sehen ist. Sie ist von der Architektur her ganz genauso aufgebaut wie die Vertex-ALU. Es gibt Input-Register, temporäre Register, Konstanten-Register und nicht zuletzt auch ein Output-Register. Mit diesen Registern arbeitet die Pixel-ALU, um genau ein Ziel zu erreichen. Ihre Aufgabe ist es nämlich, die Farbe eines Pixels zu berechnen und diesen dann wieder in die 3D-Pipeline zu entlassen, damit er den Weg eines jeden irdischen Pixels weitergehen kann. Dabei sollte man noch eine Kleinigkeit klären: Im Gegensatz zu der vielleicht aufkommenden Vermutung, dass Pixel-Shader mit dem Frame-Buffer arbeiten, ist das nicht so. Ein Pixel-Shader wird auf jeden Pixel angewendet, der für eine zu rendernde Primitive generiert wird. Ob dieser dann hinterher durch den Depth-Buffer oder das Alpha-Blending vom Frame-Buffer abgeschmettert wird, ist hierbei ganz egal.

Der Pixel-ALU stehen zwei Input-Register für Farbwerte zur Verfügung, nämlich `v0` und `v1`. Diese beiden entsprechen direkt den Output-Registern `oD0` und `oD1` des Vertex-Shaders.

Input-Register

Die Textur-Register `t0`, ..., `tn` eines Pixel-Shaders sind auch eine Art von Input-Registern. Sie beinhalten eigentlich die Texturkoordinaten aus dem Vertex-Shader der entsprechenden Stage. In der Regel berechnet man daraus aber direkt den interpolierten Farbwert aus der zugehörigen Textur und arbeitet nicht mit den Texturkoordinaten im Pixel-Shader. Diese Register sind aber auch eine Möglichkeit, andere Daten aus dem Vertex-Shader an den Pixel-Shader weiterzugeben.

Textur-Register

Ebenso wie der Vertex-ALU stehen auch der Pixel-ALU verschiedene Konstanten-Register zur Verfügung. Im Gegensatz zu den Registern der Vertex-ALU nehmen diese aber nur jeweils vier `float`-Werte im Bereich von `-1` bis `+1` auf.

Konstanten-Register

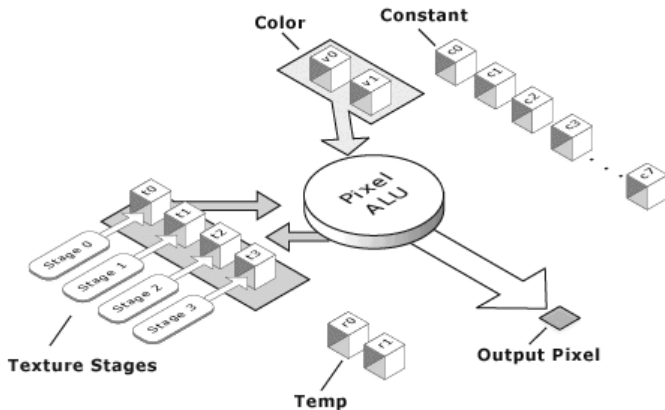


Abbildung 7.3:
Aufbau der Pixel-ALU mit ihren Registern (Die Abbildung ist Eigentum von Microsoft.)

Nun wissen wir über den prinzipiellen Aufbau von Vertex-Shadern und Pixel-Shadern Bescheid. Beide erlauben uns wesentlich mehr Flexibilität, als wenn wir die Fixed-Function-Pipeline verwenden würden. Aber alle Theorie nützt uns natürlich nichts, wenn wir nicht auch ein ordentliches Beispiel dazu haben. In den folgenden Abschnitten dieses Kapitels werden wir einige Shader-Paare implementieren, um von den grundlegenden Transformationen über Multitexturing bis hin zum Per-Pixel-Lighting einiges zu sehen, was man mit Shadern alles machen kann.

Kurz gesagt

7.2 Shader-Techniken und Beispiele

Nun haben wir viel über Shader gehört und darüber, wie viel Flexibilität man durch sie erreichen kann. Daher ist es an der Zeit, sie endlich in Aktion zu sehen. Im folgenden Abschnitt werden wir einige Shader implementieren

Ran an die Shader

und in einer kleinen Demo-Applikation zu Laufen bringen. Diese Shader demonstrieren kurz und knackig, wie man bestimmte Effekte über eine Shader-Combo aus einem Vertex- und einem Pixel-Shader umsetzen kann. Die Shader an sich sind zwar recht kurz gehalten, dienen aber durchaus dazu, euch Wege aufzuzeigen, die man gehen kann. Über Shader gibt es mittlerweile auch etliche Bücher, auf die ich an dieser Stelle verweisen möchte. Der Schwerpunkt dieses Buches liegt ganz klar auf dem Design der Engine und einem Indoor-Spielchen. Nichtsdestotrotz benötigen wir später auch den einen oder anderen Shader für unser Spiel, und daher zeige ich hier einige kleine Shader, mit denen ihr experimentieren und die ihr dann auch zusammenführen könnt.

Demo 1: Basistransformationen

*Wenn Shader,
dann richtig
shaden*

Als Erstes beginnen wir mit den Basistransformationen, um Geometrie korrekt perspektivisch projiziert und mit einer Textur überzogen darstellen zu können. Den Code dazu haben wir eigentlich schon gesehen, nämlich im letzten Kapitel. Unsere Engine enthält ja implizit schon einen Basis-Vertex- und Pixel-Shader, wie wir im letzten Kapitel gesehen haben. Wie genau diese funktionieren, das schauen wir uns gleich hier an. Zuerst sei aber noch vorweg geschickt, dass viele Tutorials im Internet und auch Bücher immer strikt zwischen den Shadern trennen und erst mit Vertex-Shadern beginnen, um dann später Pixel-Shader hinzuzunehmen. Das machen wir hier nicht so, sondern wir verwenden immer eine Combo aus einem Vertex- und einem Pixel-Shader. Das hat auch seinen Grund, denn einige Grafikkarten mögen es gar nicht, wenn man die Fixed-Function-Pipeline mit der flexiblen vermischt, und können dort teilweise Probleme bereiten. Wenn wir also einen Vertex-Shader haben möchten, dann implementieren wir bitteschön auch gleich einen Pixel-Shader dazu.

Ein einfacher Vertex-Shader

*Ab ins kalte
Wasser – Vertex-
Shader*

Ohne lange um den heißen Brei herumzureden schauen wir uns gleich den Vertex-Shader an. Wer ein bisschen Assembler und ein bisschen 3D-Grafik kann, der wird auf Anhieb sehen, was hier geschieht. Und alle anderen entnehmen das der folgenden Beschreibung des Programms.



```
vs.1.1
decl_position v0
decl_normal v3
decl_texcoord v6
decl_tangent v8
m4x4 oPos, v0, c0
mov oD0, c4
mov oT0, v6
```

In einem Shader, egal ob Vertex- oder Pixel-Shader, steht zuerst immer die Version. Wir arbeiten hier, wie man sehen kann, mit der Version 1.1, da wir noch sehr weit abwärtskompatibel bleiben wollen. Schließlich sind die neueren Shader-Versionen zwar mächtiger, aber auch nur auf entsprechenden Grafikkarten verfügbar. Dann folgen ein paar Deklarationen mit dem Präfix `decl_`, die angeben, in welche Input-Register die einzelnen Daten aus dem Vertex-Stream gemappt werden sollen. Wir haben hier vier Datenfelder eines Vertex, die wir den verschiedenen Input-Registern zuweisen. Zuerst natürlich die Position des Vertex, dann seinen Normalenvektor und seine Texturkoordinaten.

Zuerst die Version, dann alles andere

Als Letztes folgt der so genannte Tangent-Vektor. Der sagt uns jetzt noch nichts, da wir ihn erst für das Bump-Mapping brauchen, das wir weiter unten behandeln. Trotzdem habe ich ihn hier schon mal in den Shader mit aufgenommen, damit wir in unseren Beispielen durchgängig dasselbe Vertex-Format verwenden können. Unserer Engine habe ich unter anderem auch die folgende Vertex-Struktur spendiert:

Tangente und neue Vertex-Struktur

```
typedef struct TVERTEX_TYPE {
    float  x, y, z;
    float  vcN[3];
    float  tu, tv;
    float  vcU[3];
} TVERTEX;
```



Diese neue Struktur hat nichts mit den Shadern an sich zu tun, sondern damit, dass wir für das Bump-Mapping eben noch den Tangent-Vektor brauchen. Der obige Tangent-Vertex hat neben der Position, der Normalen und den Texturkoordinaten noch einen weiteren Vektor, der dort `U` genannt wird. Und genau dieser `U`-Vektor wird auch Tangent-Vektor genannt. Aber dazu später beim Bump-Mapping mehr.

Kommen wir zu unserem Shader zurück. Nun wissen wir, in welchem Input-Register was für Daten zu finden sind. Zudem erinnern wir uns, dass wir im letzten Kapitel immer die aktuelle Combo-Matrix aus Transformations-, View- und Projektionsmatrix implizit in der Engine in das Konstanten-Register `c0` für den Vertex-Shader geschoben haben – oder besser gesagt: die Transponierte dieser Matrix. Im Konstanten-Register `c4` parken wir analog über die Engine automatisch das ambiente Licht, das der Anwender für die Engine einstellt. Nun zur ersten Zeile:

Shader-Code

```
m4x4 oPos, v0, c0
```

Bei `m4x4` handelt es sich nicht um eine Instruktion, sondern um ein Makro, das eine Multiplikation eines Vektors im $\mathbb{R}^4$ mit einer 4×4 -Matrix durchführen soll. Die Konstanten-Register `cX` sind ja jeweils 128 Bit groß, können also vier `float`-Werte aufnehmen. Damit enthalten die Register `c0`, `c1`, `c2` und

$c3$ je eine Zeile der (transponierten) Matrix. Ohne das Makro zu verwenden, könnte man daher diese Anwendung der Transformation und Projektion auf den Vertex wie folgt umsetzen:

```
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3
```

*Warum
transponiert?*

Bei `dp4` handelt es sich nun endlich um eine Instruktion, und zwar um das Punktprodukt (engl. dot product) zwischen zwei Vektoren im $\mathbb{R}^4$. Man multipliziert hier also je eine Zeile der Matrix mit dem Positionsvektor des Vertex und erhält so die vier einzelnen Komponenten der transformierten und projizierten Vertex-Position. An dieser Stelle sollten wir stutzig werden, denn wenn wir einen Vektor mit einer Matrix multiplizieren, so tun wir das ja, indem wir das Punktprodukt des Vektors mit jeweils einer *Spalte* und nicht mit einer Zeile der Matrix bilden. Das ist aber in einem Shader nicht so ohne weiteres möglich, jedenfalls nicht in einer Instruktion. Daher muss man alle Matrizen, die man in einem Shader verwenden will, transponieren, bevor man sie in die Konstanten-Register schiebt. Anderenfalls müsste man die Multiplikation mit Vektoren umständlicher durchführen.

Die erste Zeile dient also dazu, die Position des Vektors mit der Transformationsmatrix (= aktuelle Weltmatrix), der View-Matrix der Kamera und der Projektionsmatrix zu transformieren. Und nun weiter im Text bzw. Shader:

```
mov oD0, c4
```

*Move your a ...
ambient!*

Die `mov`-Instruktion dient wie bei der Assembler-Programmierung dazu, einen Wert aus einem Register in ein anderes zu kopieren. Wir nehmen hier den Wert aus dem Register `c4`, also die ambiente Farbe, die unsere Engine dort hineingeschrieben hat, und schieben sie einfach weiter in das Output-Register `oD0`. Dieses Register ist dafür da, dem Pixel-Shader den Wert der diffusen Farbe mitzuteilen. Man könnte hier auch schon basierend auf einem Vektor zum einfallenden Licht eine Berechnung unterschiedlicher Beleuchtungsintensität per Vertex vornehmen. Doch dazu später. Wir nehmen also als Farbwert das ambiente Licht und gut. Die letzte Zeile des Vertex-Shaders sieht wie folgt aus:

```
mov oT0, v6
```

In das Output-Register `oT0` gehören die Texturkoordinaten bzw. das erste Paar an Texturkoordinaten. Mehr als das eine Paar haben wir hier nicht. Das schieben wir entsprechend aus dem Input-Register einfach durch, aber natürlich könnte man auch die Texturkoordinaten im Vertex-Shader mani-

pulieren. Das macht man aber eher selten, zumindest bei Texturkoordinaten, die auch wirklich für diffuse Texturen genutzt werden sollen.

Unser Vertex-Shader macht also tatsächlich nichts anderes, als die Position des Vertex zu transformieren sowie die Texturkoordinaten und das ambiente Licht als Wert für die diffuse Farbe an den Pixel-Shader weiterzureichen. Das ist ein sehr kleiner Shader, man könnte natürlich auch noch auf die Texturkoordinaten und das ambiente Licht verzichten. Aber wir wollen doch wenigstens auch verschiedenfarbiges Licht und Texturen auf dem Bildschirm sehen, nicht wahr?

Resümee

Ein einfacher Pixel-Shader

Bei dem einfachen Pixel-Shader kommen wir mit drei Zeilen aus. Auch hier steht als Erstes die Versionsnummer. Was dann folgt, das sehen wir uns hier erst mal an:

```
ps.1.1
tex t0
mul r0, t0, v0
```

Und nun der Pixel-Shader

Das ist in der Tat nicht viel. Die Instruktion `tex t0` dient dazu, den zu einem Pixel zugehörigen Texel aus der Textur der entsprechenden Stage zu *samplen*, also den korrekten Farbwert für den Pixel zu interpolieren, gegebenenfalls zu filtern usw. Die folgende Zeile mit der `mul`-Instruktion dient wie bei der Assembler-Programmierung dazu, die beiden zuletzt angegebenen Faktoren zu multiplizieren und das Ergebnis dann in das zuerst angegebene Zielregister zu schieben. Wie wir ja wissen, ist `v0` das Input-Register für den diffusen Farbwert des Pixels, der aus den verschiedenen Vertices eines Triangles interpoliert wird, wenn man nicht gerade Flat-Shading aktiviert hat. Das Register `v0` entspricht also dem Output-Register `o00` im Vertex-Shader und enthält in diesem Fall hier den Wert des ambienten Lichts, das für alle Vertices gleich ist. Diesen Farbwert multiplizieren wir mit dem Farbwert, den wir aus der Textur geholt haben, und schieben das Ergebnis in das Output-Register `r0`, das den finalen Farbwert für den Pixel am Ende eines jeden Pixel-Shaders erhalten muss.

Schnell erklärt

Das war auch schon der ganze Pixel-Shader: nicht gerade lang und kompliziert, oder? Aber es ginge auch noch kürzer. Die Textur ist ja nur Verschönerung. Wir könnten beispielsweise die Textur ganz weglassen und nur den Wert der diffusen Farbe aus dem Input-Register in das Output-Register schieben. Oder wir könnten auf die diffuse Farbe verzichten und nur den Farbwert aus der Textur verwenden. Das wäre dann die Darstellung analog einem Flat-Shading bei weißem, ambientem Licht.

Es ginge noch kürzer

Demo-Framework

*Einfach – dank
ZFXEngine*

Jetzt, da wir unsere Shader fertig haben, brauchen wir noch eine Demo, um das Ganze mal in Aktion zu sehen. Dazu schreiben wir uns schnell ein Framework, das uns diese und die noch folgenden Shader demonstrieren können soll. Dank unserer *ZFXEngine* und ihrer überragenden Fähigkeiten, uns innerhalb von drei, vier Zeilen eine 3D-Umgebung zur Verfügung zu stellen, ist das auch ein Klacks, und wir kommen mit ein paar Zeilen Quellcode für die Demo-Applikation aus. Diese schreiben wir einfach im C-Stil und implementieren die folgenden Funktionen:



```
LRESULT  WINAPI MsgProc(HWND, UINT, WPARAM, LPARAM);
HRESULT  ProgramStartup(char *chAPI);
HRESULT  ProgramCleanup(void);
HRESULT  ProgramTick(void);
HRESULT  Render(int);
HRESULT  BuildAndSetShader(void);
HRESULT  BuildGeometry(void);
void     CreateCube(ZFXVector, float, float, float,
                  TVERTEX*, WORD*, bool);
```

Funktionen

Wir haben also eine Message-Prozedur wie in einem WinAPI-Programm üblich, dazu die `ProgramStartup()`- und `ProgramCleanup()`-Funktionen. Diese Methoden unterscheiden sich kaum von der Demo-Applikation des letzten Kapitels, daher gehe ich hier nicht weiter darauf ein. Auch die Funktion `CreateCube()` zeige ich hier nicht, denn sie dient nur dazu, die vier Rechtecke für einen Würfel zu erzeugen und mit Texturkoordinaten zu versehen. Das wäre ja sonst weit unter unserem Niveau. Als Parameter übernimmt die Funktion einen Vektor für die Position, an der die Geometrie erzeugt werden soll, drei `float`-Werte für die Breite, Höhe und Tiefe des Würfels, einen Zeiger auf ein Array von 24 Vertices und einen Zeiger auf ein Array von 36 WORDs für die Indices. Der abschließende `bool`-Parameter muss auf `true` gesetzt werden, wenn man möchte, dass die Flächen des Würfels nach innen schauend erzeugt werden.

*Kompilieren der
Shader*

Unsere Engine ist sehr flexibel, was die Aufnahme von Shadern angeht. Für dieses Kapitel schreiben wir alle Shader als einfache ASCII-Dateien mit der Endung `*.vsh` und `*.psh` und übergeben die Dateinamen an unsere Engine. Diese lädt die Shader-Programme dann, kompiliert sie und gibt uns ein Handle auf den einsatzbereiten Shader zurück.



```
UINT g_Base[2] = { 0, 0 };

HRESULT BuildAndSetShader(void) {
    if (!g_pDevice->CanDoShaders()) return S_OK;

    g_pDevice->CreateVShader("base.vsh", 0,
                          true, false,
```

```

        &g_Base[0]);

    g_pDevice->CreatePShader("base.psh", 0,
                           true, false,
                           &g_Base[1]);

    return ZFX_OK;
} // BuildAndSetShader

```

So einfach kann das sein, wenn man sich vorher die Mühe gemacht hat, alles ordentlich und komfortabel im Interface zu designen. Und bevor jemand auf die globalen Variablen schimpft: Das hier ist eine kleine Demo-Applikation, und sie dient nicht zur Demonstration von C++. In einem richtigen Programm würde man die Handles von der Engine in den Attributen einer Instanz irgendeiner Klasse abspeichern.

C++

Gehen wir gleich weiter zur Initialisierung der Geometrie für dieses Beispiel. Wir erzeugen einen Würfel mit nach innen schauenden Flächen, um einen kleinen Demo-Raum zu schaffen. Dazu laden wir gleich eine Textur, damit unser Shader nicht ins Leere sampelt.

```

UINT g_sRoom=0;

HRESULT BuildGeometry(void) {
    HRESULT hr=ZFX_OK;
    TVERTEX v[24];
    WORD    i[36];
    UINT    s=0;

    memset(v, 0, sizeof(TVERTEX)*24);
    memset(i, 0, sizeof(WORD)*36);

    ZFXCOLOR c = { 1.0f, 1.0f, 1.0f, 1.0f };
    ZFXCOLOR d = { 0.0f, 0.0f, 0.0f, 1.0f };

    g_pDevice->GetSkinManager()->AddSkin(&c, &c, &d,
                                         &c, 1, &s);
    g_pDevice->GetSkinManager()->AddTexture(s,
      "texture.bmp", false, 0, NULL, 0);

    // Geometrie für den "Raum"
    CreateCube(ZFXVector(0,0,0), 10.0f, 7.0f, 10.0f,
              v, i, true);

    return g_pDevice->GetVertexManager()->
        CreateStaticBuffer(
            VID_TV, 0, 24, 36,
            v, i, &g_sRoom);
} // BuildGeometry

```



Emissive Light

Dazu gibt es nicht viel zu sagen. Wir erzeugen ein Material für eine Skin, fügen eine Textur hinzu und erzeugen die Geometrie. Dies packen wir dann in einen statischen Buffer unserer Engine und erhalten ein Handle darauf zurück. Man sollte nur darauf achten, dass man im Material das Emissive Light auf Schwarz stellt. Ansonsten erzeugt das Material selbst eine entsprechende Farbe, die dann die von uns berechneten Lichteffekte beeinträchtigen kann. Solange ein Objekt also nicht selbst unabhängig vom Umgebungslicht hell sein soll, muss diese Farbe mit Schwarz angegeben werden. Das Handle für die Skin brauchen wir uns hier übrigens nicht zu merken, da wir nur eine Skin haben und deren ID bzw. Handle daher 0 sein muss.

Update-Funktion

Damit kommen wir zu der Update-Funktion, die in jedem Frame der Anwendung aufgerufen wird. Diese kapselt einfach das Starten und Beenden der Szene, ruft die eigentliche Render-Funktion auf und gibt noch ein wenig Text aus.



```
HRESULT ProgramTick(void) {
    HRESULT hr = ZFX_FAIL;
    ZFXMatrix mat;
    mat.Identity();

    // Back-Buffer Clear
    g_pDevice->BeginRendering(true,true,true);

    Render(-1);

    g_pDevice->UseShaders(false);
    g_pDevice->DrawText(g_nFontID, 10, 10, 255, 255, 0,
        "Basic Shader Demo");

    // Flip Back-Buffer
    g_pDevice->EndRendering();
    return hr;
} // Tick
```

Render-Funktion

Nun kommen wir zu dem interessantesten und zugleich auch letzten Teil des Frameworks für die Shader-Demos: zur eigentlichen Render-Funktion. Diese wird sich in den folgenden Demos mehr oder weniger als einzige ändern. Schauen wir mal, was dort passiert.



```
HRESULT Render(int n) {
    ZFXMatrix mat;
    mat.RotaY(-0.4f);
    mat._42 -= 0.5f;
    mat._41 -= 1.5f;

    // zirkuliere die Farbe des ambienten Lichts
```

```

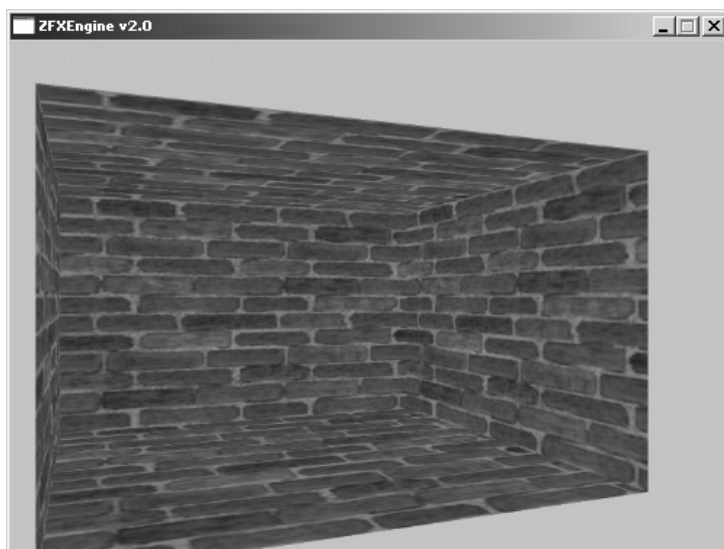
float fT = GetTickCount() / 1000.0f;
float fR = 0.5f + 0.5f * sinf(fT*1.2f);
float fG = 0.5f + 0.5f * sinf(fT*2.0f);
float fB = 0.5f + 0.5f * sinf(fT*1.7f);

g_pDevice->SetAmbientLight(fR, fG, fB);
g_pDevice->SetWorldTransform(&mat);

// Korrekte Shader, weißt du!
g_pDevice->ActivateVShader(g_Base[0], VID_TV);
g_pDevice->ActivatePShader(g_Base[1]);

return g_pDevice->GetVertexManager()->Render(
                                                g_sRoom);
} // Render

```

**Abbildung 7.4:**

Ein texturierter Raum in weißem, ambientem Licht, gerendert über Shader

Wir erstellen uns eine Rotationsmatrix und rotieren und verschieben die Szene ein bisschen, damit wir den Raum gut im Blick haben. Dann berechnen wir in jedem Frame die Farbe des ambienten Lichts über die trigonometrischen Funktionen neu. Wenn man einfach nur Zufallswerte nehmen würde, dann hätte man einen sehr abrupten Wechsel in der Farbe. Durch die Sinus- und Cosinus-Funktionen erhält man einen sauberen Übergang über das Farbspektrum. Dann aktivieren wir noch die Shader, und das war auch schon alles. Nun haben wir den Raum auf dem Bildschirm, zusammen mit einer Steinwand-Textur, und die Farbe des ambienten Lichts, das gleichmäßig auf allen Flächen auftritt, zirkuliert über das gesamte Spektrum und wird dann zu dem roten Grundton der Textur hinzumultipliziert. Damit haben wir unseren ersten echten Shader schon komplett. Einen Screenshot wollte ich mir hier eigentlich sparen, aber ich zeige ihn trotzdem, weil wir

*Unsere drei
trigonometrischen
Freunde*

anhand dieses Screenshots die Entwicklung unserer Shader später besser vergleichen können. Abbildung 7.4 zeigt den im Moment noch recht öde aussehenden Raum.

Demo 2: Single-Pass-Multitexturing

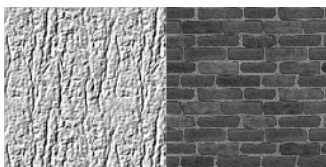
Diffuse Texturen und Detail- Mapping

Dieser Raum ist uns zu öde? Dann pepen wir ihn halt ein wenig auf. Zuerst packen wir eine Detailmap dazu. Falls jemand noch nicht wissen sollte, was eine Detailmap ist, erkläre ich das hier schnell. Die Textur, deren Farben man direkt auf die einzelnen Pixel eines Triangles mappt und nur durch die Lichtintensität abdunkelt, nennt man in Anspielung auf das diffuse Licht in einer Szene auch diffuse Textur. Natürlich kann man Texturen auch zu anderen Zwecken benutzen, beispielsweise kann man eine zusätzliche Textur als Lightmap oder als Detailmap einsetzen. Dabei ist die Farbe der Textur nicht entscheidend, da sie nicht direkt angezeigt wird. Der Farbwert aus dieser Textur wird auch dazu verwendet, die einzelnen Pixel eines Triangles aufzuhellen oder abzdunkeln. Durch Lightmaps möchte man hauptsächlich helle Lichtkegel auf der diffusen Textur erzeugen, ohne das Licht jedoch berechnen zu müssen. Eine Detailmap setzt man dazu ein, der diffusen Textur eine zusätzliche Struktur zu geben, damit diese detaillierter erscheint. Das setzt man vor allem dann ein, wenn man eine diffuse Textur über eine sehr große Fläche strecken muss und diese dann sehr grobkörnig und undetailliert aussieht. Eine Detailmap kann man auf dieser großen Fläche dann sehr oft wiederholen und so Schattierungsunterschiede in der diffusen Textur erzeugen.

Single-Pass- Multitexturing

Genau eine solche Detailmap wollen wir unserem Raum nun zusätzlich verpassen. Abbildung 7.5 zeigt die beiden Grafiken, die wir dazu als Texturen verwenden. Die links gezeigte Grafik nehmen wir dabei als Detailmap. Die Frage ist nun, wie wir diese dazu bekommen, ihre Struktur auf die diffuse Textur zu übertragen? Dazu verwenden wir natürlich das Multitexturing. Das bedeutet, dass wir multiple Texturen gleichzeitig beim Rendern verwenden, also in einem Render-Pass. Darum nennt man das Verfahren *Single-Pass-Multitexturing*. Die Alternative auf älteren Grafikkarten, die nur eine Textur verwenden können, besteht darin, die Geometrie einmal mit einer Textur zu rendern und dann mit speziellen Einstellungen ein zweites Mal mit einer anderen Textur. Das nennt man dann entsprechend *Multi-Pass-Rendering*.

Abbildung 7.5:
Die beiden verwendeten
Texturen:
links die Detailmap
und rechts die
diffuse Textur



Vertex-Shader

Dann mal ran an den Speck, sonst beißen ihn die Hunde. An unserem bisherigen Vertex-Shader müssen wir gar nichts ändern – besser gesagt: fast nichts. Die folgende Zeile müssen wir noch ergänzen:

```
mov oT1, v6
```

Wir kopieren die Texturkoordinaten aus dem Input-Register `v6` also nicht nur in das Output-Register `oT0`, sondern nun zusätzlich auch in das Output-Register `oT1`. Wir verwenden in unserem Beispiel zwei Texturen in den Stages 0 und 1, und daher müssen wir auch für beide Stages Texturkoordinaten zum Pixel-Shader schieben.

Pixel-Shader

Im Pixel-Shader tut sich nun schon etwas mehr. Allerdings haben wir auch keine Schwierigkeiten damit, es zu verstehen. Wir müssen nun natürlich zunächst zwei Texturen sampeln und nicht mehr nur eine. Dann multiplizieren wir wie gehabt den gesampelten Texel mit dem ambienten Licht aus dem Vertex-Shader, das sich im Input-Register `v0` befindet. Allerdings tun wir das diesmal auch nicht nur einmal, sondern zweimal in zwei verschiedene Register – nämlich einmal für die diffuse Textur in der Stage 0 und einmal für die Detailmap in der Stage 1. Zuletzt multiplizieren wir diese beiden Zwischenergebnisse zusammen und erhalten somit eine gleichwertige Blending der beiden Texturen. Unser neuer Shader sieht daher so aus:

```
ps.1.1
tex t0
tex t1
mul r0, v0, t0
mul r1, v0, t1
mul r0, r0, r1
```

Für den letzten Schritt hätte man auch die beiden vorher berechneten Farbwerte addieren können, anstatt sie zu multiplizieren. Dann wäre das Endergebnis natürlich wesentlich heller gewesen. Einen solchen Effekt kann man für *Glowmaps* verwenden, die schwarz grundiert sind und an den Stellen, an denen sie die diffuse Textur zum Leuchten bringen sollen, helle Flecken haben. Resümierend kann man hier nur sagen, dass das Blenden multipler Texturen erschreckend einfach ist. Wer hätte das gedacht.

Demo-Framework

An unserem Demo-Framework müssen wir auch nichts ändern, außer dass wir unserer Skin noch eine zweite Textur hinzufügen. Das Zuordnen der Textur zu der korrekten Stage erledigt ja unsere fleißige Engine bereits intern für uns. Wir müssen einfach nur mit dem Shader auf sie zugreifen.

*Kaum was Neues
hier*

Ich bin drin!

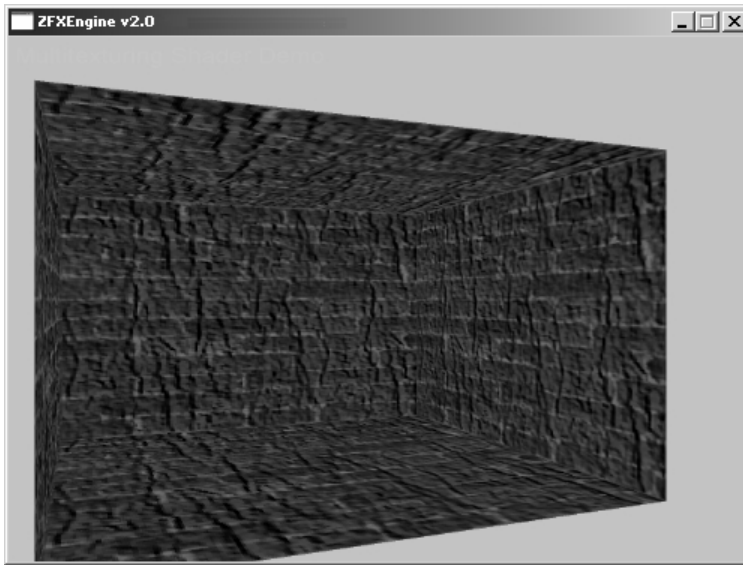


Das ist ja einfach.

*Es gibt nichts zu
tun. Lassen wir es
bleiben.*

Abbildung 7.6 zeigt, wie das Programm in Aktion aussieht. Im Vergleich zu Abbildung 7.4 erkennt man deutlich die Strukturen, die die Detailmap auf der diffusen Textur hinterlässt. Klasse.

Abbildung 7.6:
Derselbe Raum,
aber diesmal mit
diffuser Textur und
Detailtextur
zusammen



Demo 3: Directional Lighting per Pixel

Her mit dem Licht!

Das ist ja alles gut und schön, aber bisher haben wir nur ambientes Licht, das alle Triangles in einer Szene gleich beleuchtet und in der gleichen Helligkeit erscheinen lässt. Mit der Detailmap haben wir auch eine Technik kennen gelernt, wie wir beim Licht ein bisschen faken können. Doch nun wollen wir endlich sehen, wie man in einem Shader wirklich etwas Licht ins Dunkel bringen kann, das eben nicht statisch für alle Triangles gleich ist, sondern sich je nach Ausrichtung der Triangles zum Lichtvektor unterscheidet. Kurz gesagt: Directional Light. Wiederholen wir also schnell die Grundlagen gerichteten Lichts. Es handelt sich dabei um Licht, das parallel läuft und von einer sehr weit entfernten Lichtquelle (oder einem Laser) stammt. Somit hängt die Intensität der Beleuchtung eines Triangles nur von seiner Ausrichtung zum Vektor des einfallenden Lichts ab. Alle Triangles mit derselben Ausrichtung werden mit derselben Intensität beleuchtet. Und das implementieren wir nun über einen Shader.

Vertex-Shader

Wenig ist oft mehr

In unserem Vertex-Shader kommt auch diesmal wieder nur wenig hinzu. Aber das hat es in sich. Wir geben dem Shader aus unserer Applikation heraus den Richtungsvektor des gerichteten Lichts an. In unserem Vertex-Shader führen wir dann die Berechnung durch, die die Fixed-Function-Pipe-

line auch vornehmen würde. Wir berechnen nämlich das Punktprodukt zwischen dem Lichtvektor und dem Normalenvektor des Vertex, um so zu evaluieren, wie stark der Normalenvektor zum Licht hinzeigt und entsprechend intensiv beleuchtet werden muss. Diese Zeile sieht wie folgt aus:

```
dp3 oD1, v3, -c20
```

Durch die Instruktion `dp3` berechnen wir das Punktprodukt nur für die ersten drei Komponenten der jeweiligen Register, sprich für die x -, y - und die z -Koordinate, denn w ist hierbei vollkommen uninteressant. Im Input-Register `v3` befindet sich ja die Vertex-Normale, und in das Konstanten-Register `c20` muss unsere Applikation den Richtungsvektor des Lichts schieben. Diesen müssen wir hier jedoch negieren, damit wir den Vektor erhalten, der vom Vertex genau zum Licht hinzeigt, denn sonst ist das Punktprodukt quasi invers und sagt nicht das aus, was wir von ihm wissen wollen. Das Ergebnis des Punktprodukts schieben wir in das Output-Register `oD1`, um es dem Pixel-Shader zugänglich zu machen.

Pixel-Shader

In unserem bisherigen Pixel-Shader haben wir einfach die aus der Textur errechnete Farbe mit dem Wert des ambienten Lichts multipliziert, um sie entsprechend des Lichts in der Szene anzudunkeln. Nun haben wir jedoch zwei Lichtwerte je Pixel: zum einen weiterhin das ambiente Licht und zum anderen noch das parallele Licht, dessen Intensität wir per Vertex über das Punktprodukt ausgerechnet haben. Nun müssen wir einfach diesen Wert zu dem ambienten Licht addieren, weil sich beide Lichtwerte in der Szene aufaddieren. Danach erst multiplizieren wir mit der Farbe aus der Textur. Hier ist der komplette Pixel-Shader:

*Lichtwerte
aufaddieren*

```
ps.1.1
tex t0
tex t1

add r0, v0, v1
mul r0, r0, t0
add r1, v0, v1
mul r1, r1, t1

mul r0, r0, r1
```

Das ist keine große Kunst. Wir müssen zuerst alle in der Szene vorhandenen Lichtwerte aufaddieren und erhalten so einen endgültigen Wert für die Lichtintensität, die ein bestimmter Pixel hat. Mit dieser modifizieren wir die Farbe aus der Textur, und fertig. Hier machen wir das auch gleich für zwei Texturen (wie im vorherigen Beispiel), die wir dann zusammenblenden.



In den hier gezeigten Beispielen tun wir eines nicht: Wir berücksichtigen das Material nicht. Unsere Engine fügt aber automatisch beim Rendern im Vertex-Cache-Manager die Werte des Materials eines Triangles für ambient, diffuse, emissive und specular reflection in die Konstanten-Register c1 bis c4 für den Pixel-Shader ein. Man kann also ganz bequem über das Material festlegen, dass ein Objekt gar kein ambientes Licht reflektieren soll oder nur die Hälfte des diffusen Lichts usw. Dazu müsste man den Wert der jeweiligen Lichtintensität, wie hier beispielsweise ambientes Licht im Input-Register v0 und diffuses Licht im Input-Register v1, noch mit dem entsprechenden Konstanten-Register multiplizieren. Den Wert für das emissive Light würde man danach hinzuaddieren, weil das Objekt dieses Licht ja selbst erzeugt. Erst dann hat man die tatsächliche Lichtintensität eines Pixel unter Berücksichtigung des Materials und kann mit der Farbe aus der Textur multiplizieren.

Demo-Applikation

Weltkoordinaten
ungleich Objekt-
koordinaten

Wieder einmal müssen wir an unserer Demo-Applikation kaum etwas ändern. Allerdings sollte uns weiter oben schon eine kleine Ungereimtheit aufgefallen sein. Wir multiplizieren im Vertex-Shader den immer noch *untransformierten* Normalenvektor mit dem inversen Vektor der Lichtrichtung. Der Letztgenannte liegt aber normalerweise direkt im Weltkoordinatensystem vor, während der Normalenvektor in der untransformierten Form natürlich im lokalen Koordinatensystem des Objekts ist, zu dessen Vertex er gehört. Dies nennt man auch das *Objektkoordinatensystem* (engl. object space) oder in einigen anderen Schriften auch das *Modellkoordinatensystem* (engl. model space). Berechnungen zwischen Vektoren ergeben jedoch nur dann sinnvolle Werte, wenn man sie im selben Koordinatensystem vorliegen hat. Nun könnten wir den Normalenvektor im Vertex-Shader auch transformieren und ihn somit in das Weltkoordinatensystem überführen. Das würde auch funktionieren, wenn wir es richtig machen. Es ist aber unnötig viel Arbeit, denn diese Berechnung müssten wir für jeden Vertex einmal durchführen. Zudem bräuchten wir die Weltmatrix des Objekts als zusätzlicher Input für den Shader, was einem erhöhten Traffic gleichkommt. Stattdessen greifen wir wieder in die Optimierungskiste und transformieren in unserer Applikation den Vektor des Lichts einfach vom Weltsystem in das lokale System des Modells. Dazu verwenden wir natürlich die inverse Weltmatrix des Objekts. Der Shader darf also davon ausgehen, dass der Vektor des Lichts bereits im selben Koordinatensystem ist wie die untransformierte Normale eines Vertex. Und hier ist unsere leicht geänderte `Render()-Funktion`:



```
HRESULT Render(int n) {
    static float fR = -0.4f;

    ZFXMatrix mat, matInv;
    ZFXVector vcLightDir(0.0f, 0.0f, 1.0f);
```

```

// rotiere den Raum
if (fR < -6.283185f)
    fR += 6.283185f;

fR -= 0.02f;

mat.RotaY(fR);
mat._42 -= 0.5f;
mat._41 -= 0.5f;

g_pDevice->SetAmbientLight(0.2f, 0.2f, 0.2f);
g_pDevice->SetWorldTransform(&mat);

// Lichtvektor zum lokalen System transformieren
matInv.InverseOf(mat);
vcLightDir = matInv * vcLightDir;

// Lichtvektor in Register c20 des Vertex-Shaders
g_pDevice->SetShaderConstant(SHT_VERTEX, DAT_FLOAT,
                             20, 1, &vcLightDir);

g_pDevice->ActivateVShader(g_Base[0], VID_TV);
g_pDevice->ActivatePShader(g_Base[1]);

return g_pDevice->GetVertexManager()->Render(
                                         g_sRoom);
} // Render

```

Damit man von dem Licht auch etwas sehen kann, versetzen wir den Raum einfach in Rotation. So kann man sehen, dass das Licht immer in dieselbe Richtung scheint und die Flächen des invertierten Würfels tatsächlich ihre Schattierung ändern, wenn sie sich vom Licht wegdrehen. Abbildung 7.7 zeigt einen Screenshot unserer Applikation.

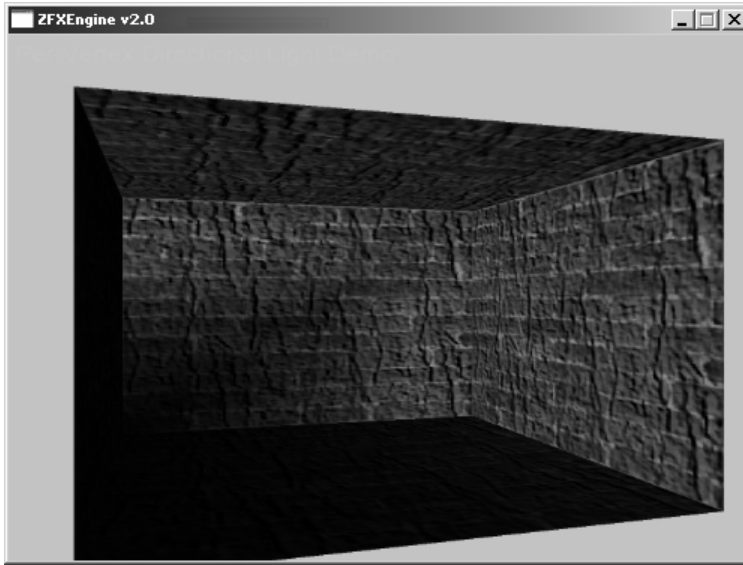
Demo 4: Per-Pixel-Omni-Lights

Cool. Aber doch eigentlich ziemlich langweilig, wenn man bedenkt, dass sich weder eine direktionale Lichtquelle noch die Geometrie eines Raumes normalerweise bewegen und die Szene entsprechend lahm aussähe. Wir brauchen also etwas Peppigeres, und das machen wir jetzt. Das letzte Beispiel war ja auch nur eine Beleuchtung per Vertex. Und da die Normalen aller Vertices einer Fläche des Raums auch noch identisch waren, hatten wir sogar nur eine Beleuchtung per Plane, und die Lichtintensität war auf allen Triangles in einer Ebene gleich. Wenn man irgendwo das Wort Pixel-Shader hört, so schallt es doch aber meistens *Per-Pixel-Lighting* von irgendwo anders zurück. Wo ist also das famose Per-Pixel-Lighting ohne den billigen Ausweg über Lightmaps oder Ähnliches? Wir hätten doch gern echtes Real-time-Per-Pixel-Lighting zur Laufzeit.

*Krass, das Pferd.
Aber da geht noch
was.*

Abbildung 7.7:

Derselbe Raum mit Multitexturing, aber diesmal zusätzlich zum ambienten Licht mit Per-Vertex-Directional-Light beleuchtet.

**Omni-Lights**

Nun gut. Dann implementieren wir das hier mal. Wir spezialisieren uns dabei auf so genannte Omni-Lights. Im Direct3D-Kontext nennt man diese auch *Point-Lights* bzw. *Punktlichtquellen*. Diese geben ihr Licht gleichmäßig in alle Richtungen ab und werden innerhalb eines vordefinierten Radius immer dunkler. Diese Art von Licht ist sehr einfach zu berechnen, macht aber ordentlich was her. Als kleinen Appetithappen zeige ich hier schon mal den Screenshot in Abbildung 7.8. Dort sieht man zwei Omni-Lights (die beiden weißen Rechtecke) und das Licht, das sie in den Raum pumpen. Das Directional Light habe ich aus dem Demo wieder herausgenommen, da wir in unserer Welt mit Ausnahme der Sonne eigentlich keine relevanten Quellen für paralleles Licht haben. Der Screenshot sieht hier auch nicht so schön aus. Zum einen ist er nur schwarzweiß, und zum anderen sieht man nicht, dass sich die Lichter durch den Raum bewegen und ihre Farbe dabei ändern.

DirectX SDK-Lighting-Sample

Im DirectX SDK gibt es ein recht ähnliches Beispiel mit einem Raum, das allerdings vollkommen ohne Shader auskommt und sogar `D3DLIGHT9`-Objekte, also echte Hardware-Lichter, verwendet. Warum machen wir dann hier so einen Aufriss um diese tolle Beleuchtung? Nun der erste Punkt ist ganz einfach, dass keine Grafikkarte mehr als acht solcher Hardware-Lichtquellen gleichzeitig aktiv haben kann. Das ist bei weitem nicht genug, um eine vernünftige Beleuchtung in einem Indoor-Spiel umzusetzen. Ein anderer Punkt ist aber der kleine Schwindel, den das Sample verwendet. In Abbildung 7.9 erkennt man den kleinen Trick, wenn man sich das Sample im Wireframe-Modus anschaut. Man erkennt, warum das Per-Vertex-Lighting dort so unheimlich gut aussieht. Der Raum besteht eben nicht aus nur 24 Vertices und zwölf Triangles wie unser Raum.

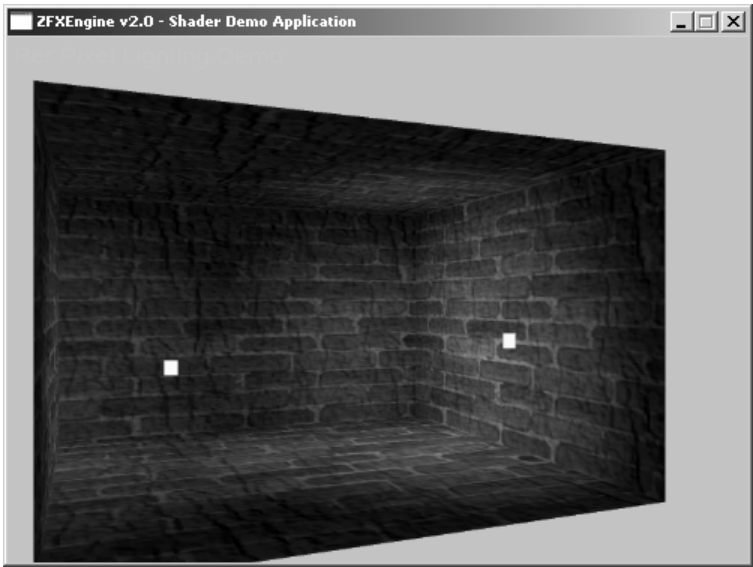


Abbildung 7.8:
Und wieder dieser Raum – diesmal mit zwei verschiedenfarbigen Omni-Lights, die ihr Licht spendieren

Man hat dort eine einzige Fläche des Raums so weit in kleine Mini-Flächen zerlegt, dass die Vertices sehr dicht beieinander liegen und so hinreichend Helligkeitsunterschiede zwischen den einzelnen Vertices vorhanden sind, so dass das normale Gouraud-Shading ausreicht, um einen schönen Lichtkegel zu erzeugen.

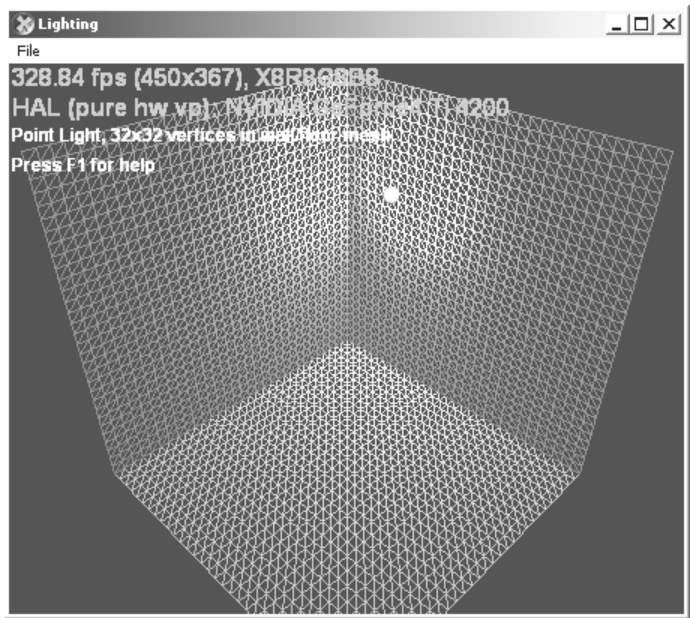


Abbildung 7.9:
Das DirectX SDK-Sample mit einem Point-Light im Wireframe-Modus

*Performance,
Performance,
Performance*

Würden wir ein solches Hardware-Licht auf unseren Raum loslassen, wo jede Fläche wirklich nur vier vergleichsweise weit auseinander liegende Vertices hat, so sähen wir allenfalls ein bisschen Veränderung in der Helligkeit der Fläche, aber garantiert keinen Lichtkegel. Für so ein kleines Sample ist das natürlich cool und sieht richtig fett aus. Aber wenn man versucht, in einem Level aus 50.000 Flächen jede davon so weit zu unterteilen, wie man es müsste, damit das Per-Vertex-Lighting akzeptabel aussieht, dann hätte man eine Anzahl an Vertices und Triangles, die auf keine Kuhhaut mehr gehen, geschweige denn in Echtzeit gerendert werden würde.

Radiosity

Nach demselben Prinzip arbeitet übrigens Radiosity. Hier unterteilt man jede Fläche in kleine Patches und berechnet dann ausgehend von allen Lichtern in der Szene den Helligkeitwert, den ein Patch dort haben müsste. Dadurch kann man ein sehr schön schattiertes Bild generieren, aber keinesfalls in Echtzeit – selbst mit der schnellsten GeForce XXL-Karte nicht. Ein Problem dabei ist auch, wie man ein beliebiges Input-Mesh fehlerfrei in Patches zerlegen kann. Das Verfahren wird jedoch gern benutzt, um Lightmaps für einen Level zu erzeugen. Aber das führt hier ein wenig zu weit.

Omni-Light-Matrix berechnen

*Das Licht als Maß
aller Dinge*

Bevor wir mit unseren Shadern loslegen können, brauchen wir noch eine Funktion für unsere Applikation, die uns für ein gegebenes Omni-Light, also eine Position im Raum und einen Radius für die Reichweite des Lichts, eine bestimmte Matrix berechnet, mit der wir nachher jeden Vertex transformieren müssen. Ich zeige zuerst einmal die Funktion und erläutere sie dann.



```
ZFXMatrix CalcTransAttenNoRot(const ZFXVector &vcPos,
                               float fRadius) {
    ZFXMatrix mA, mS, mTL, mB2, mTP, mX;

    float invRad = 0.5f / fRadius;

    mS.Identity();
    mB2.Identity();
    mTL.Identity();

    mS._11 = mS._22 = mS._33 = invRad;
    mTL.Translate(-vcPos.x, -vcPos.y, -vcPos.z);
    mB2.Translate(0.5f, 0.5f, 0.5f);

    mA = mTL * mS;
    mX = mA * mB2;

    mTP.TransposeOf( mX );
    return mTP;
} // CalcTransAttenNoRot
```

Wir möchten mit dieser Matrix genau zwei Dinge erreichen: Erstens soll ein Vertex so verschoben werden, dass das Licht im Zentrum des Koordinatensystems liegt. Dazu müssen wir die Position des Vertex einfach um das Inverse der Position des Lichts verschieben. Dies tun wir mit der Matrix m_{TL} in der Funktion. Zum anderen wollen wir die Position des Vertex so skalieren, dass die Lichtkugel des Omni-Lights einen Einheitsradius hat, und nicht mehr einen Radius der Länge $fRadius$. Dazu teilen wir ganz einfach $1 / Radius$ und skalieren die Position des Vertex mit diesem Wert. Das erledigen wir in der Funktion über die Matrix m_S . Nun hat sich aber noch eine $0.5f$ dort in die Funktion gemogelt. Aber das ist schnell erklärt. Wenn wir die eben erklärte Berechnung durchführen, dann haben wir innerhalb der Lichtkugel des Omni-Lights Werte für die Vertices zwischen -1 und $+1$. In den Shadern werden die Werte der Register aber auf den Bereich 0 bis $+1$ geclippt, also müssen wir dem vorbeugen und die Informationen der Werte dennoch erhalten. Dazu multiplizieren wir alle Werte mit 0.5 und addieren dann noch 0.5 . Dadurch verschieben wir die Werte aus dem Bereich -1 bis $+1$ in den Bereich 0 bis $+1$ und erhalten dennoch ihren Informationswert. Wenn wir die Originalwerte wieder herstellen wollen, dann ziehen wir einfach $0.5f$ ab und multiplizieren mit 2 (dividieren also durch $0.5f$). Die Multiplikation mit $0.5f$ haben wir in dieser Funktion gleich in den Term $1 / Radius$ integriert. Zu guter Letzt multiplizieren wir die einzelnen Werte miteinander und transponieren die Matrix. Letzteres ist nur nötig, damit die Matrix im Shader richtig eingesetzt werden kann. Da wir diese Matrix aber auch nur im Shader brauchen, habe ich das gleich mit in die Funktion gepackt.

Nun schauen wir uns an, was wir mit Hilfe dieser Matrix und einem Vertex-Shader machen können.

Vertex-Shader

Wir stehen vor einem Dilemma. Es ist nicht sehr einfach, wenn nicht gar in den ersten Shader-Versionen unmöglich, eine Schleife über eine Variable laufen zu lassen. Wir möchten aber eine Szene rendern können, die beliebig viele Omni-Lights enthält. Wir müssen aber jedes Omni-Light in Betracht ziehen und dessen Einfluss auf einen Pixel korrekt berechnen. Eine Möglichkeit wäre es, verschiedene Shader für verschiedene Anzahlen an Omni-Lights zu verwenden. Aber einerseits limitieren wir uns damit selbst, wie beispielsweise die Hardware-Lights auf acht Stück limitiert sind, und andererseits ist auch die Anzahl möglicher Instruktionen und Register in einem Shader begrenzt. Der bessere Weg ist es also, multiple Render-Passes zu verwenden. Wir werden also unsere Szene einmal in rein ambienter Darstellung rendern. Dann aktivieren wir das additive Blending unserer Engine und rendern die Szene noch einmal, und zwar noch je einmal pro Omni-Light in der Szene.

*Das Licht als
Zentrum des
Universums*

*Multiple Shader
versus multiple
Passes*

Performance? Das klingt zwar nun sehr langsam, ist es aber nicht. Im ambienten Pass wird der Depth-Buffer bereits vollständig betankt. Daher kann die Grafikkarte in den folgenden Passes ein so genanntes *early z-culling* durchführen, also das schnelle Aussortieren von Pixeln, die in der Szene nicht sichtbar sind, weil sie durch Overdraw mit anderen Pixeln übermalt würden. Die Lichtberechnungen mit dem Omni-Light werden also nur für diejenigen Pixel ausgeführt, für die sie wirklich nötig sind. Auf einer GeForce4 Ti können wir in Echtzeit etliche Dutzend Omni-Lights eine Szene aus ein paar Tausend Polygonen beleuchten lassen und können die gesamte Szene im View-Frustum haben und in Echtzeit darstellen. Performance wird nur dann zu einem Problem, wenn ein Omni-Light eine sehr große Menge an Polygonen beleuchtet und wir sehr viele dieser sehr großen Omni-Lights haben. Aber unsere Lichter scheinen normalerweise nur ein paar Meter weit und daher auch nur auf ein paar hundert Polygone.

Das sind zwei Paar Shader Wir benötigen also im Folgenden je zwei Vertex- und Pixel-Shader. Ein Paar dient dazu, den ambienten Pass zu rendern. Dazu verwenden wir die Basic-Shader, die wir als erstes Beispiel in diesem Abschnitt entworfen haben. Das zweite paar Shader wird für den Pass eines Omni-Lights verwendet. Diese Shader wollen wir uns hier näher ansehen. Beginnen wir mit dem Vertex-Shader:

```
vs.1.1
dcl_position0 v0
dcl_normal0 v3
dcl_texcoord0 v6
m4x4 oPos, v0, c0
mov oT0, v6
m4x4 oT1, v0, c20
```

Omni-Light Matrix in c20 Langweilig. Da passiert nichts, was uns nicht schon bekannt wäre. Wir transformieren die Position des Vertex und schieben sie raus. Die Texturkoordinaten der ersten Stage leiten wir einfach durch. Es sei hier noch erwähnt, dass wir zwar für den ambienten Pass Multitexturing verwenden, aber nicht für die Passes der Omni-Lights. Dadurch wird die Struktur der Detailmap ein wenig überstrahlt, aber wir sparen das Sampling der zweiten Textur für jedes Omni-Light, und das Ergebnis sieht optisch zufrieden stellend aus. Mit der letzten Instruktion im Shader transformieren wir die Position des Vertex mit der Matrix, die zu diesem Omni-Light gehört und die wir in unserer Applikation in das Konstanten-Register `c20` geschoben haben. Hier missbrauchen wir das Register `oT1`, um diese transformierte Position an den Pixel-Shader weiterzuleiten. Anders kann ein Vertex-Shader keine Informationen an den Pixel-Shader liefern. Damit ist der Shader schon zu Ende.

Pixel-Shader

Bisher haben wir im Pixel-Shader immer fleißig die Instruktion `tex` verwendet. Vielleicht sollten wir nun etwas mehr dazu erklären. Sie dient dazu, die Texturkoordinaten zu nehmen, die der Vertex-Shader in sein entsprechendes Output-Register geschoben hat, sie zu interpolieren und die korrespondierende Farbe aus der Textur in das Register `tx` zu füllen. Wenn wir nun aber andere Werte über den Vertex-Shader dort geparkt haben, dann wäre das ja unsinnig. Es gibt hierfür noch die Instruktion `texcoord` im Pixel-Shader. Über diese erhalten wir direkt den interpolierten Wert aus dem Register und nicht den Farbwert aus der Textur. Hier müssen wir diese Instruktion anwenden, um aus dem Register `t1` die Position des Pixels im Omni-Light Space zu erhalten. Aufgrund unserer fluffigen Matrix ist das der Abstandsvektor des Pixels zur Lichtquelle in der Einheitskugel um das Omni-Light herum. Der gesamte Pixel-Shader sieht so aus:

```
ps.1.1
tex      t0
texcoord t1
dp3_sat r0, t1_bx2, t1_bx2
mul r0, c0, 1-r0
mul r0, r0, t0
```

Die Zeile mit dem Punktprodukt ist hier des Pudels Kern. Wir multiplizieren den Abstandsvektor mit sich selbst, haben dabei aber noch die Suffixe `_sat` und `_bx2` an der Instruktion bzw. an den Variablen hängen. Letzteres sorgt dafür, dass zunächst ein so genannter Bias ausgeführt wird. Das ist nichts anderes als die Subtraktion von $0.5f$ von den einzelnen Koordinaten. Das `x2` sorgt dann dafür, dass alle Koordinaten mit 2 multipliziert werden. Das ist genau die Wiederherstellung der Originalwerte im Bereich von -1 bis $+1$ aus dem Bereich von 0 bis $+1$, von der ich vorhin sprach. So haben wir das Clipping der Vertex-Shader-Output-Register ausgetrickst. Das Suffix `_sat` macht nun nichts anderes, als das Ergebnis in den Bereich 0 bis 1 zu clampen. Das erscheint zunächst vielleicht etwas unsinnig, weil wir grad erst den negativen Bereich wieder hergestellt haben, aber wir quadrieren den Vektor ja, und dabei werden alle negativen Werte entsprechend positiv. Es geht hier nur darum, Werte größer als 1 auszuschließen, weil diese ungültige Helligkeitsintensitäten wären. Am Ende der Instruktion haben wir im Register `r0` in den ersten drei Koordinaten jeweils das Quadrat der Länge des Vektors stehen, denn wir wissen ja, dass der Betrag (also die Länge) eines Vektors sich über die Formel $\sqrt{x*x + y*y + z*z}$ berechnet. Wenn wir in dieser Formel das Wurzelzeichen weglassen, dann haben wir genau die Formel für das Punktprodukt, deren Ergebnis nun in den ersten drei Feldern von `r0` steht.

Nun multiplizieren wir $1-D^2$ mit dem Konstanten-Register `c0`. In dieses Register muss unsere Anwendung die Farbe schieben, die das Omni-Light hat. Auf diese Weise wenden wir hier die Formel an, dass die Helligkeitsin-

Input auswerten

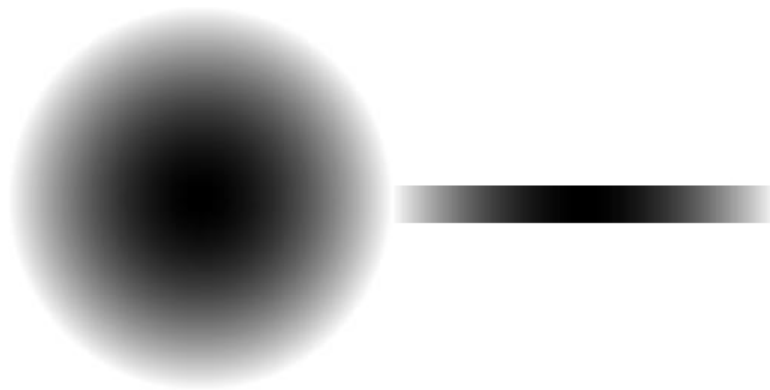
Saturation und Bias

Farbe des Lichts

tensität eines Punktes im Raum mit dem Quadrat seines Abstandes zur Lichtquelle abnimmt. Das bezeichnet man auch allgemein als *Fall Off* oder *Attenuation*. Wir haben nun endlich die Helligkeitsintensität berechnet, die der Pixel in Relation zu dem Omni-Light haben muss. Diese multiplizieren wir noch mit der Farbe aus der Textur und haben damit den endgültigen Farbwert für den Pixel ermittelt. Ganz analog wird diese Berechnung über so genannte *Attenuation Maps* umgesetzt, die in Abbildung 7.10 zu sehen sind.

Abbildung 7.10:

Links die 2D-Attenuation Map und rechts die 1D-Attenuation Map, die zur besseren Kennntlichkeit vertikal gestreckt ist.



Attenuation Maps

Die Attenuation Maps dienen nur dazu, die Beleuchtungsintensität im Bereich von 0 bis 1 darzustellen, mit einem quadratischen Abfall über die Distanz. Wir haben die oben gezeigte Funktion verwendet, um eine schlaue Matrix zu erzeugen die uns über den Trick mit der Einheitskugel um ein Omni-Light dasselbe ermöglicht wie diese Attenuation Maps. Idealerweise würde man eine 3D-Textur verwenden, um eine solche Einheitskugel darzustellen. Aber das unterstützen nur sehr moderne Grafikkarten. Also zerlegt man, wie in Abbildung 7.10 zu sehen ist, die Kugel in eine 2D-Textur und eine 1D-Textur, wobei erstere den x- und y-Teil darstellt und letztere den verbleibenden z-Teil. Über den Shader kann man nun die entsprechenden Werte aus den beiden Texturen auslesen. Bei unserer Methode der Umrechnung der Vertex-Position haben wir allerdings den Vorteil, dass wir keine zusätzlichen Texturen verwenden müssen.

Demo-Applikation

Additives Rendering

Damit wissen wir auch schon alles, was wir über Per-PixelOmni-Lights wissen müssen. Jetzt können wir so viele Omni-Lights wie wir wollen in unsere Szene setzen. Abbildung 7.8 zeigte ja bereits einen Screenshot der Applikation. In unserem Framework müssen wir aber ein paar Dinge umstricken. Beginnen wir am besten mit der neuen `ProgramTick()`-Funktion.



```
// Geometrie
UINT g_sRoom=0;
UINT g_sLight=0;

// Shader
UINT g_Base[2] = { 0, 0 };
UINT g_Omni[2] = { 0, 0 };

// Licht-Attribute
ZFXVector g_vcL[2];
ZFXCOLOR g_clrL[2] = { { 1.0f,1.0f,1.0f,1.0f},
                       { 1.0f,1.0f,1.0f,1.0f } };

HRESULT ProgramTick(void) {
    HRESULT hr = ZFX_FAIL;
    ZFXMatrix mat;
    mat.Identity();

    // Wert über die Zeit bilden
    float fT = GetTickCount() / 1000.0f;

    // Licht smooth bewegen
    g_vcL[0].Set( sinf( fT*2.0f) + 2.0f,
                  cosf( fT*2.5f),
                  sinf( fT*2.0f) );

    g_vcL[1].Set( cosf( fT*2.5f) - 4.0f,
                  sinf( fT*1.8f) * 2.0f,
                  sinf( fT*2.3f) + 2.0f);

    // Lichtfarbe smooth ändern
    g_clrL[0].fR = 0.5f + 0.5f * sinf(fT*2.0f);
    g_clrL[0].fG = 0.5f + 0.5f * sinf(fT*2.35f);
    g_clrL[0].fB = 0.5f + 0.5f * sinf(fT*2.7f);

    g_clrL[1].fR = 0.5f + 0.5f * cosf(fT*2.0f);
    g_clrL[1].fG = 0.5f + 0.5f * sinf(fT*1.8f);
    g_clrL[1].fB = 0.5f + 0.5f * sinf(fT*2.0f);

    g_pDevice->BeginRendering(true,true,true);

    RenderLight(g_vcL[0].x, g_vcL[0].y, g_vcL[0].z);
    RenderLight(g_vcL[1].x, g_vcL[1].y, g_vcL[1].z);
    Render(-1); // Ambierter Pass
    Render(0);  // Light-0-Pass
    Render(1);  // Light-1-Pass

    g_pDevice->UseShaders(false);
    g_pDevice->DrawText(g_nFontID, 10, 10, 255, 255, 0,
                       "Per Pixel Lighting Demo");
}
```

```
g_pDevice->EndRendering();
return hr;
} // Tick
```

*Lichtfarbe und
Position
verändern*

Unsere drei trigonometrischen Freunde helfen uns auch hier dabei, die Position und die Farbe der Lichtquellen zu verändern. Na gut, es sind nur zwei der drei Freunde, aber die reichen aus. Interessant sind dann die verschiedenen Render-Passes. Die Funktion `RenderLight()` zeige ich hier nicht. Sie dient nur dazu, einen kleinen Würfel an der Position der Lichtquelle zu rendern, und zwar mit dem normalen Basic-Shader-Paar. Das brauchen wir nur, um besser zu sehen, von wo ein Omni-Light gerade leuchtet. Aber unsere Render-Methode ist hier sehr interessant, weil dort alles Relevante passiert, was nun neu ist.



```
HRESULT Render(int n) {
    ZFXMatrix mat, matA;
    mat.RotaY(-0.4f);
    mat._42 -= 0.5f;
    mat._41 -= 1.5f;

    g_pDevice->SetAmbientLight(0.5f, 0.5f, 0.5f);
    g_pDevice->SetWorldTransform(&mat);

    // Ambierter Pass
    if (n<0) {
        g_pDevice->ActivateVShader(g_Base[0], VID_TV);
        g_pDevice->ActivatePShader(g_Base[1]);
    }

    // Additiver Pass pro Omni-Light
    else {
        matA = CalcTransAttenNoRot( g_vcL[n], 6.0f );

        g_pDevice->SetShaderConstant(SHT_VERTEX,
                                    DAT_FLOAT, 20,
                                    4, (void*)&matA);

        g_pDevice->SetShaderConstant(SHT_PIXEL,
                                    DAT_FLOAT, 0, 1,
                                    (void*)&g_clrL[n]);

        g_pDevice->ActivateVShader(g_Omni[0], VID_TV);
        g_pDevice->ActivatePShader(g_Omni[1]);
        g_pDevice->UseAdditiveBlending(true);
    }

    HRESULT hr = g_pDevice->GetVertexManager()->Render(
                                                g_sRoom);

    g_pDevice->UseAdditiveBlending(false);
    return hr;
} // Render
```

Der ambiente Pass ist nicht weiter interessant. Wir müssen nur den richtigen Shader einstellen und dann rendern. Wenn wir aber einen Parameter der Funktion erhalten, der auf einen Pass für ein Omni-Light hinweist, dann berechnen wir die Matrix für dieses Omni-Light anhand seiner Position und des Radius, den ich hier auf 6.0f festgelegt habe. Diese Matrix schieben wir dann in das entsprechende Konstanten-Register für den Vertex-Shader. Die Farbe des Lichts kommt in das Konstanten-Register c0 des Pixel-Shaders, und dann aktivieren wir die entsprechenden Shader für diesen Pass und natürlich auch das additive Rendering, so dass die zusätzlichen Passes jeweils zu dem bereits vorhandenen Bild im Back-Buffer addiert werden und dort nichts überschreiben.

Rendern des Raums

Das war auch schon alles, was man über das Rendern beliebig vieler Omni-Lights auf einer Per-Pixel-Basis wissen muss.

Demo 5: Graustufenfilter

Pixel-Shader kann man nicht nur dazu verwenden, Beleuchtungseffekte umzusetzen, also per Pixel-Lighting. Man kann natürlich jede Art von Farbmanipulation an einem Pixel durchführen, die einem in den Sinn kommt und die man durch mathematischen Formeln ausdrücken kann. Eine solche Formel ist die folgende:

Schnell erledigt

```
NewColor = { 0.30*OldColor.R,
              0.59*OldColor.G,
              0.11*OldColor.B };
```

Man nimmt hier einen Farbwert und skaliert die einzelnen Farbanteile für Rot, Grün und Blau mit den oben gezeigten Faktoren. Die daraus resultierende Farbe ist der vorherige Farbwert, der zu einem Grauton umgerechnet wurde. Wenn man diese Formel in seinen Pixel-Shader einbaut, um den endgültigen Farboutput zu modifizieren, dann erhält man einen Filter, der ein Graustufen-Bild erzeugt. Für unseren Basic Pixel-Shader mit Multitexturing sieht das dann so aus:

```
ps.1.1
def c1, 0.30, 0.59, 0.11, 1.0
tex t0
mul r1, t0, c3
mad r0, t0, v0, r1
dp3 r0, r0, c1
```

Die Instruktion `def` dient dazu, einen vierdimensionalen Vektor für ein Konstanten-Register im Shader zu erzeugen. Die Faktoren für die Graustufen tun wir also in dieses Register und multiplizieren den Farbwert des Pixels am Ende mit diesen Werten. Damit erhalten wir den Pixel in der korrekten Graustufe. Um diesen Abschnitt nicht ganz so kurz zu halten, habe

Definitionen im Shader

ich auch das Konstanten-Register `c3` hier mit eingebaut. Dorthin speichert unsere Engine ja den Wert der Emissive Reflection aus dem Material der Skin. Wenn ihr nun den entsprechenden Wert beim Erzeugen der Skin ändert, so werdet ihr sehen, welchen Einfluss das auf die Beleuchtung hat. Der Raum wird dann immer heller, weil seine Wände selbst hell leuchten, auch wenn sie nicht von Licht beschienen werden.

Als Beispiel habe ich die Applikation mit den Omni-Lights verwendet und dort in beide Pixel-Shader den Graufilter eingebaut.

Demo 6: Bump-Mapping

Mehr Per-Pixel-Lighting

Durch die Omni-Lights haben wir eine Möglichkeit gesehen, wie wir unsere flache Geometrie sehr detailliert beleuchten können, ohne zusätzliche Hilfs-vertices einzufügen. Diffuse Texturen haben ja insbesondere die Aufgabe, einen hohen Detaillierungsgrad eines Objekts vorzutäuschen. Wenn der Betrachter jedoch die Silhouette oder eine Kante des Objekts sieht, so wird er erkennen, dass die Textur nur ein Fake ist. Als Beispiel sei hier ein Bücherregal genannt, das man als Würfel modelliert, auf dem man vorn eine Textur legt, die ein paar Buchrücken zeigt, ohne die Bücher als Geometrie zu modellieren. Doch nicht nur von der Seite gesehen, fällt dieser Schwindel auf. Wenn man das Objekt beleuchtet, wird ein flaches Polygon auch mit entsprechend gleicher Intensität beleuchtet, es gibt keine Schattierungen und Abstufungen, die ein detailliert modelliertes Objekt haben sollte. Auch dort versagen diffuse Texturen in ihrer Täuschungswirkung. Mit den oben gezeigten Detailmap versucht man, dem Problem ein wenig zu Leibe zu rücken. Das Problem dabei ist jedoch, dass diese Detailmaps statisch sind. Wenn sich die Lichtquelle bewegt, bleibt die Abstufung in der Beleuchtung auf dem Objekt dennoch gleich, weil diese Daten ja aus einer statischen Textur genommen werden.

Bump-Mapping

Wir brauchen also eine bessere Möglichkeit, Per-Pixel-Lighting durchzuführen, ohne dabei jedoch mehr Geometrie erzeugen zu müssen. Eine solche Möglichkeit gibt es natürlich. Oder besser gesagt: Es gibt einige Möglichkeiten. Eine der berühmtesten dürfte wohl das Bump-Mapping sein, das sogar alle modernen Grafikkarten in Hardware unterstützen. Aber wir wollen ja hier auf die Fixed-Function-Pipeline verzichten und verwenden daher eine eigene Implementierung in Shadern. Doch zurück zum Bump-Mapping an sich. Die Idee dabei ist total simpel. Wenn wir eine Beleuchtung per Vertex durchführen, dann funktioniert das nur, weil wir pro Vertex einen Normalenvektor haben und die dort berechneten Farbwerte zwischen den Vertices eines Triangle interpolieren. Nun wollen wir eine Per-Pixel-Beleuchtung durchführen. So formuliert liegt die Antwort sehr nah: Wir brauchen einen Normalenvektor pro Pixel!

Doch wo bekommen wir den her? Bei den Vertices können wir die Normalen in der Vertex-Struktur speichern. Doch für Pixel haben wir so etwas natürlich nicht. Dazu gesellt sich auch noch die Frage, wie wir überhaupt den Normalenvektor eines Pixels festlegen sollen, also welche Ausrichtung er hat. Kommen wir gleich zur Lösung des Rätsels, und werfen wir dazu einen Blick auf Abbildung 7.11. Dort ist eine diffuse Textur zu sehen, die wir für die Bump-Mapping-Demo verwenden werden und darunter eine Heightmap. Diese zeigt quasi eine Draufsicht auf eine Oberfläche aus der Vogelperspektive. Je heller ein Pixel dort ist, desto höher liegt er über Grund. Die schwarzen Bereiche sind entsprechend die niedrigsten Punkte der von oben betrachteten Fläche. Die Heightmap in der Abbildung 7.11 kann man sich hier auch als ein Schwarzweiß-Foto eines Kopfsteinpflasters von oben vorstellen.

Pixel-Normalen-vektor

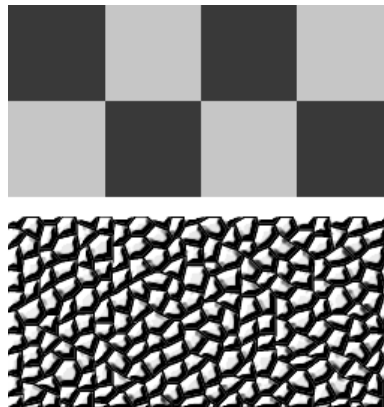


Abbildung 7.11:
Oben die diffuse
Textur und unten
eine Heightmap
Textur

Wozu dient die Heightmap? Wir können basierend auf den verschiedenen Höheninformation der Heightmap zwischen benachbarten Pixeln eine Steigung aufgrund ihres Höhenunterschiedes berechnen. Wir werden sogar noch weiter gehen und aus jeweils drei Pixeln ein imaginäres Dreieck aufspannen. Zu diesem Dreieck können wir dann in der Tat einen Normalenvektor berechnen. Die Höhe und Breite der Textur kann man als x- und y-Koordinate der Pixel auffassen und die Helligkeit eines Pixels als seinen Wert auf der y-Achse. Wir können also jeden Pixel der Heightmap als einen Vertex betrachten. Und aus drei Vertices können wir doch wohl zwei Kanten eines Dreiecks bilden und mittels des Kreuzprodukts den Normalenvektor erstellen, nicht wahr?

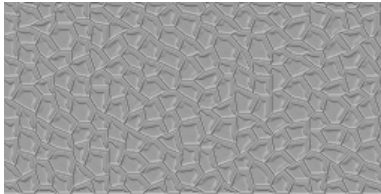
*Sinn der
Heightmap*

Genau das tun wir gleich auch. Den so errechneten Normalenvektor speichern wir dann auch wieder in der Textur ab und überschreiben damit die Farbinformation eines Pixels. Die RGB-Werte eines Pixels entsprechen nun nicht mehr einer Farbe, sondern den XYZ-Koordinaten eines Vektors. Im Pixel-Shader können wir dann auf die Textur zugreifen, den entsprechenden Normalenvektor eines Pixels auslesen und mit diesem dann die Intensität

Normalmap

der Beleuchtung auf diesem Pixel genau bestimmen. Diese Textur ist nun keine Heightmap mehr, sondern eine so genannte *Normalmap*. Abbildung 7.12 zeigt die Normalmap der oben dargestellten Heightmap. Aus der Abbildung kann man nicht wirklich etwas ansehen, nur dass sie anders aussieht als die Heightmap. Im Original ist sie in Lila-blau-Tönen gehalten.

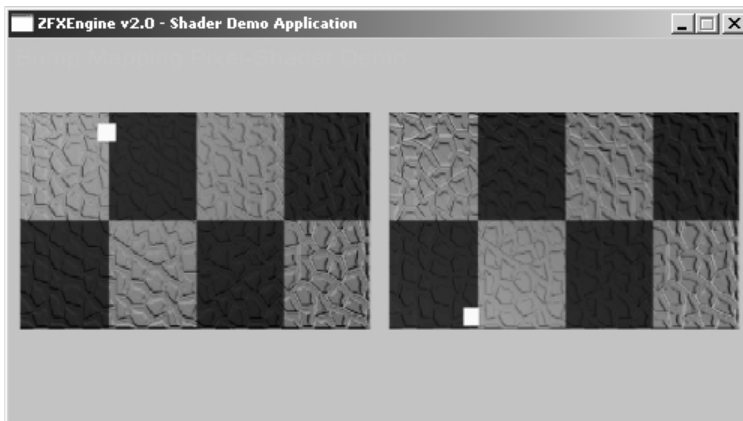
Abbildung 7.12:
Aus der Heightmap
berechnete
Normalmap



**Optischer
Eindruck**

Kommen wir nun zu der Wirkung, die das Ganze hat. Zwei Bilder sagen da mehr als tausend Worte, daher die Abbildung 7.13. Dort erkennt man zwei Rechtecke mit der diffusen Textur. Nun sieht es zunächst so aus, als ob wir dort durch Multitexturing eine Detailmap aufgelegt hätten. Dem ist aber nicht so. Auf den zweiten Blick erkennt man, dass vor dem linken Rechteck links oben ein Omni-Light schwebt, und vor dem rechten Rechteck ist ein Omni-Light links unten. Wenn man sich nun noch einmal genau die vermeintliche Detailmap anschaut, so wird man erkennen, dass es dort tatsächlich einen Schattenwurf in Relation zur Lichtquelle gibt. Wir verwenden hier nämlich keine statische Detailmap, sondern die oben erklärte Normalmap und berechnen die Helligkeit jedes Pixels basierend auf seiner Normalen und deren Winkel zum einfallenden Licht. Dementsprechend erkennt man die Struktur der ursprünglichen Heightmap sehr gut wieder, und durch den unterschiedlichen Lichteinfall hat man den Eindruck, dass das Kopfsteinpflaster wirklich als 3D-Mesh vorhanden ist. Schließlich wirft es ja in Echtzeit einen korrekten Schatten. Und genau das ist einfaches Bump-Mapping mit einer Normalmap.

Abbildung 7.13:
Auswirkung der
Bump-Map. Auf der
linken Seite ist die
Lichtquelle oben
links, auf der rechten
Seite ist sie
unten links. Man
beachte den ent-
sprechenden Schat-
tenwurf auf dem
flachen Rechteck.



Wenn das nicht Lust auf mehr macht, dann weiß ich auch nicht, was noch. Wir werden im Folgenden insbesondere auch zwei Dinge implementieren, die andere Beispiele aus dem Internet gern überspringen oder wozu sie Hilfsmethoden aus der D3DX-Bibliothek verwenden. Wir werden hier direkt aus der Heightmap die Normalmap zu Fuß berechnen, und wir werden auch die Basisvektoren des Texturkoordinatenraums für jeden Vertex berechnen. Wenn man nämlich die Normalmap bereits gestellt bekommt und die eben erwähnten Vektoren durch die D3DX-Funktionen berechnet, dann ist Bump-Mapping in der Tat nichts anderes als ein einfaches Punktprodukt zwischen zwei Vektoren. Aber wir sind ja hier, um etwas zu lernen. Also los!

*Saubere
Handarbeit*

Heightmap zu Normalmap konvertieren

Der eine oder andere hat vielleicht schon den Code des Skin-Managers genau untersucht und dort ein paar neue Methoden gefunden, die in diesem Buch bisher nicht erwähnt wurden. Da wir das Laden von Grafiken als Texturen komplett über den Skin-Manager abwickeln, gehört auch das Konvertieren einer Grafik, die eine Heightmap darstellt, zu einer Normalmap in den Aufgabenbereich des Skin-Managers. In dessen Interface finden wir die folgende Methode:

*Ergänzungen im
Skin-Manager*

```
virtual HRESULT AddTextureHeightmapAsBump(  
    UINT nSkinID,  
    const char *chName)=0;
```

Diese Methode ist nahezu analog zu ZFXSkinManager::AddTexture implementiert, lediglich alle Funktionalitäten, die die Alpha-Werte der Textur betreffen, sind dort entfernt worden. Trotzdem müssen wir das Direct3D-Textur-Objekt im 32-Bit-Format D3DFMT_A8R8G8B8 anlegen. Diese Methode fügt auch der angegebenen Skin eine Textur für die erste noch freie Stage hinzu. Der große Unterschied zwischen den Methoden ist aber, dass diese Methode hier nicht ZFX_OK zurückgibt, sondern den Aufruf der Methode ZFXSkinManager::ConvertToNormalmap.

Nachdem wir die Heightmap als Textur geladen haben, müssen wir sie nur noch zu einer Normalmap konvertieren. Dazu verriegeln wir die Textur, durchlaufen alle Pixel und speichern für jeden Pixel einen Normalenvektor, dessen Koordinaten zu einem DWORD konvertiert worden sind. Dann entriegeln wir die Textur und sind fertig. So weit, so gut, doch wie berechnen wir die Normalen an sich? Ich hatte ja bereits gesagt, dass wir in einer Heightmap jeden Pixel als dreidimensionalen Punkt auffassen können, der wie folgt aussieht:

*Normalmap aus
Heightmap
erzeugen*

```
ZFXVector vcPoint( x, y, Farbe );
```

Mit x und y ist hier die Position des Pixels in der Grafikdatei gemeint. Für die dritte Komponente des Vektors nehmen wir den Höhen-Wert aus der Heightmap. Je heller die Farbe ist, desto größer ist der Wert. Wenn wir uns

nun einen Pixel als Punkt v_0 betrachten, dann nehmen wir den Pixel, der rechts neben ihm in der Datei liegt, und denjenigen, der unter dem Pixel rechts neben ihm liegt. Damit haben wir die drei Punkte v_0 , v_1 und v_2 im Raum, aus denen wir die beiden Kanten v_1-v_0 und v_2-v_0 bilden können. Diese beiden Kanten, oder besser gesagt Vektoren, verwenden wir dazu, das Punktprodukt zu berechnen, um so einen Vektor zu erhalten, der orthogonal zu der Ebene steht, in der das Dreieck (v_0 , v_1 , v_2) liegt. Und das ist genau der Normalenvektor, den wir für den Pixel suchen. Das ist wirklich so einfach, wie es hier klingt.

Perturbed Normals

Man beachte, dass wir den veränderlichen Wert der Höhe aus der Heightmap, der ausschlaggebend für die Ausrichtung der Normalen sein wird, in der z-Koordinate speichern. Eine ungestörte Normale (engl. unperturbed normal) ist also im Tangent-Space ein Vektor der Form (0,0,1), der entlang der positiven Z-Achse läuft. Jeder andere Vektor im Tangent-Space ist eine gestörte Normale (engl. perturbed normal) und führt zu einer Lichtintensität, die von der der Vertex-Normalen abweichend ist.



```
HRESULT ZFXD3DSkinManager::ConvertToNormalMap(
    ZFXTEXTURE *pTexture) {
    HRESULT hr=ZFX_OK;
    D3DLOCKED_RECT d3dRect;
    D3DSURFACE_DESC desc;
    LPDIRECT3DTEXTURE9 pTex = ((LPDIRECT3DTEXTURE9)
                                pTexture->pData);
    pTex->GetLevelDesc(0, &desc);

    if (FAILED(pTex->LockRect(0, &d3dRect, NULL, 0)))
        return ZFX_BUFFERLOCK;

    // Pointer auf die Pixeldaten
    DWORD* pPixel = (DWORD*)d3dRect.pBits;

    // erzeuge den Normalenvektor für jeden Pixel
    for (DWORD j=0; j<desc.Height; j++) {
        for (DWORD i=0; i<desc.Width; i++) {
            DWORD color00 = pPixel[0];
            DWORD color10 = pPixel[1];
            DWORD color01 = pPixel[d3dRect.Pitch
                                   / sizeof(DWORD)];

            // benutze nur den R-Anteil aus ARGB,
            // shifte im 32-Bit-DWORD ganz nach rechts,
            // um ihn als Wert 0-255 zu erhalten, und
            // skaliere mit 1/255 auf Werte von 0.0-1.0
            float fHeight00 = (float)((color00&0x00ff0000)
                                       >>16) / 255.0f;
            float fHeight10 = (float)((color10&0x00ff0000)
                                       >>16) / 255.0f;
```

```

float fHeight01 = (float)( (color01&0x00ff0000)
                          >>16 ) / 255.0f;

// erstelle die Kanten
ZFXVector vcPoint00(i+0.0f,j+0.0f,fHeight00);
ZFXVector vcPoint10(i+1.0f,j+0.0f,fHeight10);
ZFXVector vcPoint01(i+0.0f,j+1.0f,fHeight01);
ZFXVector vc10 = vcPoint10 - vcPoint00;
ZFXVector vc01 = vcPoint01 - vcPoint00;

// berechne die Normale
ZFXVector vcNormal;
vcNormal.Cross(vc10, vc01);
vcNormal.Normalize();

// speichere die Normale als RGB-Wert
*pPixel++ = VectortoRGBA(&vcNormal, fHeight00);
}
}
pTex->UnlockRect(0);
return ZFX_OK;
} // ConvertToNormalmap

```

Der Quelltext ist ja eigentlich hinreichend kommentiert. Interessant ist es hierbei vielleicht noch, dass wir nur den Rot-Anteil des Pixels verwenden. Eine Heightmap ist immer in Grautönen gehalten, wobei alle drei Farbwerte eines Pixels relativ ähnliche Werte haben sollten, idealerweise sogar die gleichen Werte. Der Pixel liegt in der Bit-Form 0xAARRGGBB vor, und durch die Bitmaske 0x00ff0000 schmeißen wir mittels einer Bitweisen UND-Verknüpfung alles aus dem Pixel raus, was nicht zu den acht Bits für Rot an der Position Bit 16 bis Bit 23 (von rechts nach links gesehen) liegt und shiften die verbleibenden Bits um 16 Stellen nach rechts, also auf die Positionen der Bits 0 bis 7. Damit haben wir den Pixel als Wert im Bereich von 0 bis 255, wobei 255 Weiß entsprechen würde und der maximale Höhenwert ist. Jetzt teilen wir noch durch 255, um den Wert so auf einen Bereich zwischen 0.0f und 1.0f zu skalieren. Mittels der x- und y-Position des Pixels und der errechneten Höhe können wir nun, wie oben besprochen, den Normalenvektor berechnen.

Bitshifting

Nun müssen wir diesen Normalenvektor auch irgendwie speichern. Dazu codieren wir ihn auch als DWORD und speichern ihn an der Stelle des eben behandelten Pixels ab. Da wir den Normalenvektor normalisiert haben, liegen die einzelnen Werte des Vektors alle im Bereich zwischen -1.0f und +1.0f. Diese Werte skalieren wir nun auf den Bereich von 0 bis 255 und speichern die Werte für die jeweiligen RGB-Bits ab. Dazu müssen wir die Werte einfach, wie wir bereits weiter oben gesehen haben, mit der Hälfte des maximalen Wertes multiplizieren und diese dann nochmals addieren. In der A-Komponente des Pixels speichern wir einfach die entsprechend skalierte Höhe, die immer ein positiver Wert zwischen 0.0f und 1.0f ist.

Speicherung der Normalen



```

DWORD VectortoRGBA(ZFXVector *vc, float fHeight) {
    DWORD r = (DWORD)( 127.0f * vc->x + 128.0f );
    DWORD g = (DWORD)( 127.0f * vc->y + 128.0f );
    DWORD b = (DWORD)( 127.0f * vc->z + 128.0f );
    DWORD a = (DWORD)( 255.0f * fHeight );
    return ( (a<<24) + (r<<16) + (g<<8) + (b<<0) );
} // VectortoRGBA

```

Fertig Das ist schon die ganze Arbeit, die wir leisten müssen, um aus einer Heightmap eine Normalmap zu berechnen; eigentlich eine triviale Aufgabe. Ich habe in der Engine auch noch einen D3DX-Aufruf ergänzt, der die jeweils letzte erzeugte Normalmap als `normal.bmp`-Datei im Verzeichnis der Anwendung speichert. Das dient nur dazu, dass ihr euch eine solche Datei mal in einem Grafikprogramm anschauen könnt.

Tangent-Space aufmachen

Mythen und Wahrheiten

Um den berühmigten Tangent-Space ranken sich etliche Mythen und Sagen, und sowohl im Internet als auch in diversen Büchern findet man eine extraterrestrische Herleitung zur Bedeutung und vor allem für die Berechnung des Tangent-Space bzw. der drei Basisvektoren, durch die er aufgespannt wird. Als ich aber letztens durch das Internet surfte, fand ich eine Seite, die es tatsächlich schaffte, das Ganze auf einen kurzen knappen Satz zu bringen, den ein normalsterblicher Mensch auch verstehen kann.¹ Doch bevor wir dazu kommen, schicke ich vorweg, wozu wir den Tangent-Space überhaupt brauchen. Weiter oben haben wir beispielsweise eine Lichtberechnung durchgeführt, indem wir den Vektor des einfallenden Lichts aus dem Weltkoordinatensystem zuerst in den Object-Space transformiert haben, also in das lokale Koordinatensystem der Dreiecke, die beleuchtet werden sollten. Das ist nötig, weil wir natürlich nur innerhalb desselben Koordinatensystems sinnvolle Berechnungen zwischen Vektoren durchführen können. Nun haben wir also einen Vektor im Object-Space, doch wir haben unsere Normalenvektoren nicht mehr per Vertex, also auch im Object-Space, sondern per Pixel der Normalmap im so genannten ... na? Genau, im *Tangent-Space*.

Transformieren und immer und immer wieder

Wir könnten nun die Normalenvektoren aus dem Tangent-Space in das lokale System des Modells transformieren, aber das wäre dann eine Per-Pixel-Berechnung und wir müssten entsprechend viel davon durchführen. Besser ist es, den Vektor des einfallenden Lichts noch mal zu transformieren, nämlich aus dem Object-Space in den Tangent-Space. Diese Aufgabe können wir nur bewerkstelligen, wenn wir die drei Basisvektoren haben, die den Tangent-Space aufspannen. Unsere Aufgabe ist es also nun, diese Vektoren für jeden Vertex zu berechnen. Abbildung 7.14 zeigt das Bild eines Dreiecks und die Basisvektoren für den Tangent-Space. Der Vektor U wird als *Tangent-Vektor* bezeichnet, der Vektor V als *Binormale* und der Vektor $U \times V$, also das Kreuzpro-

¹ http://tfpsly.planet-d.net/english/3d/pplight_bump.html; eine sehr gute Seite über das Bump-Mapping und die Berechnung des Tangent-Space

dukt von U und V , bildet die so genannte *Tangent-Normale*. Man beachte, dass jeder Vertex seinen eigenen Tangent-Space hat. Der Name rührt daher, dass die Vektoren U und V jeweils eine Tangente zu einem Vertex bilden. Sie liegen nicht in der Ebene des Dreiecks, zu dem der Vertex gehört.

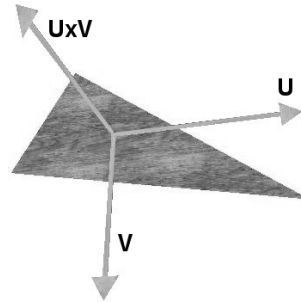


Abbildung 7.14: Die drei Basisvektoren des Tangent-Space. Der Vektor U ist der Tangent-Vektor, V ist die Binormale, und UxV ist die Tangent-Normale als Kreuzprodukt von U und V .

Können wir endlich anfangen? Gut, Ärmel hoch und los. Als Erstes brauchen wir eine neue Struktur, in der wir die Ergebnisse unserer Berechnung speichern. Unsere Engine soll ja möglichst unabhängig von Vertex-Formaten bleiben, daher können wir nicht direkt mit einer Vertex-Struktur aus der Engine arbeiten. Wir definieren die folgende Struktur für einen Vertex, zu dem wir die Basisvektoren des Tangent-Space berechnen wollen.

Ein bisschen Struktur

```
typedef struct TANGENTVERTEX_TYPE {
    ZFXVector vcPos; // Vertex-Position
    ZFXVector vcN;   // Vertex-Normale
    float      fu;   // Vertex-u-Koordinate
    float      fv;   // Vertex-v-Koordinate
    ZFXVector vcU;   // neuer Tangent-Vektor
    ZFXVector vcV;   // neuer Binormal-Vektor
    ZFXVector vcUxV; // neue Tangent-Normale
} TANGENTVERTEX;
```



Die letzten drei Vektoren sind die Outputs unserer Berechnung; die anderen Felder sind die Inputs, die ein Vertex liefern muss. Wenn man also ein Modell hat, zu dem man den Tangent-Space berechnen möchte, dann muss man alle Vertices, des Modells durchlaufen und jeweils drei Vertices die ein Dreieck bilden, in Variablen vom Typ `TANGENTVERTEX` ablegen und mit diesen dreien dann die gesuchten Basisvektoren berechnen. Drei Vertices eines Dreiecks müssen es sein, damit man die Laufrichtung der u - und der v -Koordinaten über das Dreieck bestimmen kann. Ist die Berechnung erfolgt, so speichert man die drei Basisvektoren, oder das, was man von ihnen benötigt, in die entsprechenden drei Vertices um, aus denen man den Input gezogen hat und nimmt sich das nächste Dreieck vor.

Drei Vertices spannen einen Raum auf

Sagte ich da eben, dass wir nur diejenigen Basisvektoren umspeichern, die wir wirklich brauchen? Brauchen wir denn nicht alle? Nein, sonst würde ich ja nicht noch einmal nachhaken. Es ist so, dass die Tangent-Normale i.d.R.

Aus drei mach einen

sehr wenig von der Normalen des Dreiecks abweicht. Wenn man anstelle der Tangent-Normalen $U \times V$ also den Normalenvektor N eines Triangles verwendet, so wird man schlimmstenfalls minimale und kaum merkliche Abweichungen haben. Dafür spart man sich je Vertex, den man auf der Grafikkarte parkt oder über den Bus schickt, zwei Vektoren. Zwei spart man sich deshalb, weil man natürlich den dritten Basisvektor des Tangent-Space über ein Kreuzprodukt im Shader wiederherstellen kann, wenn man den ersten Basisvektor und die Normale des Triangles hat. Ich zeige hier zwar, wie man alle drei Vektoren berechnet, aber wir werden von dem Output nur den Tangent-Vektor speichern. Als Tangent-Normale nehmen wir später die Normale des Vertex zu Hilfe, die bei uns der Normalen des Triangles entspricht, und berechnen daraus mit dem ebenfalls im Vertex gespeicherten Tangent-Vektor die noch fehlende Binormale. Die drei so erstellen Basisvektoren mögen nicht 100%ig genau sein, aber wir sparen so satte zwei Vektoren je Vertex an Speicher und Bandbreite.

*Was sind nun
Tangente,
Binormale und
Tangent-Normale
für Vektoren?*

Um wieder auf die Einleitung zu diesem Abschnitt zurückzukommen, sollten wir uns nun fragen, was denn diese Tangent-Vektoren, Binormalen und Tangent-Normalen eines Vertex für Vektoren sind bzw. wonach sie sich richten. In diversen Büchern findet man dazu Beschreibungen wie: »*Die partiellen Ableitungen von u und v werden in Relation zu X , Y und Z im Weltkoordinatensystem berechnet.*«
Erinnern wir uns ein wenig an unseren letzten Mathematik-Kurs zurück, und uns fällt spontan wieder ein, dass die Ableitung die Änderung der Funktion über einem fest definierten Intervall beschreibt. Wir können also dem Mathe-Prof den Rücken kehren und ganz keck sagen, dass der Tangent-Vektor U in die Richtung zeigt, in der sich die u -Texturkoordinate entlang des Dreiecks in positiver Richtung verändert. Die Binormale V beschreibt analog die Richtung, in der sich die v -Texturkoordinate auf dem Dreieck in positiver Richtung ändert. Die Tangent-Normale ist dann ganz einfach der zu den beiden orthogonal stehende Vektor. Konzentrieren wir uns also auf U und V , da wir $U \times V$ ganz einfach berechnen können. Weiter oben haben wir schon etwas von partiellen Ableitungen gehört. Was leiten wir denn da ab? U und V sind wie folgt definiert:

$$U = [du/dx, du/dy, du/dz]$$

$$V = [dv/dx, dv/dy, dv/dz]$$

Man bezeichnet U und V daher auch als Gradienten der Textur, wobei gilt, dass ein Gradient in seinen einzelnen Komponenten die partiellen Ableitungen einer Funktion oder eines Gleichungssystems enthält. Von der Schule her kennt man sicherlich nur die Funktionen mit einer Variablen der Form $f(x)$, deren Ableitung $f'(x)$ auch als df/dx geschrieben wird. Dabei sagt die zweite Darstellung eindeutig aus, dass durch die Ableitung die Veränderung von f in Abhängigkeit von einer Veränderung von x beschrieben wird. Wenn man nun den Fall einer Funktion mit n Veränderlichen im $\mathbb{R}^n$ hat, so kann man nur partielle Ableitungen bilden. Dabei beschreibt jede der n partiellen Ableitungen

gen jeweils die Veränderung der Funktion in Abhängigkeit von der gerade betrachteten Veränderlichen:

$f(x_1, \dots, x_n)$ mit den partiellen Ableitungen:

$$f_{x_1} = df/dx_1$$

...

$$f_{x_n} = df/dx_n$$

Der Gradient einer Funktion im $\mathbb{R}^n$ ist also ein Vektorfeld der partiellen Ableitungen. Betrachten wir nun die Gradienten U und V , so sehen wir, dass sie jeweils die Veränderung der Texturkoordinaten u bzw. v in Abhängigkeit von der Veränderung der Variablen x , y und z beschreiben.

Das machen wir mit der folgenden Funktion:

```
void CalcTangentSpace(TANGENTVERTEX *tv1,
                    TANGENTVERTEX *tv2,
                    TANGENTVERTEX *tv3) {

    ZFXVector vc, vcA, vcB;

    float fu21 = tv2->fu - tv1->fu,
          fv21 = tv2->fv - tv1->fv,
          fu31 = tv3->fu - tv1->fu,
          fv31 = tv3->fv - tv1->fv;

    vcA.Set(tv2->vcPos.x - tv1->vcPos.x, fu21, fv21);
    vcB.Set(tv3->vcPos.x - tv1->vcPos.x, fu31, fv31);
    vc.Cross(vcA, vcB);
    if (vc.x != 0.0f) {
        tv1->vcU.x = -vc.y / vc.x;
        tv1->vcV.x = -vc.z / vc.x;
    }

    vcA.Set(tv2->vcPos.y - tv1->vcPos.y, fu21, fv21);
    vcB.Set(tv3->vcPos.y - tv1->vcPos.y, fu31, fv31);
    vc.Cross(vcA, vcB);
    if (vc.x != 0.0f) {
        tv1->vcU.y = -vc.y / vc.x;
        tv1->vcV.y = -vc.z / vc.x;
    }

    vcA.Set(tv2->vcPos.z - tv1->vcPos.z, fu21, fv21);
    vcB.Set(tv3->vcPos.z - tv1->vcPos.z, fu31, fv31);
    vc.Cross(vcA, vcB);
    if (vc.x != 0.0f) {
        tv1->vcU.z = -vc.y / vc.x;
        tv1->vcV.z = -vc.z / vc.x;
    }
}
```



```

// normalisiere U- und V-Vektoren
tv1->vcU.Normalize();
tv1->vcV.Normalize();

tv2->vcU = tv3->vcU = tv1->vcU;
tv2->vcV = tv3->vcV = tv1->vcV;

// berechne die Tangent-Vektoren, und stelle sicher,
// dass sie ca. in dieselbe Richtung wie die Triangle
// Normale zeigen
tv1->vcUxV.Cross(tv1->vcU, tv1->vcV);
if ( ( tv1->vcUxV * tv1->vcN ) < 0.0f )
    tv1->vcUxV *= -1.0f;

tv2->vcUxV.Cross(tv2->vcU, tv2->vcV);
if ( ( tv2->vcUxV * tv2->vcN ) < 0.0f )
    tv2->vcUxV *= -1.0f;

tv3->vcUxV.Cross(tv3->vcU, tv3->vcV);
if ( ( tv3->vcUxV * tv3->vcN ) < 0.0f )
    tv3->vcUxV *= -1.0f;
} // CalcTangentSpace

```



Wie man hier sehen kann, berechnet man die jeweiligen Vektoren für jeden Vertex gleich, also nur auf einer Per-Triangle-Basis. Des Weiteren müssen wir auch darauf achten, dass jeder Vertex daher nur zu einem Triangle gehören sollte bzw. dass alle Triangles, zu denen er gehört, zu demselben ebenen Polygon gehören sollten. Ansonsten müsste man mehrere verschiedene Vektoren für verschiedene Tangent-Räume im Vektor speichern. Diese sind dann jeweils nur für ein bestimmtes Polygon gültig. Da wir aber im Verlauf des Buches Vertices immer nur für genau ein Polygon verwenden, ist das kein Problem.

Vertex-Shader

Es gibt viel zu tun

Endlich mal ein Vertex-Shader, bei dem ein paar mehr Zeilen herausspringen. Den Anfang des Shaders bilden jedoch wie gewohnt die Versionsnummer, die Deklarationen und die Transformation und Projektion der Vertex-Position, mit der im Vertex-Shader-Konstanten-Register c0 abgespeicherten Combo-Matrix aus der Weltmatrix, der View-Matrix und der Projektionsmatrix.

```

vs.1.1
dcl_position v0
dcl_normal v3
dcl_texcoord v7
dcl_tangent v8
m4x4 oPos, v0, c0

```


Hierbei müssen wir wieder beachten, dass wir Berechnungen zwischen Vektoren nur dann sinnvoll vornehmen können, wenn alle Vektoren im selben Koordinatenraum vorliegen – egal welcher das auch immer ist. Wir haben nun unseren Tangent-Vektor und den Normalenvektor des Vertex, den wir auch als Tangent-Normale verwenden, im lokalen System des Modells vorliegen, zu dem der Vertex gehört. Aber wir müssen beachten, dass sich das Modell aller Wahrscheinlichkeit nach auch transformiert hat, beispielsweise rotiert oder verschoben. Also müssen wir auch die beiden untransformierten Vektoren für die Tangente und die Tangent-Normale zunächst entsprechend rotieren. Dazu erwartet unser Vertex-Shader die Transponierte der Weltmatrix des Modells im Konstanten-Register c31.

```
m3x3 r5.xyz, v8, c31 ; rotiere Tangente (U)
mov r5.w, c30.w
m3x3 r7.xyz, v3, c31 ; rotiere Vertex-Normale
mov r7.w, c30.w
```

*Tangente auf die
5, Tangente bitte!*

Wie ihr seht, verwenden wir nicht die volle 4x4-Matrix, sondern nur die 3x3-Matrix aus der oberen linken Ecke. Das hat den Grund, dass wir Vektoren, die eine Ausrichtung angeben, keinesfalls verschieben dürfen und die Verschiebung in der vierten Zeile der Matrix bzw. der vierten Spalte der Transponierten gespeichert ist. Eine Verschiebung würde die Ausrichtung eines Richtungsvektors fehlerhaft beeinflussen. Die rotierten Vektoren speichern wir dann in den temporären Registern r5 und r7, wobei wir explizit .xyz angeben, um auch nur die entsprechenden Koordinaten zu modifizieren. Die mov-Instruktion kopiert dann entsprechend nur die ersten drei float-Werte aus dem Quellen-Register. In das Konstanten-Register c30 speichern wir den Vektor (0.5, 0.5, 0.5, 1.0), um diese Werte im Shader als Konstanten zur Verfügung zu haben. Wir könnten auch die def-Anweisung verwenden, aber diese würde dann bei jedem Aufruf des Shaders ausgeführt werden.

*Alles, nur keine
Verschiebung*

Nun machen wir uns daran, die Binormale zu berechnen. Den Tangent-Vektor haben wir ja aus der Vertex-Struktur, und als Normalen verwenden wir die Normale des Vertex. Den dazu orthogonalen Vektor finden wir ganz simpel über das Kreuzprodukt der beiden anderen.

*Binormale
berechnen*

```
mul r0, r5.zxyw, -r7.yzxw;
mad r6, r5.yzxw, -r7.zxyw, -r0;
```

Hier sehen wir, dass es der Shader auch anbietet, die Instruktionen nicht in der linearen Reihenfolge der vier Komponenten eines Registers auszuführen. Über die Anweisungen .zxyw und .yzxw können wir gezielt die Komponenten der Register angeben, die miteinander multipliziert werden sollen. Vergleicht man diese Multiplikationen mit der Formel des Kreuzprodukts, so wird man eine nicht zufällige Ähnlichkeit feststellen:

$$\begin{aligned} A \times B &= \begin{pmatrix} A_y B_z - A_z B_y \\ A_z B_x - A_x B_z \\ A_x B_y - A_y B_x \end{pmatrix} \end{aligned}$$

Die ominöse `mad`-Anweisung dient dazu, eine `mul`- und eine `add`-Instruktion in einer Zeile auszuführen. Dabei gibt man zuerst das Zielregister für das Ergebnis an, danach die beiden Register, deren Inhalt zunächst multipliziert wird. Als Letztes folgt dann das Register, dessen Inhalt zu dem Ergebnis der Multiplikation addiert wird.

```
mad Ziel, Faktor1, Faktor2, Summand2
```

```
⇔ Ziele = (Faktor1 * Faktor2) + Summand2
```

Nun haben wir die folgenden Daten in den temporären Registern des Shaders, und zwar jeweils im Koordinatensystem des transformierten Modells:

- ➡ r5 Tangent-Vektor
- ➡ r6 Binormale
- ➡ r7 Tangent-Normale

*Der Weg zum
Licht*

Als Nächstes müssen wir nun den Vektor berechnen, der von der Vertex-Position direkt zum Licht hinführt. Dazu speichert unsere Applikation die Position des Lichts in lokalen Koordinaten des Modells im Konstanten-Register `c25`. Nun bilden wir mit der Position des Vertex den Differenzvektor, der von dem Vertex zum Licht hinläuft, und transformieren ihn dann mit den drei Basisvektoren des Tangent-Space in denselbigen.

```
sub r2, c25, v0
dp3 r8.x, r5.xyz, r2
dp3 r8.y, r6.xyz, r2
dp3 r8.z, r7.xyz, r2
```

*Tangent-Space-
Transformations-
matrix*

Um einen Vektor in den Tangent-Space zu transformieren, erstellen wir einfach eine Rotationsmatrix aus den drei Basisvektoren des Tangent-Space und multiplizieren den Vektor zum Licht damit. Doch dieser Vektor ist nun aller Wahrscheinlichkeit nach alles andere als normalisiert, was wir aber von ihm erwarten, wenn das Punktprodukt aussagekräftig sein soll. Also normalisieren wir den Vektor zum Licht erst einmal:

```
dp3 r8.w, r8, r8
rsq r8.w, r8.w
mul r8.xyz, r8, r8.w
```

*Reziproke
Wurzeln*

Um den Vektor zu normalisieren, berechnen wir zunächst das dreidimensionale Punktprodukt ($x*x + y*y + z*z$) und speichern das Ergebnis, das das Quadrat der Länge des Vektors ist, im Feld `w` des temporären Registers. Nun ziehen wir daraus nicht die Wurzel, sondern die reziproke Wurzel, also $1/\sqrt{}$. Danach multiplizieren wir die `x`-, `y`- und `z`-Koordinate mit diesem Wert und haben somit die ersten drei Komponenten effektiv durch die Länge des Vektors geteilt. Damit ist er normalisiert.

Nun sind wir mit unseren Berechnungen fertig. Im temporären Register `r8` haben wir nun den normalisierten Vektor vom Vertex zum Licht im Tangent-Space. Das Problem ist nun aber, dass dieser im Wertebereich von -1 bis $+1$ liegt. Wenn wir den Vektor aber über ein Output-Register zum Pixel-Shader schieben, dann wird er auf den Bereich 0 bis $+1$ geclippt – ein für uns altbekanntes Problem. Also multiplizieren wir die Komponenten des Vektors mit 0.5 aus dem Konstanten-Register `c30` und addieren dann noch einmal 0.5 . So behalten wir die Information des Vektors vollkommen bei, haben ihn nur dem entsprechenden Wertebereich angepasst. Im Pixel-Shader werden wir den originalen Wertebereich wieder herstellen.

Und wieder Bias

```
mad oD0.xyz, r8.xyz, c30.x, c30.x
mov oT0.xy, v7.xy
mov oT1.xy, v7.xy
```

Wie gewohnt verschieben wir auch die Texturkoordinaten in die entsprechenden Output-Register des Vertex-Shaders. Damit ist unser Vertex-Shader für das Bump-Mapping vollendet. Nun heißt es: »Pixel-Shader, übernehmen sie ...«

Texturkoordinaten

Pixel-Shader

Der zugehörige Pixel-Shader für das Bump-Mapping ist wiederum total kurz. Warum sollte er auch lang werden? Wir haben nun alles, was wir brauchen. Wir haben den Vektor, der zum Licht hinzeigt, im Input-Register `v0` sowie den Normalenvektor des Pixels in der Textur-Stage 1. Beide sind im selben Tangent-Space definiert, also können wir einfach das Punktprodukt zwischen ihnen bilden und erhalten so den Wert für die Intensität des auf den Pixel einwirkenden Lichts unter Berücksichtigung seiner eigenen Normalen und damit vollkommen unabhängig von der Normalen der Vertices des Triangles. Mit diesem Intensitätswert müssen wir dann noch die Farbe des Lichtes aus dem Konstanten-Register `c0` multiplizieren, und diesen Wert multiplizieren wir dann mit der Farbe für diesen Pixel aus der diffusen Textur. So erhalten wir einen entsprechend der Lichtfarbe gefärbten und entsprechend dem Lichteinfall an dem konkreten Pixel abgedunkelten Farbwert als Output.

Mach's kurz!

```
ps.1.1
tex t0
tex t1
dp3 r1, t1_bx2, v0_bx2
mul r0, c0, r1
mul r0, t0, r0
```

Nun müssen wir hier, wie eben schon angedeutet, den Wertebereich des Vektors zum Licht wieder durch das Biasing (Abziehen von 0.5) und dann durch das Multiplizieren mit 2 in den ursprünglichen Wertebereich von -1

Wertebereiche

bis 1 verschieben. Für die Werte, die wir aus der Normal-Map auslesen, tun wir dasselbe, denn diese hatten wir ja auch entsprechend in den positiven Bereich verschoben und skaliert.

*Ein Wort zum
sinnvollen Einsatz
des Bump-
Mapping*

Man beachte, dass wir die diffuse Textur nun nicht mehr dazu brauchen, Details auf einem flachen Polygon vorzutauschen. Wir brauchen sie wirklich nur noch dazu, die diffuse Farbe an einer bestimmten Stelle des Triangles festzulegen. Die geometrischen Details auf dem flachen Polygon sind nun zwar immer noch nicht wirklich vorhanden, werden jetzt aber durch die Normalmap vorgetäuscht – und das auch noch wesentlich besser als durch eine entsprechende diffuse Textur. Denn nun täuschen wir auch noch eine recht realistische dynamische Beleuchtung in Abhängigkeit zu einer beliebig bewegten Lichtquelle vor. In Relation zu einer statischen Lichtquelle sieht ein statisches Polygon mit Bump-Mapping jedoch ganz genauso aus, als hätte es nur eine diffuse Textur mit vorgetäuschten Details oder eine zusätzliche Detailmap zu der diffusen Textur. Bump-Mapping kommt daher nur wirklich zur Geltung, wenn man entweder die Lichtquellen in seiner Szene bewegt oder das Objekt selbst.

TIPP

Durch die Interpolation der Pixel-Normalen aus der Normalmap verlieren die im Pixel-Shader verwendeten Normalenvektoren ihre Einheitslänge, sind also nicht mehr normalisiert. In den älteren Pixel-Shader-Versionen kann man eine Normalisierung auch nicht im Pixel-Shader durchführen. Daher verwendet man so genannte Normalizing-Cube-Maps, die Informationen enthalten, um die interpolierte Normale in einem Pixel-Shader zu normalisieren.

Demo-Applikation

Bewegtes Licht

Diesmal verwenden wir in unserer Applikation wieder ein Omni-Light. Allerdings erzeugt dies nicht über Attenuation einen Lichtkegel, sondern dient nur dazu, die Position für die Lichtquelle zu bestimmen und eine Farbe zu liefern. Das Licht rendern wir wie gewohnt als kleinen Würfel und rufen dann unsere eigentliche Render-Funktion auf.

```
HRESULT ProgramTick(void) {
    HRESULT hr = ZFX_FAIL;
    ZFXMatrix mat;
    mat.Identity();

    // bewege Licht
    float fT = GetTickCount() / 1000.0f;
    g_vcL[0].Set( 2.5f,
                  cosf( fT*2.5f) * 3.0f -0.5f,
                  1.0f );

    // verändere die Lichtfarbe
    g_clrL[0].fR = 0.5f + 0.5f * sinf(fT*2.0f);
```

LIST

```

g_clrL[0].fG = 0.5f + 0.5f * sinf(fT*2.35f);
g_clrL[0].fB = 0.5f + 0.5f * sinf(fT*2.7f);

g_pDevice->BeginRendering(true,true,true);

RenderLight(g_vcl[0].x, g_vcl[0].y, g_vcl[0].z);
Render( g_vcl[0] );

g_pDevice->UseShaders(false);
g_pDevice->DrawText(g_nFontID, 10, 10, 255, 255, 0,
                  "Bump-Mapping PS Demo");
g_pDevice->EndRendering();
return hr;
} // Tick

```

Der Render-Funktion übergeben wir nun die Position der verwendeten Lichtquelle. Diese kommt in ein Register für den Vertex-Shader, ebenso wie unser Hilfsvektor mit den Konstanten 0.5f und 1.0f (implizit für Koordinate w erstellt). Die Weltmatrix des Raumes machen wir dem Vertex-Shader auch bekannt, und das Licht verschieben wir zu den Modellkoordinaten des Raumes. Dann aktivieren wir die Shader und rendern einfach.

Rendern

```

HRESULT Render(ZFXVector vcLight) {
    ZFXMatrix mat, matInv, matT;
    ZFXVector vcHalf(0.5f,0.5f,0.5f);
    mat.Identity();
    mat._42 -= 0.5f;

    matInv.InverseOf(mat);
    vcLight = matInv * vcLight;

    g_pDevice->SetAmbientLight(0.5f, 0.5f, 0.5f);
    g_pDevice->SetWorldTransform(&mat);

    g_pDevice->SetShaderConstant(SHT_VERTEX, DAT_FLOAT,
                                25, 1, (void*)&vcLight);

    g_pDevice->SetShaderConstant(SHT_VERTEX, DAT_FLOAT,
                                30, 1, (void*)&vcHalf);

    matT.TransposeOf(mat);
    g_pDevice->SetShaderConstant(SHT_VERTEX, DAT_FLOAT,
                                31, 4, (void*)&matT);

    g_pDevice->SetShaderConstant(SHT_PIXEL, DAT_FLOAT, 0,
                                1, (void*)&g_clrL[0]);

    g_pDevice->ActivateVShader(g_Bump[0], VID_TV);
    g_pDevice->ActivatePShader(g_Bump[1]);
}

```



```
return g_pDevice->GetVertexManager()->Render(  
                                                g_sRoom);  
} // Render
```

Farbrausch Dadurch, dass wir die Farbe des Lichtes hier ziemlich berauscht in jedem Frame ändern, erhält man vielleicht keinen so schönen Eindruck von dem Bump-Mapping an sich, insbesondere bei relativ dunklen Farben des Lichts. Um das pure Bump-Mapping zu sehen, sollte man den Farbwert einfach auf weißes Licht setzen.

7.3 Ein Blick zurück, zwei Schritt nach vorn

Break In diesem Kapitel haben wir auf wenigen Seiten so einiges durchgenommen, über das man ganze Bücher schreiben kann. Natürlich sind wir nicht sehr weit in die Tiefe von Shadern gegangen, aber dazu empfehle ich jedem ein Buch, das sich ausschließlich mit Shadern beschäftigt. Auch im Internet wird man viele Tutorials und Beispiele finden. Es sollte einem auch nicht weiter schwerfallen, die sehr interessanten Beispiele, die man im Netz z.B. für Cg finden wird, in reine Assembler-Shader zu übersetzen. Mit Shadern kann man noch viel mehr interessante Dinge tun. Aber ich hoffe trotzdem, dass ich euch in diesem Kapitel einiges gezeigt habe, womit ihr bei euren Projekten etwas anfangen könnt. Insbesondere das Per-Pixel-Lighting über Omni-Lights und das Bump-Mapping sollte man verinnerlichen und diese Ansätze weiter ausbauen.

Anregungen Als Nächstes sollte man sich einige Erweiterungen zum Bump-Mapping anschauen, beispielsweise Bump-Mapping mit einem Self-Shading-Term, wie es u.a. als Sample im DirectX SDK zu finden ist. Dann sollte man Bump-Mapping mit anderen Beleuchtungsarten kombinieren, insbesondere mit den Omni-Lights. Und dann gibt es da ja auch noch das Specular Highlighting für metallische Oberflächen und Reflexionen für spiegelnde Oberflächen. Man kann so viel machen, doch wie viel man macht, das hängt letzten Endes immer von der Applikation ab, in der man das einsetzt. Ein fluffiger Shader kann auch eine sehr gute Frame-Bremse sein, insbesondere wenn es sich um aufwändige und lange Pixel-Shader handelt und man sich so exponentiell der Fill-Rate als Bottleneck annähert.

Weiter geht's Im nun folgenden Kapitel schauen wir uns an, wie man Characters, also Modelle von Menschen, Monstern und anderen Figuren animieren kann. Auch diese Animation kann man über Shader realisieren, was jedoch aus verschiedenen Gründen nicht immer sinnvoll ist. Wir werden sehen, wie man das Ganze ohne Shader macht, weil das in unserem Kontext sinnvoller sein wird. Dazu aber mehr im folgenden Kapitel. Jetzt solltet ihr euch die Demos zu diesem Kapitel ansehen und ein wenig damit herumspielen – falls ihr das nicht eh schon längst gemacht habt.

8 Skeletale Animation von Charakteren

»Der Mensch überwindet Hindernisse, um endlich Ruhe zu haben, und findet dann nichts so unerträglich wie Ruhe.« (Henry Brooks Adams 1838 – 1918)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Überblick über Animationstechniken
- ➔ File-Format laden und parsen (*.cbf)
- ➔ Technik der skeletalen Animation
- ➔ Erstellung eines CBF-Viewers

8.1 Eine Revolution?

Wie in sehr vielen Bereichen der Spiele-Entwicklung ist auch bei der Animation von Charakteren in den letzten Jahren eine Fortentwicklung der Techniken zu erkennen.

Anfänglich wurden die Animationen von Charakteren und Objekten in Texturen gespeichert, die dann wie ein einfaches Daumenkino vor dem Anwender abgespielt wurden. Noch in *Doom 1* und *Doom 2* wurde diese Technik sehr erfolgreich angewandt. Hier wurde ein Billboard genommen, also ein einfaches Quad, das immer zum Spieler ausgerichtet ist, und auf dieses wurde dann der passende Animationsframe gerendert. So entstand der Eindruck, dass die Monster und Charaktere sich wirklich bewegten. Aber letzten Endes handelte es sich dabei nur um zweidimensionale Grafiken im 3D-Raum.

*Daumenkinos aus
Sprites*

Ich kann mich noch sehr lebhaft an eine Diskussion mit Freunden erinnern, in der wir den Unterschied zwischen *Doom 1 / 2* zu *Quake* diskutierten. Heiß diskutiert wurde, ob es Sinn macht, jetzt um eine Waffe herum laufen zu können oder nicht. In *Doom 1 / 2* war die Waffe ja auch ein Billboard, das immer auf den Spieler ausgerichtet war. In *Quake* dagegen wurden erstmalig echte 3D-Objekte genutzt. Und nun ratet mal, warum diese Dinger

sich unbedingt drehen mussten. Ich denke, ihr könnt euch vorstellen, welchen Standpunkt ich hartnäckig verteidigt habe.

Ein sehr anschauliches Spiel, einer meiner Favoriten, war *Half-Life*. In diesem Spiel ist die *skeletale Animation* in voller Pracht implementiert. Abbildung 8.1 zeigt einen Screenshot dieses Spiels.

Abbildung 8.1:

Eine sehr gut erstellte Szene aus *Half-Life*. Hier versucht der Wissenschaftler, den Soldaten zu reanimieren. Die vielfältigen und flüssigen Bewegungen sind dank der skeletalen Animation möglich.



Inzwischen gibt es kein großes Spiel mehr, das nicht mit skeletalender Animation arbeitet. Uns sind Spiele wie *Half-Life 2*, *Farcry* und *Doom 3* bekannt. Sie alle nutzen die skeletale Animation.

Aus der Natur

Die Natur hatte viele Millionen Jahre Zeit, ein geeignetes »Animationssystem« für Mensch und Tiere zu entwickeln. Wie so oft im Leben sind die genialsten Dinge immer sehr einfach, und daher wurden Knochen entwickelt, die über Gelenke miteinander verbunden waren. Diese Knochen werden über Muskeln und Sehnen bewegt. Wenn wir gleich tiefer in die Technik einsteigen, dann werdet ihr deutliche Parallelen zu der natürlichen Technik erkennen. Inzwischen sind die Rechner leistungsstark genug, um uns diese Technik zu ermöglichen. Das war nicht immer der Fall. Früher wurde eher mehr Speicherverbrauch in Kauf genommen, als die CPU oder gar GPU zu belasten. Heute hat sich die Rechnerleistung vervielfacht, so dass es keinen Grund mehr gegen das Anwenden der Skeletalen-Animation gibt.

Der Siegeszug

Diese Technik hat durch Flexibilität ihren Siegeszug angetreten. In dem MD2-Format (aus dem Spiel *Quake 2*) wurde zum Beispiel jeder einzelne mögliche Frame eines Objekts oder Spielers vorher berechnet und in der Datei gespeichert. Das heißt: Wenn ein Objekt aus 650 Vertices besteht, dann werden diese 650 Vertices für jeden einzelnen Frame neu gespeichert. Das reduzierte natürlich den Rechenaufwand, erhöht aber deutlich den Speicherbedarf, und es sind nur die Animationen möglich, die vorher berechnet wurden. Auch ist keine Möglichkeit vorhanden, das einfach und effektiv an eine Physik-Engine anzubinden.

MD2-Format

Weiterhin musste für eine flüssige und weiche Animation immer zwischen den einzelnen Frames interpoliert werden. Und das bedeutet wiederum auch hier CPU-Belastung. Diese frame-basierende Technik heißt übrigens *Keyframe-Animation*.

*Keyframe-
Animation*

In Abbildung 8.2 ist auf der linken Seite die erste Position zu erkennen. Die Position aller Vertices ist gespeichert. Auf der rechten Seite der Abbildung 8.2 ist das Alien in einem anderen Frame zu sehen. Alle Vertices sind jetzt so gespeichert, dass sie sich an der neuen Position befinden. Für den nächsten Frame gilt das Gleiche, so muss also für jedes einzelne Frame immer wieder alles abgespeichert werden. Ihr könnt euch sicherlich vorstellen, was das bedeutet...

Keyframe-Animation

Wir halten also für die Keyframe-Animation Folgendes fest:

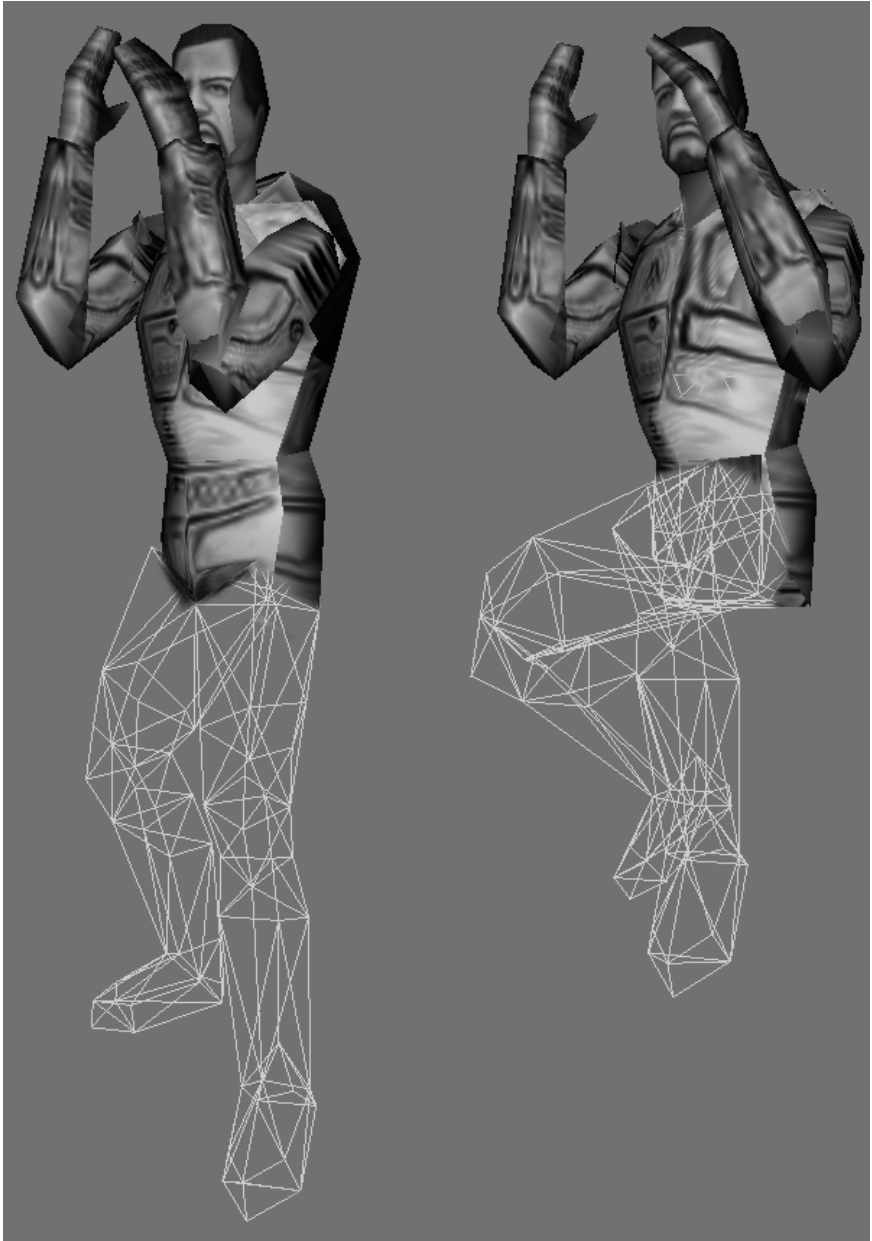
1. *Sie ist speicherintensiv.*
2. *Sie ist unflexibel; nur die vorgefertigten Animationen sind möglich.*
3. *Es gibt keine oder nur geringe Möglichkeiten zur Anbindung an eine Physik-Engine.*
4. *Sie ist jedoch einfacher zu implementieren.*

Somit ist diese Art der Animation auf den herkömmlichen PCs und Konsolen eher als antiquierte Technik anzusehen.

Für unsere Modelldaten nutzen wir ein eigenes Format. Es nennt sich *CBF*; wir werden es uns in Kürze detailliert ansehen. Es dient dazu, uns alle Informationen des Modells zur Verfügung zu stellen. Dazu gehören auch Bones (dt. Knochen), die in dem System der skeletalen Animation einfach unverzichtbar sind.

Abbildung 8.2:

Das 3D-Modell eines mittels Keyframes animierten Modells. Die beiden Seiten der Abbildung zeigen das-
selbe Modell jeweils in einem anderen Animationsframe.



***Skeletale
Animation und
Skinned-Mesh***

In unserem Rennen um die Animations-Krone heißt der Gewinner skeletale Animation. Dieses Verfahren ist auch als Skinned-Mesh bekannt. Beides ist eine Bezeichnung für dasselbe Verfahren.

Das grundlegende Konzept ist, wie oben schon erwähnt, aus der Natur abgeschaut. Schauen wir zum Beispiel einfach mal unsere Hand an. Sie ist

über ein Gelenk mit dem Unterarmknochen zum Ellenbogengelenk verbunden. Auf der anderen Seite sind viele Knochen und Gelenke, die letztendlich in unsere Finger münden. Wenn wir jetzt den Arm im Ellenbogen anwinkeln, dann bewegt sich die Hand entsprechend mit. Logisch, sagt ihr, sie ist ja auch über das Gelenk am Knochen angebracht. Und genau das ist unser skeletales Animationssystem. So einfach ist es. Abbildung 8.3 demonstriert dies am Beispiel eines Arms.

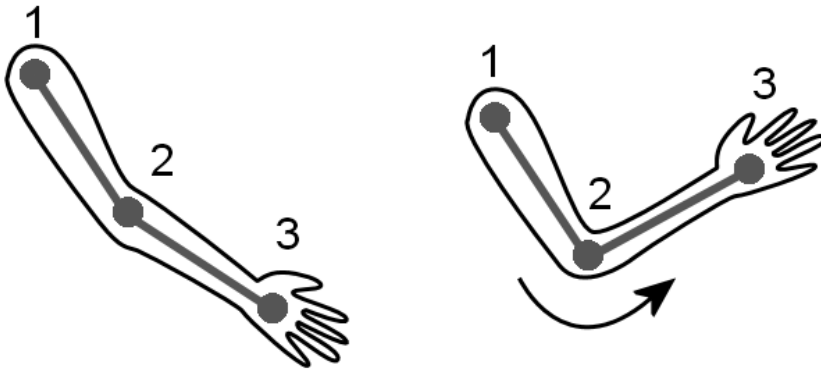


Abbildung 8.3:

Die Darstellung zeigt einen Arm mit drei Gelenken. Auf der rechten Seite sieht man den Arm in einer angewinkelten Position. Dazu wurde das Gelenk Nummer 2 rotiert, und folglich befindet sich das untergeordnete Handgelenk Nummer 3 an einer neuen Position.

Im Fachjargon nennen wir ein Gelenk *Joint*. Joints sind immer hierarchisch angeordnet. In unserem Fall ist der Rootjoint (dt. Wurzelgelenk) unsere Schulter (1). In der nächsten Ebene kommt das Ellenbogengelenk (2) und zu guter Letzt dann unser Handgelenk (3). Eine Rotation des Handgelenks wirkt sich nur auf die Hand aus. Wenn wir aber eine Ebene weiter nach unten gehen, also zu Gelenk 2, unserem Ellenbogen, dann wirkt sich die Rotation auf unseren Unterarm und die Hand aus. Als logische Konsequenz haben wir Gleiches bei der Schulter zu erwarten. Wenn wir einen Charakter hätten, der nur aus diesem Arm bestehen würde, zum Beispiel als unheimliches Objekt in einem Geisterschloss, dann würden wir beim Translieren des Gruselarms den Rootjoint (1) bewegen. Alles, was wir am Rootjoint machen, bewegen oder rotieren, wirkt sich auf das gesamte Modell aus.

Joints

Aber wo sind die Bones? Die Frage ist berechtigt. Die Antwort lautet: Es gibt keine.

Die Verbindung zwischen zwei Joints kann jedoch als Bone bezeichnet werden, und das wäre in der Natur ja auch so. In der Natur brauchen wir die kalkige Masse zwischen den Gelenken, um sie exakt an der Position zu halten, wo sie hingehören, um unseren Körper zu formen. Also trinkt mehr Milch! Da haben wir es bei der Animation einfacher. Wir sagen dem Joint (Gelenk) einfach, wo er sich befindet, und basta. Lasst uns noch eine Information aus der Natur mitnehmen. Ich sagte gerade, dass wir die Knochen brauchen, um die Gelenke zu positionieren. Das heißt, dass wir diese nur

*Bones sind
implizit vorhanden*

rotieren und nicht translieren können und dürfen. Zugegeben, natürlich *könnten* wir sie translieren, wir können sie ja auch löschen oder erst gar nicht verwenden. Aber wir sollten sie nicht translieren, weil das zu einer Verzerrung des Mesh führen würde. Wir merken uns jetzt eine Regel, die für die normale Charakter-Animation gilt:

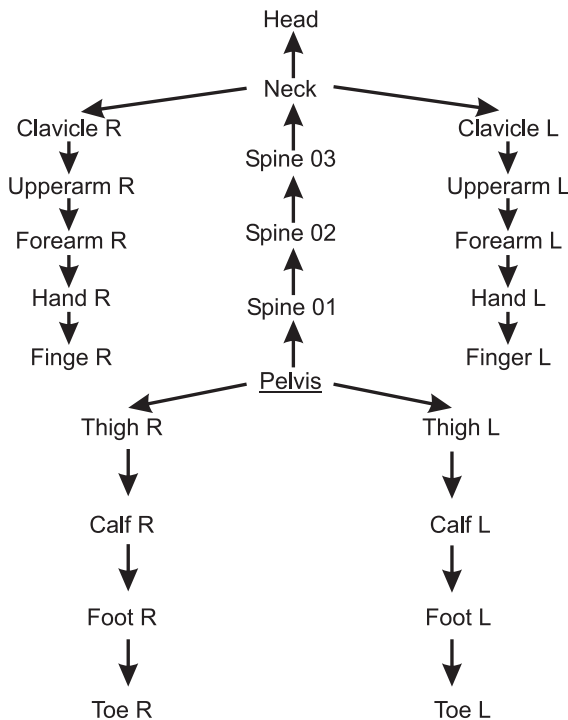
!!
STOP

Joints eines Skinned-Mesh dürfen nur rotiert und nicht transliert werden.

Ich gebe euch mein Ehrenwort: Wir können damit alle physikalisch korrekten Bewegungen mit unserem Charakter darstellen.

Wir haben gerade gelernt, dass die Joints hierarchisch angeordnet sind. Ein Joint besitzt immer nur einen Parent, kann aber viele Children haben. Nur das Rootjoint hat keinen Parent, und daran können wir es im Zweifel immer identifizieren.

Abbildung 8.4:
Die Anatomie eines Bipeds. Das Rootjoint ist das Becken (engl. Pelvis). Zur besseren Identifizierung habe ich es unterstrichen.



Auswirkung der Hierarchie

Wir sehen in Abbildung 8.4 eine Vielzahl von Joints. Es ist aber nicht immer notwendig, auch wirklich alle Joints zu nutzen. Was aber eindeutig zu erkennen ist, ist unser Rootjoint. Es hat nämlich keinen Parent. Das Becken (engl. Pelvis) besitzt nur Children. Wenn wir unseren Zweifüßler (engl. Biped) zum Beispiel komplett drehen wollen, bräuchten wir nur unseren Rootjoint zu drehen. Das überträgt sich dann auf alle anderen Joints, weil diese dem Rootjoint hierarchisch untergeordnet sind. Ist das nicht einfach?

Nun wissen wir, was Joints sind, wie sie sich auf alle Children auswirken, und auch, in welcher Weise wir sie bearbeiten müssen, um eine Animation zu bekommen. Damit unser Charakter aber auch wirklich wie ein solcher aussieht und nicht nur durch Joints dargestellt werden kann, müssen wir die Vertices mit den Joints verbinden. Dies wird in dem Modelleditor gemacht, während das Modell erstellt wird. Hier weist der Modellbuilder jeden Vertex einem bestimmten Joint zu, so dass wir beim Auslesen der Vertexinformation exakt wissen, durch welchen Joint dieser Vertex beeinflusst wird. Später beim Rendern des Modells gehen wir wie folgt vor:

1. Auslesen der originalen Vertexposition (X, Y, Z)
2. Rotieren des Vertex mit der Joint absolut Matrix
3. Speichern der neuen Information

Rendern

Auch das liest sich doch recht einfach, oder? Natürlich müssen wir hier noch ein paar Vorarbeiten leisten, der Joint muss ja von uns korrekt positioniert werden. Dazu kommen wir aber gleich.

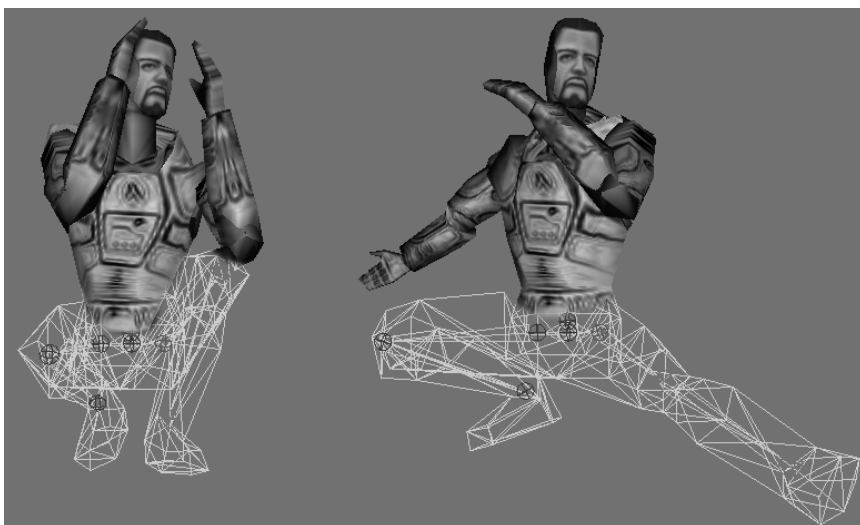


Abbildung 8.5:
Ein über Joints /
Bones animiertes
Modell in zwei ver-
schiedenen Posen

In Abbildung 8.5 können wir erkennen, dass zwar auch wieder sehr viele Vertices im Spiel sind, jedoch nur von uns die genaue Joint-Position gespeichert werden muss. Daher befindet sich in unserem Speicher unser Modell nur einmal im Urzustand. Diese Position nennt man übrigens **Bindposition**. Jede weitere Position eines Vertex wird über seine Abhängigkeit zu seinem Joint bestimmt.

Ausgangsposition

Es ist extrem wichtig, dass unser geladenes Modell sich am Anfang in der Bindposition befindet. Das heißt, dass die Joints (und damit implizit auch die Bones) sich an der Position befinden, an der sie mit den Vertices verbunden werden sollen.

Bindposition

Stellt euch vor, ihr habt einen hübschen Charakter in eurem Level stehen. Dieser besitzt zwei Arme. Die exakte Position der Vertices, die ja nun das Mesh und damit den Charakter ausmachen, sind in einer Laufposition. Wenn jetzt die Bones aber gerade bei der ersten Position in einer Sprunganimation sind, werden die gesamten Vertices mit dieser verbunden. Wie das wohl aussehen mag, wenn dann wirklich mal gelaufen wird? Das wird aber für gewöhnlich nicht der Fall sein, weil das schon im Modelleditor definiert wurde und im Regelfall die Animationssequenz 0-1 diese Bindposition ist. Also: Don't Panic.

Skeletale Animation

Wir halten also für die Skeletale Animation Folgendes fest:

1. *Sie ist speicherschonend, da keine Vertices redundant gespeichert werden.*
2. *Beliebige Animationen sind möglich, i. d. R. jedoch vordefinierte.*
3. *Die Anbindung an eine Physik-Engine ist möglich.*
4. *Sie ist jedoch etwas komplexer zu implementieren.*

Für kommerzielle Spiele ist gerade der Punkt 3 am wichtigsten. Half-Life 2 wird hier neue Maßstäbe setzen. In Unreal 2 wurde das Ragdoll-Verfahren verwandt; auch das funktioniert nur mit diesem System.

8.2 Das Modellformat CBF

Für die *ZFXEngine* haben wir ein eigenes Modellformat erstellt. Es heißt CBF, was so viel wie **Chunkbased Format** bedeutet. Zugegeben, das ist nicht sehr einfallsreich, es kommt aber wie immer auf den Inhalt und nicht auf die Verpackung an.

Kompatibel Um auch kompatibel zu anderen Formaten zu sein, habe ich einen Konverter geschrieben, der bis dato zwei Formate konvertieren kann:

1. Milkshape *.MS3D
2. 3D Studio Max *.3DS

Das Format befindet sich in der Version 1.0 und wird in Zukunft von allen ZFX-Produkten unterstützt und genutzt.

Der Loader befindet sich in der Modellklasse namens `CZFXModel`. Diese enthält alle notwendigen Informationen, um das Modell von Anfang bis Ende zu handeln. Wir müssen also in unserem Spiel einfach eine kleine Instanz

von der Klasse erstellen, ihr einen Namen zu einem Modell übergeben und dann dieses mit ein paar einfachen Aufrufen animieren. Alle notwendigen Strukturen sind über Membervariablen eingehängt. Die wichtigsten sind:

```
class CZFXModel
{
protected:
    CVERTEX      *m_pVertices;      // CAVertices
    CVERTEX      *m_pVertices_Orig; // CAVertices
    LPFACE       m_pFaces;          // Faces
    LPMESH       m_pMeshs;          // Mesh
    LPMATERIAL   m_pMaterials;      // Materialien
    LPJOINT      m_pJoints;          // Joints
    LPANIMATION  m_pAnimations;     // Animationen
    CHUNKHEAD_S  m_sHeader;         // Modelheader
    ...
}
```



Sämtlicher Quelltext zu diesem Kapitel findet sich natürlich vollständig und lauffähig auf der beiliegenden CD-ROM.



Natürlich sind noch viele weitere Membervariablen vorhanden. Die hier gezeigten sind die wichtigsten, die wir betanken werden. `CVERTEX` wird in unserem Render-Device der *ZFXEngine* deklariert. Die anderen Strukturen definieren wir im Laufe des Kapitels selbst. Bitte beachtet, dass unser Header mit der Membervariablen `m_sHeader` eine »echte« Struktur ist, im Gegensatz zu den restlichen Strukturen, die »nur« Zeiger sind.

Was ist ein Chunk?

Ein Chunk (dt. Brocken) ist einfach ein Teil in unserer Datei. Über eine ID, die **ChunkID**, können wir diesen Teil identifizieren und dann die passende Struktur einlesen.

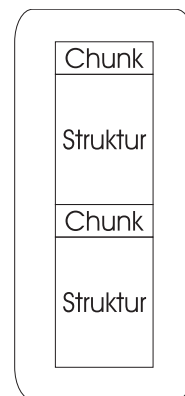


Abbildung 8.6:

So werden die Daten in der Datei abgelegt, Chunk für Chunk. Jeder Chunk beinhaltet dabei eine Struktur mit Daten oder auch wiederum einen Chunk.

Die Strukturen können beliebig lang bzw. kurz sein. In dem Chunk wird neben der eindeutigen ID auch die Länge mit gespeichert. Dadurch erhalten wir den Vorteil, dass wir auch mal eine Struktur überspringen können, wenn sie von uns nicht benötigt wird. Die Chunk-Struktur sieht so aus:



```
// --- Chunk ---
typedef struct _CHUNK{
    WORD    wIdentifier;
    ULONG   ulSize
} CHUNK_S;

typedef CHUNK_S* LPCHUNK;
```

Wir kennen in der aktuellen Version 1.0 des CBF-Formats bereits folgende Chunks:

```
#define V1_HEADER           0x0100 // Header
#define V1_VERTEX          0x0200 // Vertices
#define V1_FACE            0x0300 // Faces
#define V1_MESH            0x0400 // Meshes
#define V1_MATERIAL        0x0500 // Material
#define V1_JOINT           0x0600 // Joints
#define V1_JOINT_MAIN      0x0610 // Joints Main
#define V1_JOINT_KEYFRAME_ROT 0x0620 // Keyf.-Rotation
#define V1_JOINT_KEYFRAME_POS 0x0630 // Keyf.-Position
#define V1_ANIMATION       0x0700 // Animation
#define V1_END             0x9999 // End-Chunk
```

Zu jeder ID gibt es eine eigene Struktur. Somit ist das Verfahren beim Einlesen ist sehr trivial.

Einladen von Chunks

1. Einlesen einer Chunk-Struktur.
2. Über die ChunkID die folgende Struktur identifizieren.
3. Entscheiden, ob wir die Struktur lesen wollen, wenn nicht, einfach um die in der Chunk-Struktur stehende Größe vorspulen.
4. In die Subroutine zum Einlesen der aktuellen Struktur springen.
5. Die Struktur einlesen, evtl. verarbeiten und wieder zurück.

Das ist doch reines Pfannkuchenessen, oder?

Einen Chunk auslesen (GetNextChunk)

Unsere wichtigste Methode in unserem CBF-Loader ist die Methode `CZFXModel::GetNextChunk`. Diese Methode braucht einen gültigen Zeiger auf eine zu beschreibende Chunk-Struktur.

```
// -----
// Name: GetNextChunk( CHUNK_S &pChunk )
// Info: Reads the the next ChunkID
// Return      = (WORD)      next chunk id
// pChunk      = (CHUNK_S&) Pointer to Chunk
// -----
WORD CZFXModel::GetNextChunk( CHUNK_S &pChunk )
{
    // lies den nächsten Chunk
    fread( &pChunk, sizeof( CHUNK_S ), 1, m_pFile );

    // return ID
    return pChunk.wIdentifizier;
}
// -----
```



Sie liest an der aktuellen Dateiposition eine komplette Chunk-Struktur ein. Aus dieser Struktur wird dann der Identifier zurückgegeben. Damit lässt sich diese Methode schön in einer Switch-Anweisung wie in der Hauptmethode verwenden.

Unsere Hauptmethode

In der Hauptmethode testen wir Chunk für Chunk, was wir machen sollen. Es ist eine einfache große Switch/Case-Anweisung. Sie switcht auf den Rückgabewert von `GetNextChunkID()`. Das tut sie so lange, bis sie ein explizites `V1_END` für sich selbst erhält. Zum Schluss wird noch kurz getestet, ob Joints vorhanden sind. Sollte das nicht der Fall sein, dann löschen wir den sekundären Speicherbereich für die Vertices. Die bräuchten wir ansonsten für Animationen; da ohne Joints aber keine Animationen möglich sind, löschen wir sie in diesem Fall einfach wieder.

*Switch für alle
Chunks*

```
// -----
// Name: CheckForChunks( void )
// Info: Checks for Chunks
// Return      = (HRESULT)   Status
// -----
HRESULT CZFXModel::CheckForChunks( void )
{
    // Variablen init
    bool  bLoop = true;           // LoopFlag
```



```

// Schleife bis zur Unendlichkeit
do{
    // suche den nächsten Chunk
    switch( GetNextChunk( m_sChunk ) ){
        case V1_HEADER:    ReadHeader();    break;
        case V1_VERTEX:    ReadVertices();  break;
        case V1_FACE:      ReadFaces();     break;
        case V1_MESH:      ReadMesh();      break;
        case V1_MATERIAL:  ReadMaterials(); break;
        case V1_JOINT:     ReadJoints();    break;
        case V1_ANIMATION: ReadAnimations();break;
        case V1_END:       bLoop = false;   break;
    }
}while( bLoop );

// Haben wir Animationen?
if( m_sHeader.uiNumJoints == 0 )
{
    // die brauchen wir nicht mehr
    delete [] m_pVertices_Orig;
    m_pVertices_Orig = NULL;
}

// return OK
return S_OK;
}
// -----

```

Den Kopf einlesen (ReadHeader)

Header

Der Header ist zumeist am Anfang zu finden, dort macht er auch Sinn, weil wir in der Header-Struktur all die wichtigen Informationen zu unserem Modell speichern. Schauen wir mal in die Struktur:



```

// --- Header ---
typedef struct _CHUNKHEAD{
    UCHAR  ucIdentifier[ 4 ]; // Identifier
    UCHAR  ucName[ 32 ];     // Name
    UCHAR  ucAuthor[ 32 ];   // Autor
    UCHAR  ucEmail[ 32 ];    // E-Mail
    UCHAR  ucType;           // Typ
    UCHAR  ucVersion;        // Version
    ULONG  ulNumVertices;    // Anzahl Vertices
    ULONG  ulNumIndices;     // Anzahl Indices
    ULONG  ulNumFaces;       // Anzahl Faces
    ULONG  ulNumMeshes;      // Anzahl Meshs
    UINT   uiNumMaterials;   // Anzahl Materials
    UINT   uiNumJoints;      // Anzahl Joints
}

```

```

float  fAnimationFPS;      // FPS
float  fCurrentTime;       // aktuelle Zeit
UINT   uiNumFrames;        // Anzahl Frames
UINT   uiNumAnimations;    // Anzahl Animationen
} CHUNKHEAD_S;
typedef CHUNKHEAD_S* LPCHUNKHEAD; // Chunk-Header

```

Es sind also alle Informationen vorhanden, die für uns relevant sind. Wie immer schreibe ich zum Ende einer Struktur eine Typdeklaration, so dass wir später schön mit `LPCHUNKHEAD` einen Zeiger definieren können.

Die Methode zum Einlesen einer solchen Struktur aus einem File ist sehr einfach, wie auch alle weiteren, die einfach nur Strukturen einlesen. Zum Anfang hin loggen wir, dass jetzt ein Header eingelesen wird. Wenn wir mal einen Fehler suchen, lohnt sich solch eine Bug-Meldung. Danach initialisieren wir den Speicherbereich der Member-Variablen `m_sHeader`. Als Nächstes lesen wir den Header in `m_sHeader` ein. Und zum Schluss überprüfen wir, ob auch ein korrektes Chunk-Ende in Form eines `V1_END` vorhanden ist. Ist dem so, dann springen wir aus der Routine mit `OK` im Log und `S_OK` als Rückgabewert. Sollte dem nicht so sein, dann melden wir einen Fehler und kommen mit `E_FAIL` zurück.

*Einlesen des
Headers*

```

// -----
// Name: ReadHeader( void )
// Info: Liest den Header vom geöffneten File
//
// Return          = (HRESULT)    Status
// -----
HRESULT CZFXModel::ReadHeader( void )
{
    // Start
    LOG( 20, false, "Reading Header..." );

    // ausnullen
    ZeroMemory( &m_sHeader, sizeof( CHUNKHEAD_S ) );

    // lesen des Headers
    fread( &m_sHeader, sizeof( CHUNKHEAD_S ), 1, m_pFile);

    // suche den End-Chunk
    if( GetNextChunk( m_sChunk ) == V1_END ) {
        LOG( 20, true, " OK" );    // logit
        return S_OK;               // bye
    }

    // Fehler vermerken
    LOG( 1, true, " FAILED [Header]" );
}

```



```
// return
return E_FAIL;
}
// -----
```

Einlesen der Vertices (ReadVertices)

Was ist zu tun?

Die Methode zum Einlesen der Vertices ist etwas länger geraten als die Vorgängermethode. Sie ist vom Prinzip jedoch identisch. Als Erstes werden ein paar temporäre Variablen definiert. Dann loggen wir wie gewohnt und erstellen dann für unsere einzuladenden Vertices den notwendigen Speicherbereich. Sollte das fehlschlagen, so melden wir das gehorsamst, ansonsten wird der Speicherbereich initialisiert. Mit der Funktion `fread()` lesen wir dann alle Vertices in diesen Speicherbereich ein. Die beiden Member-Variablen `m_pVertices` und `m_pVertices_Orig` werden erstellt und initialisiert.

Da die Vertices in einem anderen Format vorliegen, als es die *ZFXEngine* verarbeiten kann, müssen wir die eingelesenen Vertices umformatieren. In unserer *ZFXEngine* wird das `CVERTEX`-Format genutzt. Die beiden Strukturen unterscheiden sich zum einen in der Komplexität und zum anderen in der Anordnung sowie Member-Definition.

Unsere Vertexstruktur aus dem CBF-Format sieht wie folgt aus:



Alle Strukturen, für die es in unserem CBF-Format eine ChunkID gibt, befinden sich in der Datei `zfxModelStructs.h`.



```
// --- Vertex---
typedef struct _VERTEX{
    float    fXYZ[ 3 ];    // Koordinaten
    float    fUV0[ 2 ];    // Texturkoordinaten 1
    float    fUV1[ 2 ];    // Texturkoordinaten 2
    ZFXVector fNormal;     // Normalenvektor
    USHORT   usReferences; // Referenzen
    UINT     uiBoneID_A;    // Bone-ID 1
    float    fWeight_A;     // Gewichtung 1
    UINT     uiBoneID_B;    // Bone-ID 2
    float    fWeight_B;     // Gewichtung 2
    BYTE     byFlags;       // Flags
} VERTEX_3F_S;
typedef VERTEX_3F_S* LPVERTEX_3F;
```

CVERTEX-Struktur

Wie gewohnt schließen wir die Struktur mit der Deklaration eines Pointers ab. Die Struktur, die intern von der Engine zur Verfügung gestellt und verarbeitet wird, sieht etwas anders aus:

```
typedef struct CVERTEX_TYPE {
    float  x, y, z;
    float  vcNE[3];
    float  tu, tv;
    float  fBone1, fWeight1;
    float  fBone2, fWeight2;
} CVERTEX;
```



Die Unterschiede sind aufgrund der unterschiedlichen Anwendung notwendig. In unserem Modellformat müssen Daten gespeichert werden, die nicht nur für diese eine Anwendung notwendig sind. In Editoren werden zum Beispiel andere Anforderungen notwendig.

Nachdem wir dann die Vertices sauber konvertiert haben, können wir die temporären Vertices aus dem Speicher entfernen. Nun muss ein END-Chunk (V1_END) an der aktuellen Dateiposition anliegen. Das wird geprüft, und wenn dem so ist, können wir diese Methode auch mit einem S_OK verlassen. Ansonsten melden wir einen Fehler und verlassen die Methode mit einem E_FAIL. Und hier ist die Methode:

```
// -----
// Name: ReadVertices( void )
// Info: Liest Vertices aus dem geöffneten File
//
// Return      = (HRESULT)      Status
// -----
HRESULT CZFXModel::ReadVertices( void )
{
    // initialisiere Variablen
    ULONG    ulNumVertices = m_sHeader.ulNumVertices;
    LPVERTEX_3F pVertices = NULL;

    LOG( 20, false, "Read Vertices [%d]", ulNumVertices );

    // Speicher allokalieren
    pVertices = new VERTEX_3F_S[ ulNumVertices ];
    if( !pVertices ){
        LOG( 1, true, " FAILED [VERTICES]" ); // logit
        return E_FAIL;                       // bye
    }

    // ausnullen
    ZeroMemory( pVertices, sizeof( VERTEX_3F_S ) *
        ulNumVertices );

    // lies alle Vertices
    fread( pVertices, sizeof( VERTEX_3F_S ),
        ulNumVertices, m_pFile );
```



```

// Speicher bereitstellen
m_pVertices      = new CVERTEX[ ulNumVertices ];
m_pVertices_Orig = new CVERTEX[ ulNumVertices ];
ZeroMemory( m_pVertices,      sizeof( CVERTEX ) *
            ulNumVertices );
ZeroMemory( m_pVertices_Orig, sizeof( CVERTEX ) *
            ulNumVertices );

// konvertiere die Vertices
for( ULONG ulCounter = 0; ulCounter <
    m_sHeader.ulNumVertices; ulCounter++ )
{
    // kopiere die Vertices
    memcpy( &m_pVertices[ ulCounter ].x,
            &pVertices[ ulCounter ].fXYZ,
            sizeof( float ) * 3 );
    memcpy( &m_pVertices[ ulCounter ].vcN,
            &pVertices[ ulCounter ].fNormal,
            sizeof( float ) * 3 );
    memcpy( &m_pVertices[ ulCounter ].tu,
            &pVertices[ ulCounter ].fUVO,
            sizeof( float ) * 2 );
    m_pVertices[ ulCounter ].fBone1 =
        (float) pVertices[ ulCounter ].uiBoneID_A;
    m_pVertices[ ulCounter ].fWeight1 =
        (float) pVertices[ ulCounter ].fWeight_A;
    m_pVertices[ ulCounter ].fBone2 =
        (float) pVertices[ ulCounter ].uiBoneID_B;
    m_pVertices[ ulCounter ].fWeight2 =
        (float) pVertices[ ulCounter ].fWeight_B;
}
// Speicher freigeben
delete [] pVertices;

// suche den End-Chunk
if( GetNextChunk( m_sChunk ) == V1_END ) {
    LOG( 20, true, " OK" );    // logit
    return S_OK;               // bye
}

LOG( 1, true, " FAILED [VERTICES]" );
return E_FAIL;
}
// -----

```

Die Methode ist zwar lang, aber dennoch ziemlich einfach, oder?

Triangle-Information einlesen (ReadFaces)

Ein Triangle, bei uns auch als Face bezeichnet, besteht immer aus drei Vertices. Wobei das nicht ganz stimmt, denn ein Face besteht aus drei Indices zu den Vertices. Diese Methode entspricht dem absoluten Standard. Wir holen uns einfach die Anzahl der einzulesenden Faces. Dann allokatieren wir den notwendigen Speicher in `m_pFaces`. Nachdem wir erfolgreich den Speicher bekommen haben, wird dieser initialisiert und mit einem einfachen `fread()` mit den gewünschten Daten gefüllt.

Triangle-Indices

Die Struktur, die wir betanken, sieht wie folgt aus:

```
// --- Face ---
typedef struct _FACE{
    ULONG        ulIndices[ 3 ];    // Indices
    ZFXVector     fNormal;           // Normalenvektor
    ULONG        ulMeshID;           // Mesh-ID
    UINT         uiMaterialID;       // Material-ID
    BYTE         byFlags;            // Flags
} FACE_S;
typedef FACE_S* LPFACE;
```



Ein Pointer auf die Struktur rundet das Bild ab. Sobald die Daten geladen sind, testen wir auf ein `V1_END`, um auch wirklich sicher sein zu können, dass alles glatt lief. Entsprechend beenden wir die Methode dann mit `S_OK` oder `E_FAIL`.

Einlesen der Faces

```
// -----
// Name: ReadFaces( void )
// Info: Liest die Faces aus dem geöffneten File
//
// Return      = (HRESULT) Status
// -----
HRESULT CZFXModel::ReadFaces( void )
{
    ULONG    ulNumFaces = m_sHeader.ulNumFaces; // temp Var

    LOG(20, false, "Reading Faces [%d]...", ulNumFaces );

    // Speicher allokatieren
    m_pFaces = new FACE_S[ ulNumFaces ];
    if( !m_pFaces ){
        LOG( 1, true, " FAILED [FACES]" );
        return E_FAIL;
    }

    // ausnullen
    ZeroMemory(m_pFaces, sizeof( FACE_S ) * ulNumFaces );
```



```

// lies alle Faces
fread(m_pFaces,sizeof( FACE_S ),u1NumFaces,m_pFile );

// suche den End-Chunk
if( GetNextChunk( m_sChunk ) == V1_END ) {
    LOG( 20, true, " OK" );
    return S_OK;
}

LOG( 1, true, " FAILED [FACES]" );

return E_FAIL;
}
// -----

```

Auch diese Methode ist wie die folgenden reines Pfannkuchenessen :-). Als Nächstes sind die Mesh-Daten dran. Die können auch ganz genauso eingelesen werden.

Das Netz (ReadMesh)

Der Unterschied zwischen dieser Methode und der Methode zum Einlesen von Faces ist trivial. Es werden nur unterschiedliche Strukturen eingelesen, mehr nicht. Ansonsten ist alles identisch. Klar dürfte sein, dass wir die Mesh-Daten in einer anderen Variable, nämlich in `m_pMeshs` speichern.

Die Struktur für unsere Meshs ist sehr einfach und pragmatisch gehalten:



```

// --- Mesh ---
typedef struct _MESH{
    char        cName[ 32 ];        // Name
    WORD        wNumFaces;          // Anzahl Faces
    PWORD       pIndices;           // Face-Index
    UINT        uiMaterialID;       // Material-ID
    BYTE        byFlags;            // Flags
} MESH_S;
typedef MESH_S* LPMESH;

```

Wie immer steht ein Pointer am Ende der Struktur. Und jetzt seht ihr die heiß erwartete Methode zum Einlesen der Mesh-Daten :-).



```

// -----
// Name: ReadMesh( void )
// Info: Lies die Meshs aus dem geöffneten File
//
// Return      = (HRESULT) Status
// -----HRESULT
CZFXModel::ReadMesh( void )

```



```

{
    ULONG    uNumMesh = m_sHeader.uNumMeshs;

    LOG( 20, false, "Reading Meshs [%d]...", uNumMesh );

    // allokiere Speicher
    m_pMeshs = new MESH_S[ uNumMesh ];
    if( !m_pMeshs ){
        LOG( 1, true, " FAILED [MESH]" );
        return E_FAIL;
    }

    // ausnullen
    ZeroMemory( m_pMeshs, sizeof( MESH_S ) * uNumMesh );

    // lies alle Meshs
    fread(m_pMeshs, sizeof( MESH_S ), uNumMesh, m_pFile );

    // suche den End-Chunk
    if( GetNextChunk( m_sChunk ) == V1_END ) {
        LOG( 20, true, " OK" );           // logit
        return S_OK;                     // bye
    }

    LOG( 1, true, " FAILED [MESH]" );

    return E_FAIL;
}
// -----

```

Das war doch wohl ein echter Pfannkuchen oder? Und was soll ich sagen? Die Materialien lesen wir exakt genauso ein. So langsam wird das etwas dröge. Aber gleich haben wir diesen Part hinter uns, und dann wird es wieder spannender, wenn es an die Animation geht.

Auf das Material kommt es an (ReadMaterial)

Wie schon bei den beiden Vorgängermethoden ist diese Methode auch nur durch eine andere Struktur sowie eine andere Variable dazu in der Lage, alle Materialien zu laden.

Hier noch mal im Schnelldurchlauf:

1. Anzahl der Materialien aus dem Header holen
2. Speicher in der Variable *m_pMaterials* allokieren
3. Materialien einlesen
4. Entsprechend S_OK oder E_FAIL zurückgeben

*Ablauf zum
Einlesen von
Materialien*

Die Materialstruktur, die wir einlesen, sieht wie folgt aus:



```
// --- Material ---
typedef struct _MATERIAL{
    char        cName[ 32 ];           // Name
    float        fAmbient[ 4 ];        // ambiente Farbe
    float        fDiffuse[ 4 ];        // diffuse Farbe
    float        fSpecular[ 4 ];       // spekuläre Farbe
    float        fEmissive[ 4 ];       // emissive Farbe
    float        fSpecularPower;       // spekulär-Stärke
    float        fTransparency;        // Transparenz
    char        cTexture_1[ 128 ];     // Textur-Name
    char        cTexture_2[ 128 ];     // Textur-Name
    BYTE        byFlags;              // Flags
} MATERIAL_S;
typedef MATERIAL_S* LPMATERIAL;
```

Auch hier haben wir als Abschluss wie gewohnt den Pointer auf die Struktur. Das Einlesen von Materialien sieht in traditioneller Quellcode-Form wie folgt aus:



```
// -----
// Name: ReadMaterials( void )
// Info: Lies die Materialien aus dem geöffneten File
//
// Return      = (HRESULT)      Status
// -----
HRESULT CZFXModel::ReadMaterials( void )
{
    UINT    uiNumMat = m_sHeader.uiNumMaterials;

    LOG(20,false, "Reading Materials [%d]...",uiNumMat );

    // Speicher allokieren
    m_pMaterials = new MATERIAL_S[ uiNumMat ];
    if( !m_pMaterials ){
        LOG( 1, true, " FAILED [MATERIALS]" );
        return E_FAIL;
    }

    // ausnullen
    ZeroMemory( m_pMaterials, sizeof( MATERIAL_S ) *
                uiNumMat );

    // lies die Materialien
    fread( m_pMaterials, sizeof( MATERIAL_S ), uiNumMat,
           m_pFile );

    // suche den End-Chunk
```

```

if( GetNextChunk( m_sChunk ) == V1_END ) {
    LOG( 20, true, " OK" );
    return S_OK;
}

LOG( 1, true, " FAILED [MATERIALS]" );
return E_FAIL;
}
// -----

```

Bei der nächsten Methode wird es wieder interessanter. Versprochen!

Die Joints, bitte (ReadJoints)

Hier lesen wir all die Informationen ein, die für unsere Joints relevant sind. Dabei kommt die Stärke des CBF-Formats zum Tragen, dass wir nämlich einfach eine Unterstruktur in dem Format integrieren können. Das heißt, dass wir durch ein `V1_JOINT` in diese Methode gesprungen sind. Jetzt fangen wir aber nicht, wie von den anderen Methoden gewohnt, damit an, gleich Speicherbereiche zu betanken, sondern legen einen neuen Joint an und lesen dann ein, welcher Teil des Joints jetzt kommt. Das machen wir einfach, indem wir einen neuen Chunk einlesen. Die Struktur ist also mit einer Ebene versehen. Dadurch haben wir wirklich alles für unsere Joints wie in einem Unterverzeichnis auf der Festplatte zusammen.

*Chunks in einem
Chunk*

Die Joint-Struktur enthält neben dem Namen und dem Parentnamen auch zwei Zeiger auf die noch einzuführenden Strukturen für die Keyframe-Rotation `KF_ROT_S` und die Keyframe-Translation `KF_POS_S`.

```

// --- Joints ---
typedef struct _JOINT{
    char        cName[ 32 ];           // Descriptor
    char        cParentName[ 32 ];    // Parentdescriptor
    WORD        wParentID;             // Parent-ID
    ZFXVector    vRotation;            // Rotation
    ZFXVector    vPosition;            // Position
    WORD        wNumKF_Rotation;       // Anzahl Rots
    WORD        wNumKF_Position;       // Anzahl Pos
    LPKF_ROT     pKF_Rotation;         // KF-Rotationen
    LPKF_POS     pKF_Position;         // Position
    bool         bAnimated;            // Animiert
    BYTE        byFlags;               // Flags
    ZFXMatrix    sMatrix;              // Matrix
    ZFXMatrix    sMatrix_absolute;     // Matrix absolut
    ZFXMatrix    sMatrix_relative;    // Matrix relativ
} JOINT_S;
typedef JOINT_S* LPJOINT;

```



*Matrizen für die
Transformation
der Joints*

Am Ende der Struktur findet ihr die notwendigen Matrizen. Diese benötigen wir, um später die exakte Position des Joints zu berechnen. Die oben schon erwähnten beiden Strukturen, die noch nachträglich geladen werden müssen, sind auch der Grund, warum wir den `V1_JOINT`-Chunk mit einer weiteren Ebene beglücken. Die beiden Ebenen werden als `V1_JOINT_KEYFRAME_ROT` sowie `V1_JOINT_KEYFRAME_POS` bezeichnet. Zu ihnen kommen wir, wenn sie in ihren eigenen Methoden eingelesen werden.

Aber schauen wir uns doch die Methode zum Einlesen der Joints direkt an. Wie gesagt: Achtet bitte auf die *Switch-case-Anweisung*.



```
// -----
// Name: ReadJoints( void )
// Info: Lies die Joints aus dem geöffneten File
//
// Return      = (HRESULT)    Status
// -----
HRESULT CZFXModel::ReadJoints( void )
{
    bool    bLoop = true;
    UINT    uiLoop = 0;
    UINT    uiNumJoints = m_sHeader.uiNumJoints;
    LPJOINT pJoint = NULL;

    LOG(20,false,"Reading Joints [%d]...",uiNumJoints );

    // Speicher allokieren
    m_pJoints = new JOINT_S[ uiNumJoints ];
    if( !m_pJoints ){
        LOG( 1, true, " FAILED [JOINTS]" );
        return E_FAIL;                // bye
    }

    // Endlos-Schleife
    do{
        // find the next chunk
        switch( GetNextChunk( m_sChunk ) )
        {
            case V1_JOINT_MAIN:
                pJoint = &m_pJoints[ uiLoop ];
                ReadJoint_Main( pJoint );
                uiLoop++;
                break;

            case V1_JOINT_KEYFRAME_ROT:
                ReadJoint_KeyFrame_Rot( pJoint ); break;

            case V1_JOINT_KEYFRAME_POS:
                ReadJoint_KeyFrame_Pos( pJoint ); break;
        }
    } while( true );
}
```

```

        case V1_END:
            bLoop = false;                break;
        }

    }while( bLoop );

    // suche den End-Chunk
    if( !bLoop ) {
        LOG( 20, true, " OK" );
        return S_OK;
    }

    LOG( 1, true, " FAILED [JOINTS]" );
    return E_FAIL;
}
// -----

```

Hier musste ein klein wenig getrickst werden. Wenn der aktuelle Chunk `V1_JOINT` (Einsprung in diese Methode) gelesen wird, dann steht als Nächstes gleich ein `V1_JOINT_MAIN`-Chunk an. Dieser signalisiert, dass ein komplett neuer Joint gelesen werden muss. Da wir zwar genau wissen, wie viele Joints vorhanden sind, uns aber eine notwendige Flexibilität erhalten wollen, ist das der beste Weg. Quasi ein »*Achtung ein neuer Joint*«-Hinweis.

»Neuer Joint«-
Marker

Also erstellen wir nach dem Hinweis einen neuen Joint in dem Array der Masterjoint-Liste `m_pJoints`. Den Zeiger darauf speichern wir in `pJoint`, mit welchem wir noch die gesamte Methode lang arbeiten, bis der nächste Joint gelesen werden soll oder wir alle gelesen haben. Der Zeiger `pJoint` wird als Erstes der Methode `ReadJoint_Main( pJoint );` übergeben. Diese liest den Joint ein. Danach wird der interne Zeiger schon mal erhöht. Damit zeigen wir dann nach der Erhöhung auf den zweiten (0,1,2,3..n) Joint in der Masterjoint-Liste `m_pJoints`.

Ist der Joint erstmal eingelesen, können dann die Keyframe-Rotation und Keyframe-Translation eingelesen werden. Dazu wird einfach geprüft, ob der nächste Chunk ein `V1_JOINT_KEYFRAME_POS` oder ein `V1_JOINT_KEYFRAME_ROT` ist. Entsprechend wird dann in die jeweiligen Untermethoden gesprungen. Ihr wisst ja noch, wir haben den aktuellen Joint nicht verloren, sondern er ist in `pJoint` gespeichert :-). Somit wird auch genau in den Joint die Rotation bzw. Translation eingelesen.

Sollte uns tatsächlich mal ein `V1_END` beim fluffigen Einlesen über den Weg laufen, dann beenden wir das Einlesen. Wir haben unsere Joints nun eingelesen und beenden mit `S_OK`.

Der Hauptjoint (ReadJoint_Main)

Die Struktur der Joints ist uns ja aus der Vormethode schon bekannt. Daher werde ich sie hier nicht wiederholen. Aus diesem Grund ist diese Methode auch wieder recht trivial gehalten und mal wieder ein reines Pfannkuchen-Schlachtfest. Also, let's go.



```
// -----
// Name: ReadJoint_Main( LPJOINT pJoint )
// Info: Lies Joints-Hauptteil aus dem geöffneten File
//
// Return      = (HRESULT)      Status
//
// pJoint      = (LPJOINT)      Parent-Joint
// -----
HRESULT CZFXModel::ReadJoint_Main( LPJOINT pJoint )
{
    // Start
    LOG( 20, false, "Reading Joint " );

    // lies die Joints
    fread( pJoint, sizeof( JOINT_S ), 1, m_pFile );

    // suche den End-Chunk
    if( GetNextChunk( m_sChunk ) == V1_END ) {
        LOG( 20, true, " OK" );
        return S_OK;
    }

    LOG( 1, true, " FAILED [JOINT_MAIN]" );
    return E_FAIL;
}
// -----
```

Wir bekommen also über den Zeiger `pJoint` den Zeiger auf den **schon** initialisierten Speicherbereich für den Joint. Daher können wir einfach frohen Mutes diesen per `fread(...)` einlesen. Wir kontrollieren nach dem Einlesen, ob noch ein `V1_END` einlesbar ist. Das hilft bei der Konsistenzprüfung. Sollten die Daten zum Beispiel korrupt (beschädigt) sein, so würde hier eher nicht ein sauberes `V1_END` stehen. Das würden wir dann merken und schneller im Format eingreifen können.

Die Rotation (ReadJoint_KeyFrame_Rot)

*Rotation von
Joints*

Für die exakte Positionierung sind die Rotationen der Joints innerhalb der einzelnen Frames sehr wichtig. Beim Animieren von Modellen werden die Bewegungen der Joints ja pro Frame gespeichert. Wenn also das Modell im Frame 5 noch den Arm gerade und im Frame 6 einen Winkel am Ellenbogen

von 23° hat, dann würde das genau so in den Daten und dieser Struktur abgespeichert werden. Wenn wir dann beim Animieren von Frame 5 zu 6 wechseln, dann lesen wir den Winkel für Frame 6 aus und rotieren entsprechend den Joint. Einfach, oder?

Und genauso simpel ist die Struktur.

```
// --- Keyframe-Rotation ---
typedef struct _KF_ROT{
    float      fTime;           // Zeit
    ZFXVector  vRotation;       // Rotation
} KF_ROT_S;
typedef KF_ROT_S*  LPKF_ROT;
```



Die Methode ist entsprechend »straight«, sie sollte uns wirklich nicht überraschen.

```
// -----
// Name: ReadJoint_KeyFrame_Rot( LPJOINT pJoint )
// InfoLies Keyframe Rotationen aus geöffnetem File
//
// Return      = (HRESULT)      Status
//
// pJoint      = (LPJOINT)      Parent-Joint
// -----
HRESULT CZFXModel::ReadJoint_KeyFrame_Rot( LPJOINT
                                           pJoint )
{
    UINT  uiNumKeys = pJoint->wNumKF_Rotation;

    LOG(20,false,"Reading KF Rot. [%d]...",uiNumKeys );

    // Speicher allokieren
    pJoint->pKF_Rotation = new KF_ROT_S[ uiNumKeys ];
    if( !pJoint->pKF_Rotation ){
        LOG(1,true,"FAILED [JOINT_KEYFRAME_ROTATIONS]" );
        return E_FAIL;
    }

    // ausnullen
    ZeroMemory( pJoint->pKF_Rotation, sizeof( KF_ROT_S )
               * uiNumKeys );

    // lies die Rotationen
    fread( pJoint->pKF_Rotation, sizeof( KF_ROT_S ),
           uiNumKeys, m_pFile );

    // suche den End-Chunk
    if( GetNextChunk( m_sChunk ) == V1_END ) {
```



```

        LOG( 20, true, " OK" );
        return S_OK;
    }

    LOG( 1, true, " FAILED [JOINT_KEYFRAME_ROTATIONS]" );
    return E_FAIL;
}
// -----

```

Wir bekommen aus der Elternmethode den Zeiger auf den aktuellen Joint in `pJoint` übermittelt. Als Erstes reservieren wir den benötigten Speicher mittels unserer Struktur `KF_ROT_S`. Danach lesen wir die Daten aus der Datei vom aktuellen Zeiger aus ein.

Zum Testen, ob auch alles glatt lief, wollen wir ein `v1_END` einlesen dürfen. Entsprechend beenden wir die Methoden in der Regel mit `S_OK`.

Die Position (`ReadJoint_KeyFrame_Pos`)

Hier wird uns nun wirklich keine Überraschung erwarten. Daher fasse ich mich hier auch entsprechend kurz. Mit einem Blick auf die Struktur werden wir feststellen, dass es hier deutliche Parallelen zu den Rotationen für einen Joint gibt.



```

// --- Keyframe-Position ---
typedef struct _KF_POS{
    float      fTime;                // Zeit
    ZFXVector   vPosition;           // Position
} KF_POS_S;
typedef KF_POS_S*   LPKF_POS;

```

Auch die Methode ist bis auf die Struktur selbst eindeutig identisch mit der Methode `CZFXModel::ReadJoint_KeyFrame_Rot`.



```

// -----
// Name: ReadJoint_KeyFrame_Pos( LPJOINT pJoint )
// Info: Lies Keyframe-Positionen aus geöffnetem File
//
// Return      = (HRESULT)      Status
//
// pJoint      = (LPJOINT)      Parent-Joint
// -----
HRESULT CZFXModel::ReadJoint_KeyFrame_Pos( LPJOINT
                                           pJoint )
{
    UINT    uiNumKeys = pJoint->wNumKF_Position;

    LOG( 20, false, "Reading KeyFrame Positions [%d]...",

```



```

        uiNumKeys );

// allokiere Speicher
pJoint->pKF_Position = new KF_POS_S[ uiNumKeys ];
if( !pJoint->pKF_Position ){
    LOG(1,true," FAILED [JOINT_KEYFRAME_POSITIONS]" );
    return E_FAIL;
}

// ausnullen
ZeroMemory( pJoint->pKF_Position, sizeof( KF_POS_S )
            * uiNumKeys );

// lies die Positionen
fread( pJoint->pKF_Position, sizeof( KF_POS_S ),
        uiNumKeys, m_pFile );

// suche den End-Chunk
if( GetNextChunk( m_sChunk ) == V1_END ) {
    LOG( 20, true, " OK" );
    return S_OK;
}

LOG( 1, true, " FAILED [JOINT_KEYFRAME_POSITIONS]" );
return E_FAIL;
}
// -----

```

Sei animiert (ReadAnimations)

Jetzt sind wir schon bei der letzten Methode, die etwas aus der Datei lädt. Da nun auch alle Informationen für die Animation geladen sind, brauchen wir die Animation noch selbst. Damit ist gemeint, welche Animation von welchem Frame bis zu welchem Frame geht. Die Struktur für die Animationen sieht wie folgt aus.

*Gleich haben wir's
geladen*

```

// --- Animations ---
typedef struct _ANIMATION{
    char        cName[ 64 ];           // Bezeichnung
    float        fStartFrame;          // Startframe
    float        fEndFrame;            // Endframe
    bool         bActive;              // Aktiv
} ANIMATION_S;
typedef ANIMATION_S*  LPANIMATION;

```



In dem Konverter, der zum Beispiel *Milkshape*-Dateien in unser Format konvertiert, braucht man immer eine Textdatei, in der die entsprechenden Modell-Animationen definiert sind. Der Aufbau ist recht einfach gehalten. Am Anfang wird die Anzahl der Animationen definiert, und danach werden

die Animationen mit dem Start und Ende sowie mit einer kleinen Beschreibung aufgeführt. Das Flag `bActive` in der Struktur wird erst später beim Anwenden der Animation genutzt. Es sagt an, welche Animation gerade aktiv ist.

```
Animation.txt // Dieses File führt alle Animationen des Modells auf
//-----
// Animationsfile für das Model
// Als Erstes die Anzahl der Animationen
Number: "14"
S: "001" E: "001" D: "Bind Position"
S: "002" E: "020" D: "Walk Cycle 1"
S: "022" E: "036" D: "Walk Cycle 2"
S: "038" E: "047" D: "Zombie being Attacked 1"
S: "048" E: "057" D: "Zombie being Attacked 2"
S: "059" E: "075" D: "Blown away onto his back"
S: "078" E: "088" D: "Still lying down and twitching
                (offset)"
S: "091" E: "103" D: "Die and fall forwards"
S: "106" E: "115" D: "Kick Attack"
S: "117" E: "128" D: "Punch/Grab Attack"
S: "129" E: "136" D: "Head Butt :-)"
S: "137" E: "169" D: "Idle 1"
S: "170" E: "200" D: "Idle 2"
```

Schauen wir uns jetzt mal die Methode an, die die Animationen aus der entsprechenden Datei einliest. Sie wird euch sicherlich nicht aus der Bahn werfen.



```
// -----
// Name: ReadAnimations( void )
// Info: Lies die Animationen aus geöffnetem File
//
// Return      = (HRESULT) Status
// -----
HRESULT CZFXModel::ReadAnimations( void )
{
    UINT    uiNumAnim = m_sHeader.uiNumAnimations; //tmp

    LOG( 20, false, "Reading Animations [%d]...",
        uiNumAnim );

    // allokieren Speicher
    m_pAnimations = new ANIMATION_S[ uiNumAnim ];
    if( !m_pAnimations ){
        LOG( 1, true, " FAILED [ANIMATIONS]" );
        return E_FAIL;
    }
}
```

```

// ausnullen
ZeroMemory( m_pAnimations, sizeof( ANIMATION_S ) *
            uiNumAnim );

// lies die Animationen
fread( m_pAnimations, sizeof( ANIMATION_S ),
      uiNumAnim, m_pFile );

// suche den End-Chunk
if( GetNextChunk( m_sChunk ) == V1_END ) {
    LOG( 20, true, " OK" );
    return S_OK;
}

LOG( 1, true, " FAILED [ANIMATIONS]" );
return E_FAIL;
}
// -----

```

Das war es mit dem Laden. Alle, aber auch wirklich alle Daten des Modells sind nun im Speicher geladen und können in der Klasse über Zeiger abgerufen werden. Bevor wir jedoch unser Modell in Animation sehen, müssen wir noch die Daten ein wenig bearbeiten.

*Melde: Laden
komplett!*

Passt es? (SetScaling)

Es passiert immer wieder – vor allem, wenn ihr freie Modelle aus dem Internet nutzt –, dass die Größen nicht stimmen oder zumindest nicht aufeinander abgestimmt sind. Und dafür müssen wir eine einfache Methode haben, die die Größe angleicht.

Dazu müssen wir als Erstes die Bounding-Box bestimmen. Dann nehmen wir die Größe auf der Y-Achse, um daraus einen Skalierungsfaktor zu berechnen. Wir übergeben dieser Methode den gewünschten Skalierungsfaktor, der einfach mittels der folgenden Formel den eigentlichen Skalierungsfaktor in `fScaling` ergibt:

*Bounding-Box
berechnen*

```

// Calc scaling
fScaling = ( m_sBBoxMax.y - m_sBBoxMin.y ) / fScale;

```

Mit diesem Skalierungsfaktor werden wir alle Vertices und alle Joints so bearbeiten, dass unser Modell die notwendige Größe hat. Schauen wir mal in die Methode selbst.



```
// -----
// Name: SetScaling( float fScale /* = 0.0f */ )
// Info: Setze Skalierung oder 0.0f für keine Skalierung
//
// fScale      = (float)      Scaling
// -----
void CZFXModel::SetScaling( float fScale /* = 0.0f */ )
{
    ULONG      ulCounter      = 0;          // Zähler
    ULONG      ulInner        = 0;          // Zähler
    CVERTEX     *pVertex       = NULL;      // temporär
    float       fScaling       = 0.0f;      // Skalierung
    LPJOINT     pJoint         = NULL;      // Joint

    // müssen wir skalieren?
    if( fScale == 0.0f ) return;

    // berechne die Bounding Box
    m_sBBoxMin.x = 999999.0f; m_sBBoxMin.y = 999999.0f;
    m_sBBoxMin.z = 999999.0f; m_sBBoxMax.x = -999999.0f;
    m_sBBoxMax.y = -999999.0f; m_sBBoxMax.z = -999999.0f;

    // Setup der Vertices
    for( ulCounter = 0; ulCounter <
        m_sHeader.ulNumVertices; ulCounter++ )
    {
        // aktuellen Vertex holen
        pVertex = &m_pVertices[ ulCounter ];

        // Box wenn nötig erweitern
        m_sBBoxMax.x = MAX( m_sBBoxMax.x, pVertex->x );
        m_sBBoxMax.y = MAX( m_sBBoxMax.y, pVertex->y );
        m_sBBoxMax.z = MAX( m_sBBoxMax.z, pVertex->z );
        m_sBBoxMin.x = MIN( m_sBBoxMin.x, pVertex->x );
        m_sBBoxMin.y = MIN( m_sBBoxMin.y, pVertex->y );
        m_sBBoxMin.z = MIN( m_sBBoxMin.z, pVertex->z );
    }

    // Skalierung berechnen
    fScaling = ( m_sBBoxMax.y - m_sBBoxMin.y ) / fScale;

    // alle Vertices skalieren
    for( ulCounter = 0; ulCounter <
        m_sHeader.ulNumVertices; ulCounter++ )
    {
        // aktuellen Vertex holen
        pVertex = &m_pVertices[ ulCounter ];

        // Vertex skalieren
        pVertex->x /= fScaling;
        pVertex->y /= fScaling;
    }
}
```

```

        pVertex->z    /= fScaling;
    }

    // kopieren, wenn wir eine Animation haben
    if( m_sHeader.uiNumJoints > 0 )
        memcpy( m_pVertices_Orig, m_pVertices, sizeof(
            CVERTEX ) * m_sHeader.uiNumVertices );

    // Bones skalieren
    for( ulCounter = 0; ulCounter <
        m_sHeader.uiNumJoints; ulCounter++ )
    {
        // aktuellen Bone holen
        pJoint      = &m_pJoints[ ulCounter ];

        // Bone skalieren
        pJoint->vPosition.x    /= fScaling;
        pJoint->vPosition.y    /= fScaling;
        pJoint->vPosition.z    /= fScaling;

        // Keyframe-Position skalieren
        for( ulInner = 0; ulInner <
            pJoint->wNumKF_Position; ulInner++ )
        {
            pJoint->pKF_Position[ ulInner ].vPosition.x
                /= fScaling;

            pJoint->pKF_Position[ ulInner ].vPosition.y
                /= fScaling;

            pJoint->pKF_Position[ ulInner ].vPosition.z
                /= fScaling;
        }

        // Box erstellen
        m_sAabb.vcMin.x = m_sBBoxMin.x;
        m_sAabb.vcMin.y = m_sBBoxMin.y;
        m_sAabb.vcMin.z = m_sBBoxMin.z;
        m_sAabb.vcMax.x = m_sBBoxMax.x;
        m_sAabb.vcMax.y = m_sBBoxMax.y;
        m_sAabb.vcMax.z = m_sBBoxMax.z;
        m_sAabb.vcCenter.x = ( m_sBBoxMax.x -
                               m_sBBoxMin.x ) / 2;
        m_sAabb.vcCenter.y = ( m_sBBoxMax.y -
                               m_sBBoxMin.y ) / 2;
        m_sAabb.vcCenter.z = ( m_sBBoxMax.z -
                               m_sBBoxMin.z ) / 2;
    }
}
// -----

```

Bounding-Box

In dieser Methode sind zwei Aufgaben enthalten. Zum einen wird die Bounding-Box des Modells berechnet und zum anderen die Skalierung vorgenommen. Es macht ja auch Sinn, das zusammenzulegen, da immer dann wenn wir die Größe ändern, sich auch die Bounding-Box verändert. Die Bounding-Box wird übrigens wie gesehen in den Member-Variablen `m_sBBox_Max`, `m_sBBoxMin` und in `m_sAabb` gespeichert.

Als zweites skalieren wir das Modell. Dazu fangen wir am Anfang an, als erstes die Größe in der Bounding-Box zu ermitteln. Wir loopen also durch alle Vertices und ermitteln durch die beiden Makros MAX und MIN jeweils die minimale und die maximale Ausdehnung.

Danach berechnen wir mit der schon oben aufgeführten Formel den Skalierungsfaktor. Durch diesen in `fScaling` gespeicherten Faktor werden dann als Erstes alle Vertices geteilt. Somit erhalten wir die korrekte Position.

Wenn Animationen vorhanden sind – das bekommen wir durch die Anzahl der Joints recht schnell mit –, dann kopieren wir die neuen Vertices in unseren originalen Vertex-Bereich, aus dem wir später die animierten Vertices ziehen. Das ist sozusagen der Bereich mit den Master-Vertices der Grundposition des Modells. Dann laufen wir noch schnell über die Joints und skalieren diese zur korrekten Position. Würden wir das vergessen, sähe unser Modell beim Animieren recht »tentaklig« aus.

Zum Ende der Methode wird noch fix die AABB aufbereitet. Sie wird in der Engine später für die Kollision benötigt.

8.3 Verarbeitung der Daten im Speicher

Da jetzt wirklich alles geladen ist, können wir uns an die Verarbeitung der Daten machen. Diese werden für die *ZFXEngine 2.0* entsprechend aufbereitet. Dazu haben wir in unserer Klasse eine Methode, die sich `Prepare()` nennt. Diese bereitet die Daten so vor, dass wirklich alles an den Platz gerät, wo es hin soll.

Schauen wir uns die Methode im Detail an.

Vorbereitung der Daten (Prepare)

Sortieren der Daten

Diese Methode sieht gewaltiger aus, als sie wirklich ist. Hier sortieren wir alle Daten in verschiedene Buffer, die uns später mehr Performance beim Rendern geben. So werden alle Faces entsprechend ihrer MaterialID sortiert. So können wir uns teure Material-Änderungen in der Engine sparen. Diese Änderungen betreffen auch die Indices: Sie werden ebenfalls passend sortiert.

Eine weitere Aufgabe ist, die Materialien und die Texturen in den Skin-Manager zu schieben, so dass sie später auch verfügbar sind. Hier ist der erste Teil der Methode:

```
// -----
// Name: Prepare( void )
// Info: Vorbereiten des Modells
//
// Return      = (HRESULT)      Status
// -----
HRESULT CZFXModel::Prepare( void )
{
    // Variablen init
    ULONG      ulNumIndices  = 0;
    ULONG      ulNumVertices = 0;
    UINT       uiCurrentMat  = 0;
    WORD       pIndex        = NULL;      // Index
    ULONG      ulCounter     = 0;
    LPMATERIAL pMaterial     = NULL;
    char       cTexture[256] = { 0 };
    PCHAR      pcSeparator   = NULL;
    ULONG      ulIndexCount  = 0;

    // 1. Setup der Bones
    SetupBones();

    LOG( 20, false, "Sort Indices by Material [%d]",
        m_sHeader.uiNumMaterials );

    // maximalen Speicher berechnen
    m_sHeader.ulNumIndices = m_sHeader.ulNumFaces * 3;
    pIndex                = new WORD[ m_sHeader.ulNumIndices ];

    m_ppIndices = new PVOID[m_sHeader.uiNumMaterials ];
    ZeroMemory( m_ppIndices, sizeof( PVOID ) *
        m_sHeader.uiNumMaterials );

    m_pIndices = new WORD[ m_sHeader.ulNumIndices ];
    ZeroMemory( m_pIndices, sizeof( WORD ) *
        m_sHeader.ulNumIndices );

    m_puiNumIndices = new UINT[m_sHeader.uiNumMaterials];
    ZeroMemory( m_puiNumIndices, sizeof( UINT ) *
        m_sHeader.uiNumMaterials );

    m_puiSkinBuffer = new UINT[m_sHeader.uiNumMaterials];
    ZeroMemory( m_puiSkinBuffer, sizeof( UINT ) *
        m_sHeader.uiNumMaterials );
}
```



```
////////////////////////////////////
// __FORTSETZUNG_FOLGT__ //
////////////////////////////////////
```

Bis hier hin haben wir nichts weiter gemacht, als die Bones mit der Methode `SetupBones()` ausgerichtet und die notwendigen Buffer für unsere Engine in unserer Klasse angelegt.

Die Buffer haben folgende Bedeutung:

<code>pIndex</code>	Temporäres Indexarray zum Umkopieren
<code>m_ppIndices</code>	Zeigerarray, das auf die einzelnen Indexarrays zeigt
<code>m_pIndices</code>	Array für alle Indices des Modells
<code>m_puiNumIndices</code>	Array, das die Anzahl der Indices in den einzelnen Indexarrays hält

Ablaufschema

Das funktioniert jetzt wie folgt: Als Erstes holen wir uns in `pIndex` den maximal notwendigen Speicher für alle Indices. Dann ordnen wir anhand der Materialien (`uiMaterialID`) des Faces die einzelnen Indices in das `pIndex`-Array. Zusätzlich speichern wir alle Indices, schön sequenziell nach Materialien geordnet, in dem globalen Array `m_pIndices` ab. Das Array ist nur dazu da, falls wir mal von außerhalb schnell auf alle Indices zugreifen wollen.

Wenn die Indices in dem `pIndexarray` gelandet sind, die alle zu dem gleichen Material gehören, dann legen wir in dem Zeigerarray `m_ppIndices` einen Speicherbereich für die exakte Anzahl der Indices an. Die Anzahl wird in `m_puiNumIndices` und in das neue fluffige Indexarray gespeichert, auf das `m_ppIndices[ uiCurrentMat ]` zeigt. Dort kopieren wir die gefilterten Indices hin. Der temporäre Index `pIndex` wird hier übrigens nicht freigegeben, weil wir ihn einfach im nächsten Durchlauf für das nächste Material überschreiben. Freigeben werden wir ihn zum Ende der Methode hin. Das spart ein wenig Zeit :-).



```
////////////////////////////////////
// __FORTSETZUNG__ //
////////////////////////////////////
```

```
// sortiere alle Faces in das Indexarray
do{

    // ausnullen
    ZeroMemory( pIndex, sizeof(WORD) *
                m_sHeader.uiNumIndices );

    // Zähler zurücksetzen
```



```

ulNumIndices = 0;

// Schleife über alle Faces
for( ulCounter = 0; ulCounter <
    m_sHeader.ulNumFaces; ulCounter++ )
{
    // Immer noch dasselbe Material?
    if( m_pFaces[ ulCounter ].uiMaterialID ==
        uiCurrentMat )
    {
        m_pIndices[ ulIndexCount++ ] =
            pIndex[ ulNumIndices++ ] =
            (WORD)m_pFaces[ ulCounter ].ulIndices[ 0 ];

        m_pIndices[ ulIndexCount++ ] =
            pIndex[ ulNumIndices++ ] =
            (WORD)m_pFaces[ ulCounter ].ulIndices[ 1 ];

        m_pIndices[ ulIndexCount++ ] =
            pIndex[ ulNumIndices++ ] =
            (WORD)m_pFaces[ ulCounter ].ulIndices[ 2 ];
    }
}
// Genug Indices?
if( !ulNumIndices )
{
    // neues Material
    uiCurrentMat++;

    LOG( 1, true, "STOP Error: Not enough
        Indices..." );

    continue;
}
m_puiNumIndices[ uiCurrentMat ] = ulNumIndices;
m_ppIndices[ uiCurrentMat ] = new WORD[ulNumIndices];
memcpy( m_ppIndices[ uiCurrentMat ], pIndex,
    sizeof(WORD) * ulNumIndices );

//////////
// __FORTSETZUNG_FOLGT__ //
//////////

```

Nachdem wir jetzt die Indices sauber zugeordnet haben, werden wir die Materialien im Skin-Manager der *ZFXEngine* anlegen. Dazu nehmen wir uns das Render-Device, das mit dem Pointer `m_pRenderDevice` schon im Konstruktor der Klasse übergeben wurde, und holen uns den Skin-Manager mit der Methode `GetSkinManager()`. Der Skin-Manager besitzt die Methode `AddSkin()`, mit der wir der *ZFXEngine 2.0* unser frisches Material zuweisen können.

Skin-Manager

Da unsere Farben aus der Materialstruktur nicht sofort mit der ZFXCOLOR-Struktur kompatibel sind, müssen wir mit Hilfe von einem Cast (ZFXCOLOR*) die Farben casten. Sie sind vom Aufbau her beide identisch.

Als Nächstes suchen wir den korrekten Texturnamen aus dem Material heraus. Wenn der Name vorhanden ist, fügen wir über den Skin-Manager mit AddTexture() diese Textur hinzu.



```

////////////////////////////////////
// __FORTSETZUNG__ //
////////////////////////////////////

// aktuelles Material setzen
pMaterial = &m_pMaterials[ uiCurrentMat ];

// Material auslesen
if( FAILED( m_pRenderDevice->
    GetSkinManager()->AddSkin(
        (ZFXCOLOR*)&pMaterial->fAmbient,
        (ZFXCOLOR*)&pMaterial->fDiffuse,
        (ZFXCOLOR*)&pMaterial->fEmissive,
        (ZFXCOLOR*)&pMaterial->fSpecular,
        pMaterial->fSpecularPower,
        &m_puiSkinBuffer[ uiCurrentMat ])))
{
    LOG( 1, true, " FAILED [LOAD SKIN %d]",
        uiCurrentMat );
}

// Texturen vorbereiten
ZeroMemory( cTexture, sizeof( char ) * 256 );
pcSeperator = strchr(strrev(strdup(m_pcFileName)),
    '/' );
if( !pcSeperator ) pcSeperator = strchr( strrev(
    strdup( m_pcFileName ) ), 92 );
if( pcSeperator ) strcpy( cTexture, strrev(
    pcSeperator ) );
strcat( cTexture, pMaterial->cTexture_1 );

// Texturen laden
if( FAILED( m_pRenderDevice->
    GetSkinManager()->AddTexture(
        m_puiSkinBuffer[ uiCurrentMat ],
        cTexture, false, 0, NULL, 0 ) ) )
{
    LOG( 1, true, " FAILED [LOAD TEXTURE %s]",
        pMaterial->cTexture_1 );
}

```

```

// neues Material setzen
uiCurrentMat++;

}while( uiCurrentMat != m_sHeader.uiNumMaterials );

// Speicher freigeben
delete [] pIndex;

LOG( 20, true, " done" );

return S_OK;
}
// -----

```

Wir loopen so lange durch die Daten, bis wir kein weiteres Material mehr haben. Das bedeutet aber auch, dass ein Modell immer mindestens ein Material haben muss, damit wir es sauber darstellen können.

Und hier, zum Ende der Methode hin, kommt dann auch unser lang erwartetes Freigeben des temporären Indexarrays.

Skeletale Animation (SetupBones)

In der `Prepare()`-Methode wurde diese Methode als Erstes aufgerufen. Wir berechnen hier die korrekte Position der Vertices in dem ersten Animations-frame. Wir müssen uns drei Matrizen merken, um uns leichter durch den Quellcode zu bewegen:

➔ `pJoint->sMatrix_relative`: Hier wird die relative Rotation und Translation des Joints vom Parent-Joint-Parent-Joint aus gespeichert. Wenn wir also unseren Unterarm um 15° drehen, dann würden diese 15° in dieser Matrix gespeichert werden, da sie relativ zu dem Parent-Joint ist.

Die drei Matrizen

➔ `pJoint->sMatrix_absolute`: In dieser Matrix wird die absolute Rotation und Translation zum lokalen Ursprung (X 0.0, Y 0.0, Z 0.0) gespeichert. Sie wird innerhalb der Hierarchie der einzelnen Joints berechnet: also $\text{Parent-Joint} \rightarrow \text{sMatrix\_absolute} * \text{Joint} \rightarrow \text{sMatrix\_relative}$.

➔ `pJoint->sMatrix`: Diese Matrix hält nur die transponierte `sMatrix_relative`. Bei dem Berechnen der Position der Vertices werden wir diese als finale Matrix verwenden.

Das Folgende ist die Methode zum Initialisieren der Bones im groben Ablauf. Wir müssen als Erstes die Matrizen für die Joints berechnen und können erst danach die Vertices mit den berechneten Matrizen positionieren. Klingt doch einfach, oder? Also auf zum Quellcode. Ich werde ihn euch Stück für Stück vorstellen und die jeweiligen Teile entsprechend erläutern.

Setup der Bones



```
// -----
// Name: SetupBones( void )
// Info: Vorbereitung der Bones
//
// Return      = (HRESULT)   Status
// -----
HRESULT CZFXModel::SetupBones( void )
{
    // Variablen
    LPJOINT    pJoint      = NULL;    // Joint
    ULONG      ulCounter   = 0;        // Zähler
    UINT       uiLoop      = 0;        // Zähler
    UINT       uiParentID  = 0;        // Parent-ID
    ZFXVector  sVector_A;              // Vector
    ZFXVector  sVector_B;              // Vector
    CVERTEX    *pVertex    = NULL;    // temporär
    ZFXMatrix  matTemp; matTemp.Identity();// temporär

    // Sind überhaupt Bones da?
    if( m_sHeader.uiNumJoints == 0 ) return S_OK;

    // Matrizen anheizen
    for( ulCounter = 0; ulCounter <
        m_sHeader.uiNumJoints; ulCounter++ )
    {
        // Joint holen
        pJoint = &m_pJoints[ ulCounter ];

        // Rotation in Matrix erzeugen
        pJoint->sMatrix_relative = CreateRotationMatrix(
            &pJoint->vRotation );

        // Position setzen
        pJoint->sMatrix_relative._14 =
            pJoint->vPosition.x;

        pJoint->sMatrix_relative._24 =
            pJoint->vPosition.y;

        pJoint->sMatrix_relative._34 =
            pJoint->vPosition.z;

        ////////////////////////////////////
        // __FORTSETZUNG_FOLGT__ //
        ////////////////////////////////////
    }
}
```

Als Erstes prüfen wir, ob überhaupt Joints vorhanden sind. Wir müssen sie ja nicht berechnen, wenn wir keine haben :-). Sobald aber klar ist, dass das Modell aus Fleisch und Knochen besteht, fangen wir an, durch alle Joints zu

lopen. In der Variablen `pJoint` wird jeweils der aktuelle Joint, der in Bearbeitung ist, gespeichert.

Wir erstellen uns in dem Joint dann aufgrund der gespeicherten Rotation mit Hilfe der entsprechenden Funktion `CreateRotationMatrix()` die Rotationsmatrix für den Joint. Jetzt fehlt zu unserem Glück in der Matrix noch die Position, und die speichern wir hier ungewöhnlicherweise innerhalb der Matrix an `_14`, `_24` und `_34` ab. Das machen wir hier so, weil wir im Moment noch mit transponierten Matrizen arbeiten, und diese erst am Ende in die von unserer Engine verwendete Form transponieren.

*Transponierte
Matrizen*

```

////////////////////
// __FORTSETZUNG__ //
////////////////////

// Parent finden...
for( uiLoop = 0; uiLoop < m_sHeader.uiNumJoints;
    uiLoop++ )
{
    // merken der Parent ID
    uiParentID = 255;

    if( strcmp( m_pJoints[ uiLoop ].cName,
        pJoint->cParentName ) == 0 )
    {
        // gefunden
        uiParentID = uiLoop;
        break;
    }
}

// gefundene ID merken
pJoint->wParentID = uiParentID;

// Haben wir den Parent gefunden?
if( uiParentID != 255 )
{
    // Parent gefunden, daher müssen wir
    // seine Absolut-Matrix mit unserer Relativ-
    // Matrix multiplizieren, um unsere Absolut-
    // Matrix zu erhalten!
    pJoint->sMatrix_absolute =
        m_pJoints[ uiParentID ].sMatrix_absolute *
        pJoint->sMatrix_relative;
} else {
    // kein Parent ... what a pity
    // => relative ist absolute Matrix
    pJoint->sMatrix_absolute =

```

}
LIST

```

        pJoint->sMatrix_relative;

    }

    // finale Matrix
    pJoint->sMatrix.TransposeOf(
        pJoint->sMatrix_absolute );

    // transponieren
    matTemp = pJoint->sMatrix_relative;
    pJoint->sMatrix_relative.TransposeOf( matTemp );
}

////////////////////
// __FORTSETZUNG_FOLGT__ //
////////////////////

```

Parent-Bone

Jetzt wird es einfach. Wir suchen anhand des Namens unseren Parent. Der Name ist jeweils in `pJoint->cParentName` gespeichert. Wenn wir einen Parent gefunden haben, dann speichern wir die ID von dem Joint. Somit sind wir später deutlich schneller. Sollte keiner gefunden werden, speichern wir als Zeichen einfach eine 255 (hex FF) in der ID ab.

Prompt testen wir gleich danach, ob eine wirkliche ID oder nur die 255 gesetzt wurde. Ist eine valide ID gesetzt, dann multiplizieren wir die absolute Matrix des Parents mit unserer relativen Matrix. Dadurch erhalten wir unsere eigene absolute Matrix und könnten mit dieser dann vom Ursprung (X 0.0, Y 0.0, Z 0.0), wie oben schon beschrieben, direkt zu dem Joint springen.

Sollten wir keinen Parent haben, so setzen wir einfach unsere relative Matrix als die absolute Matrix.

OpenGL

Danach transponieren wir die absolute Matrix, in die finale Matrix um mit dieser gleich zu arbeiten. Auch die relative Matrix muss transponiert werden. Ein kleiner Hinweis an dieser Stelle: Wenn ihr das mal in Richtung OpenGL portieren wollt, dann könnt ihr einfach die Transponierung der Matrizen weglassen.



```

////////////////////
// __FORTSETZUNG__ //
////////////////////

    // Vertices Setup
    for( ulCounter = 0; ulCounter <
        m_sHeader.ulNumVertices; ulCounter++ )
    {
        // aktuellen Vertex holen
    }

```

```

pVertex = &m_pVertices_Orig[ ulCounter ];

// nur weitermachen wenn wir einen Bone haben
if( pVertex->fBone1 != 255.0f )
{
    // aktuelle Matrix holen
    matTemp.Identity();
    matTemp = m_pJoints[ (UINT)pVertex->fBone1 ]
                .sMatrix;

    // 1. Vertices rotieren
    sVector_A.x = pVertex->x;
    sVector_A.y = pVertex->y;
    sVector_A.z = pVertex->z;
    sVector_A -= matTemp.GetTranslation();
    sVector_A.InvRotateWith( matTemp );
    pVertex->x = sVector_A.x;
    pVertex->y = sVector_A.y;
    pVertex->z = sVector_A.z;

    // 2. Normalen rotieren
    sVector_A.x = pVertex->vcN[ 0 ];
    sVector_A.y = pVertex->vcN[ 1 ];
    sVector_A.z = pVertex->vcN[ 2 ];
    sVector_A.InvRotateWith( matTemp );
    pVertex->vcN[ 0 ] = sVector_A.x;
    pVertex->vcN[ 1 ] = sVector_A.y;
    pVertex->vcN[ 2 ] = sVector_A.z;
}
}
return S_OK;
}
// -----

```

Zum Ende der Methode hin können wir dann mit der frischen finalen Matrix die einzelnen Vertices positionieren. Dazu loopen wir einfach durch alle Vertices und prüfen, ob die einem Joint zugeordnet sind. Wenn dem so ist, dann holen wir uns die finale Matrix. Dann translieren wir den Vertex entsprechend der Matrix und führen eine inverse Rotation aus. Sie ist invers, weil wir sie dann direkt ausführen können. Das bedeutet, dass wir später einfach die anzuwendenden Rotationswinkel auf den Vertex loslassen können, weil diese im Master-Array quasi in einer Position ohne jegliche Rotation gespeichert ist.

Vertices

Nachdem wir dann den Vertex passend positioniert haben, nehmen wir uns noch fix den Normalenvektor. Hier gehen wir fast identisch vor. Jedoch translieren wir ihn nicht. Das können wir uns sparen, weil er ein Einheitsvektor ist und für die Lichtberechnung nur der Winkel, jedoch nicht die Position relevant ist.

Normals

Jetzt befinden sich alle Vertices inklusive der Normalenvektoren in der initialen Position. Jetzt können wir gleich mit den Animationen starten.

Bewegung im Modell (Animation)

Endlich!

Diese Methode wird pro Frame einmal aufgerufen. Sie berechnet auf Grund der vorliegenden ausgewählten Animation, welche exakte Frame-Position wir haben. Weiterhin sorgt sie für das Update der Vertices, so dass sie dann gerendert werden können.



```
// -----
// Name: Animation( void )
// Info: Animieren des Modells
//
// Return      = (HRESULT)    Status
// -----
HRESULT CZFXModel::Animation( void )
{
    // Variablen Init
    float      fElapsed   = -1.0f;    // Zeit
    float      fStart     = -1.0f;    // Start
    float      fEnd       = -1.0f;    // Ende
    LPANIMATION pAnimation = NULL;    // Animation

    // Haben wir Animationen?
    if( m_sHeader.uiNumJoints == 0 )
        return S_OK;

    // Einmaliger Durchlauf?
    if( m_bAnimationRunOnce &&
        m_bAnimationComplete &&
        !m_bAnimationChanged )
        return S_OK;

    ///////////////////////////////////
    // __FORTSETZUNG_FOLGT__ //
    ///////////////////////////////////
}
```

Es ist natürlich klar, dass, wenn wir keine Joints haben, auch keine Animation möglich ist. Daher springen wir gleich am Anfang raus, falls die Anzahl der Joints gleich 0 ist.

Loop

Wir sind natürlich in der Lage, eine Animation im Loop oder auch einzeln laufen zu lassen. Daher fragen wir als Nächstes ab, was wir machen wollen. Die Member-Variable `m_bAnimationRunOnce` wird durch die Methode `SetAnimation()` gesetzt. Sollte die also gesetzt sein (`true`), so wird geprüft, ob die aktuelle Animation komplett durchgelaufen ist. Dazu bedienen wir uns der Member-Variablen `m_bAnimationComplete`. Sie wird weiter unten in dieser

Methode gesetzt. Falls sie auf `true` steht, muss nur noch die Variable `m_bAnimationChanged` auf `false` stehen, und wir können sofort aus der Methode herausspringen. Das können wir machen, weil sich die Animation nicht verändert hat. Um die Animation erneut zu starten, müssten wir wieder einfach noch mal `SetAnimation()` aufrufen. Wenn wir beispielsweise mit unserem Charakter schießen, dann führen wir einmal die Animation aus, und wenn wir ein weiteres Mal schießen, dann führen wir die Animation ja auch erneut durch einen Aufruf der Methode `SetAnimation()` aus. Ist simpel, oder?

```

////////////////////////
// __FORTSETZUNG__ //
////////////////////////

// Zeit abfragen
m_fTime = (float)GetTickCount();

// falls neu ist das die Startzeit
if( m_fStartTime == -1.0f )    m_fStartTime =
                               m_fTime;

// vergangene Zeit berechnen
fElapsed = m_fTime - m_fStartTime;

// aktuelle Animation holen -----
pAnimation = &m_pAnimations[ m_uiCurrentAnimation ];

fStart = pAnimation->fStartFrame;    // Startzeit
fEnd   = pAnimation->fEndFrame;      // Endzeit

// aktuelle Frameposition berechnen
m_fFrame = fStart + (m_sHeader.fAnimationFPS / 2048)
            * fElapsed;

// Startframe setzen, wenn neu
if( m_fFrame <= fStart )    m_fFrame = fStart;

// Ende erreicht?
if( m_fFrame >= fEnd )
{
    m_fStartTime = m_fTime;        // Zeit setzen
    m_fFrame     = fStart;
    m_bAnimationComplete= true;    // Flag setzen
}
else
{
    // Animation vorbereiten
    AnimationPrepare();
}

```



```

        // Vertices Setup
        AnimationVertices();
        m_bAnimationComplete= false;    // set Flag
        m_bAnimationChanged = false;
    }
    return S_OK;
}
// -----

```

Gutes Timing ist elementar

Unerlässlich für die Animation ist die korrekte Zeit. Daher holen wir uns diese in die Membervariable `m_fTime`. Sollte die Startzeit `m_fStartTime` noch immer auf `-1.0f` stehen, so bedeutet das, dass wir noch keine einzige Animation durchgefahren haben. Wir setzen in diesem Fall die Startzeit auf die aktuelle Zeit, damit ist dann die vergangene Zeit in `fElapsed` auch gleich `0.0f`. Jetzt brauchen wir die aktuelle Animation. Sie steht in `m_uiCurrentAnimation`, und in dem Zeiger `pAnimation` speichern wir den Zeiger auf die aktuelle Animation, um einen einfacheren Zugriff zu haben. In der Struktur steht ja sowohl die Start- als auch die Endzeit.

Frame

Der aktuelle Frame wird als Nächstes berechnet und in die Variable `m_fFrame` gespeichert. Wenn wir die erste Animationsequenz fahren, dann ist die vergangene Zeit in `fElapsed` `0.0f`, und daher ist die `m_fFrame`-Variable mit der `fStart`-Variable identisch. Sollte dem nicht so sein, so befinden wir uns mitten in der Animation. Daher prüfen wir, ob der aktuelle Frame schon über oder gleich dem Ende der Animation in `fEnd` ist. Entsprechend initialisieren wir beim erreichten Ende die Variablen neu und setzen `m_bAnimationComplete` auf `true`, oder wir führen mitten in der Animation mit den beiden Methoden `AnimationPrepare()` und `AnimationVertices()` die eigentliche Animation aus. Zum Ende hin springen wir dann mit `S_OK` heraus.

Vorbereitung ist alles (AnimationPrepare)

Joint-Matrix berechnen

Hier berechnen wir nichts weiter als die korrekte Position zwischen den Frames. Für diesen kurzen Satz haben wir allerdings schon recht viel Quellcode vor dem Bauch. Wir werden die Rotation sowie die Translation des Frames berechnen. Anhand dieser Ergebnisse berechnen wir dann auch die neue finale Matrix für die Joints, mit denen die Vertices in der Hierarchie dann während der Animation entsprechend transformiert werden.



```

// -----
// Name: AnimationPrepare( void )
// Info: Bereitet Animationsequenz vor
//
// Return      = (HRESULT)      Status
// -----
HRESULT CZFXModel::AnimationPrepare( void )
{

```

```

// Variablen Init
LPJOINT    pJoint      = NULL;    // Joint
ULONG      ulCounter   = 0;       // Zähler
UINT       uiLoop      = 0;       // Zähler
ZFXVector  sPosition;           // Vektor
ZFXVector  sRotation;           // Vektor
UINT       uiKeyPos     = 0;       // Key-Position
UINT       uiKeyRot     = 0;       // Key-Position
LPKF_ROT   pLastRot     = NULL;    // Rotation
LPKF_ROT   pThisRot     = NULL;    // Rotation
LPKF_ROT   pKeyRot      = NULL;    // Rotation
LPKF_POS   pLastPos     = NULL;    // Position
LPKF_POS   pThisPos     = NULL;    // Position
LPKF_POS   pKeyPos      = NULL;    // Position
float      fScale       = 0.0f;    // Skalierung
ZFXMatrix  matTemp; matTemp.Identity();
ZFXMatrix  matFinal; matFinal.Identity();

// Clip der Animation
if( m_fFrame > m_sHeader.uiNumFrames ) m_fFrame = 0;

// Matrizen anheizen
for( ulCounter = 0; ulCounter <
    m_sHeader.uiNumJoints; ulCounter++ )
{
    // aktuellen Joint holen
    pJoint = &m_pJoints[ ulCounter ];

    // andere Daten holen
    uiKeyPos = pJoint->wNumKF_Position; // Position
    uiKeyRot = pJoint->wNumKF_Rotation; // Rotation
}
// _____
// __FORTSETZUNG_FOLGT__
// _____

```

Als Erstes fällt wohl der massiv große Block mit den Variablen auf. Allein für die Rotation im Frame sowie für die Position brauchen wir je drei, also sechs Strukturen. Eine kurze Erläuterung zu den Variablen:

pKey*** Diese Variable brauchen wir zum Suchen des aktuellen und des letzten Frames.

pLast*** Hier steht jeweils die letzte bekannte Rotation oder Position, quasi der Minimum-Frame.

pThis*** Dies ist der aktuelle Frame, in dem wir uns jetzt gerade befinden.

Wir müssen gleich zu Beginn prüfen, ob wir noch innerhalb der maximalen Anzahl der Frames sind.

Danach fangen wir an, die Matrizen zu berechnen. Dazu müssen wir als Erstes den aktuellen Joint in `pJoint` holen. Aus diesen Joints holen wir uns dann die Anzahl der Position und Rotation des Joints.



```

////////////////////
// __FORTSETZUNG__ //
////////////////////

// Neuberechnung nötig?
if( ( uiKeyRot + uiKeyPos ) != 0 )
{
    // Ja, neue Position oder Rotation
    pLastPos = NULL; pThisPos = NULL;
    pKeyPos = NULL;

    for( uiLoop=0; uiLoop < uiKeyPos; uiLoop++ )
    {
        // aktuelle Position holen
        pKeyPos = &pJoint->pKF_Position[uiLoop];

        // Zeit kontrollieren
        if( pKeyPos->fTime >= m_fFrame )
        {
            pThisPos = pKeyPos;

            break;
        }
        // nix gefunden
        pLastPos = pKeyPos;
    } // alle Positionen
}
////////////////////
// __FORTSETZUNG_FOLGT__ //
////////////////////

```

Wenn natürlich keine Rotation oder Translation anliegt, dann brauchen wir die ganze Arie hier nicht durchzurechnen.

In der Schleife loopen wir durch alle Positionen, bis wir diejenige gefunden haben, deren Zeitindex größer als unser aktueller Zeitindex ist. Dazu wird `pKeyPos->fTime` mit `m_fFrame` verglichen. Wenn der Zeitindex in `m_fFrame` kleiner ist, dann setzen wir `pThisPos` auf die aktuell ausgewählte Position und beenden die Schleife mit `break`. Sollte dem nicht so sein, so speichern wir `pKeyPos` in `pLastPos` und nehmen uns den nächsten Zeitindex vor.

Nach der Schleife haben wir in `pThisPos` die aktuelle Position und in `pLastPos` die Vorgängerposition. Damit können wir jetzt zwischen diesen beiden die Interpolation durchführen, um die exakte Rotation und Translation in die Matrix zu bekommen.



```

////////////////////
// __FORTSETZUNG__ //
////////////////////
    // interpoliere die beiden Positionen
    if( pLastPos && pThisPos )
    {
        // Skalierung berechnen
        fScale =
            ( m_fFrame - pLastPos->fTime )/
            ( pThisPos->fTime - pLastPos->fTime );

        // Interpolation
        sPosition =
            pLastPos->vPosition +
            ( pThisPos->vPosition -
              pLastPos->vPosition )
            * fScale;
    } else if( !pLastPos )
    {
        // copy the position
        sPosition = pThisPos->vPosition;
    } else {
        // copy the position
        sPosition = pLastPos->vPosition;
    }

    //////////////////////
    // __FORTSETZUNG_FOLGT__ //
    //////////////////////

```

Jetzt liegen uns die beiden Positionen vor. Also müssen wir jetzt den Faktor zwischen diesen beiden Positionen berechnen. Das Ergebnis speichern wir in `fScale`. Wenn wir diesen Faktor haben, können wir die exakte Position bestimmen. Dazu addieren wir zu unserer aktuellen Position die mit dem Faktor multiplizierte Differenz zwischen der letzten und der aktuellen Position.

Wenn wir allerdings nur die aktuelle Position (`pThisPos`) haben und nicht auch noch `pLastPos`, dann kopieren wir auf die interpolierte Position `sPosition` einfach die vorhandene. Sollte allerdings beides nicht vorhanden sein, dann kopieren wir die in diesem Fall mindestens vorhandene `pLastPos` auf die interpolierte `sPosition`. So einfach geht das.

```

////////////////////
// __FORTSETZUNG__ //
////////////////////
    // Rotationen durchführen-----
    pLastRot = NULL; pThisRot = NULL;
    pKeyRot  = NULL;

```



```

for( uiLoop=0; uiLoop < uiKeyRot; uiLoop++ )
{
    // aktuelle Rotation holen
    pKeyRot = &pJoint->pKF_Rotation[uiLoop];

    // Zeit prüfen
    if( pKeyRot->fTime >= m_fFrame )
    {
        pThisRot = pKeyRot;

        break;
    }
    // nix gefunden
    pLastRot = pKeyRot;
} // all Rotitions

// interpoliere zu den Rotationen
if( pLastRot && pThisRot )
{
    // interpoliere die Rotationen
    sRotation =
        pLastRot->vRotation +
        ( pThisRot->vRotation -
          pLastRot->vRotation )
        * fScale;
} else if( !pLastRot )
{
    // kopiere die Rotation
    sRotation = pThisRot->vRotation;
} else {
    // kopiere die Rotation
    sRotation = pLastRot->vRotation;
}

//////////
// __FORTSETZUNG_FOLGT__ //
//////////

```

Nachdem wir die Position berechnet haben, führen wir das Gleiche für die Rotation aus. Es ist nahezu alles identisch, bis auf die Daten selbst. Die interpolierte Rotation befindet sich am Ende in `sRotation`.



```

//////////
// __FORTSETZUNG__ //
//////////
// Joint Matrix Setup
matTemp.SetTranslation( sPosition );
matTemp.Rota( sRotation ); // Rotation

// relative Matrix berechnen
matFinal = matTemp *

```

```

        pJoint->sMatrix_relative;

        // Haben wir einen Parent gefunden?
        if( pJoint->wParentID != 255 )
        {
            // Parent gefunden, daher müssen wir
            // seine Absolut-Matrix mit unserer Relativ-
            // Matrix multiplizieren, um unsere Absolut-
            // Matrix zu erhalten!
            pJoint->sMatrix = matFinal *
                m_pJoints[ pJoint->wParentID ].sMatrix;

        } else {
            pJoint->sMatrix = matFinal;
        }
    } else {

        // keine neue Matrix => alte kopieren
        pJoint->sMatrix = pJoint->sMatrix_relative;
    }
}
return S_OK;
}
// -----

```

Wir haben in den Variablen `sPosition` und `sRotation` die aktuelle interpolierte Rotation und Position des Joints. Mit diesen Angaben können wir eine neue Matrix erstellen, die wir dann noch mit der relativen Matrix des Joints multiplizieren müssen, um eine neue finale Matrix zu bekommen. Sollte ein Parent-Joint vorhanden sein, so müssen wir die neue Matrix noch fix mit der finalen Matrix des Parent multiplizieren.

Finale Matrix

Damit haben wir in dem Joint nun eine exakte Matrix, die den Joint so rotiert und positioniert, dass er zwischen zwei Animationsframes exakt berechnet ist. Damit müssen wir jetzt nur noch die einzelnen Vertices bearbeiten, und fertig ist die korrekte Animation. Das war doch auch mal wieder erschreckend einfach, oder?

Meine Position (AnimationVertices)

Jetzt kommt ein echter Pfannkuchen. Wir werden hier in dieser Methode die mit Schweiß und Herzblut berechnete Matrix unserer Joints dazu verwenden, die zu ihr gehörenden Vertices zu positionieren. Um natürlich später in der Lage zu sein, auch eine korrekte Kollisionserkennung durchzuführen, werden wir in der Schleife auch gleich noch die neue und exakte Bounding-Box mitberechnen. Diese Methode hat Ähnlichkeiten mit der Methode `SetupBones()`. Also, let's go.

*Transformation
der Vertices in
jedem Frame*



```
// -----
// Name: AnimationVertices( void )
// Info: Berechne Vertices für aktuelle Animation
//
// Return      = (HRESULT)      Status
// -----
HRESULT CZFXModel::AnimationVertices( void )
{
    // Variablen init
    ULONG      ulCounter      = 0;          // Zähler
    CVERTEX     *pVertex      = NULL;       // temporär
    CVERTEX     *pVertex_Orig = NULL;       // temporär
    ZFXVector   sVector_A, sVector_B;      // Vektor

    // Bounding Box Reset
    m_sBBoxMin.x = 999999.0f; m_sBBoxMin.y = 999999.0f;
    m_sBBoxMin.z = 999999.0f; m_sBBoxMax.x = -999999.0f;
    m_sBBoxMax.y = -999999.0f; m_sBBoxMax.z = -999999.0f;

    // Setup der Vertices
    for( ulCounter = 0; ulCounter <
        m_sHeader.ulNumVertices; ulCounter++ )
    {
        // aktuellen Vertex holen
        pVertex      = &m_pVertices[ ulCounter ];
        pVertex_Orig = &m_pVertices_Orig[ ulCounter ];
        ///////////////////////////////////
        // __FORTSETZUNG_FOLGT__ //
        ///////////////////////////////////
    }
}
```

Master-Array und Arbeits-Array

Ein paar Variablen werden auch hier benötigt. Nachdem diese definiert sind, initialisieren wir die Bounding-Box. Dann gehen wir in die Schleife über alle vorhandenen Vertices. Zur Bearbeitung holen wir uns Zeiger auf die beiden Arrays, mit denen wir gleich intensiv arbeiten werden. Das originale Master-Array `m_pVertices_Orig` wird als lesendes und das Arbeits-Array `m_pVertices` als schreibendes Array genutzt. Aus dem Arbeits-Array heraus wird später auch gerendert.



```
////////////////////////////////////
// __FORTSETZUNG__ //
////////////////////////////////////
// nur berechnen falls Bone gefunden
if( pVertex->fBone1 != 255.0f )
{
    // 1. Originalvertex holen
    sVector_A.x = pVertex_Orig->x;
    sVector_A.y = pVertex_Orig->y;
    sVector_A.z = pVertex_Orig->z;
}
```



```

// 2. Rotiere den Vertex
sVector_A.RotateWith( m_pJoints[
    (UINT)pVertex_Orig->fBone1 ].sMatrix );

// 3. Position holen
sVector_A += m_pJoints[ (UINT)pVertex_Orig
    ->fBone1 ].sMatrix.GetTranslation();

// 4. Berechne neue Position
pVertex->x = sVector_A.x;
pVertex->y = sVector_A.y;
pVertex->z = sVector_A.z;

// 5. Normalen animieren
sVector_A.x = pVertex_Orig->vcN[ 0 ];
sVector_A.y = pVertex_Orig->vcN[ 1 ];
sVector_A.z = pVertex_Orig->vcN[ 2 ];
sVector_A.RotateWith( m_pJoints[
    (UINT)pVertex_Orig->fBone1 ].sMatrix );
pVertex->vcN[ 0 ] = sVector_A.x;
pVertex->vcN[ 1 ] = sVector_A.y;
pVertex->vcN[ 2 ] = sVector_A.z;
////////////////////
// __FORTSETZUNG_FOLGT__ //
////////////////////

```

Wir brauchen zwingend die Zugehörigkeit des Vertex zu dem Joint. Daher lesen wir `fBone1` aus, und wenn das eine gültige ID enthält, dann fangen wir mit der Berechnung an.

Aus dem Master-Array `pVertex_Orig` holen wir uns die Position des Vertex in den Vektor `sVector_A`. Danach rotieren wir ihn mit der finalen Matrix. Als Nächstes translieren wir den Vertex noch mit der Matrix und schreiben den neu positionierten Vertex in das Arbeits-Array `pVertex` zurück.

Wir dürfen für die korrekte Lichtberechnung nicht den Normalenvektor vergessen. Hier brauchen wir jedoch nur eine Rotation des Normalenvektors vornehmen, da er ein Einheitsvektor ist und durch ihn nur der Winkel zwischen dem Licht und der Ebene berechnet wird. Also ist die Position des Normalenvektors egal.

Normals

Natürlich gilt auch hier wieder: Wir lesen aus dem Master-Array und schreiben in das Arbeits-Array.



```

////////////////////
// __FORTSETZUNG__ //
////////////////////

        // 6. Bounding Box berechnen
        m_sBBoxMax.x = MAX(m_sBBoxMax.x,pVertex->x);
        m_sBBoxMax.y = MAX(m_sBBoxMax.y,pVertex->y);
        m_sBBoxMax.z = MAX(m_sBBoxMax.z,pVertex->z);
        m_sBBoxMin.x = MIN(m_sBBoxMin.x,pVertex->x);
        m_sBBoxMin.y = MIN(m_sBBoxMin.y,pVertex->y);
        m_sBBoxMin.z = MIN(m_sBBoxMin.z,pVertex->z);

    }

    // 7. Aabb erzeugen
    m_sAabb.vcMin.x = m_sBBoxMin.x;
    m_sAabb.vcMin.y = m_sBBoxMin.y;
    m_sAabb.vcMin.z = m_sBBoxMin.z;
    m_sAabb.vcMax.x = m_sBBoxMax.x;
    m_sAabb.vcMax.y = m_sBBoxMax.y;
    m_sAabb.vcMax.z = m_sBBoxMax.z;
    m_sAabb.vcCenter.x =(m_sBBoxMax.x-m_sBBoxMin.x) / 2;
    m_sAabb.vcCenter.y =(m_sBBoxMax.y-m_sBBoxMin.y) / 2;
    m_sAabb.vcCenter.z =(m_sBBoxMax.z-m_sBBoxMin.z) / 2;
    return S_OK;
}

// -----

```

Zum Ende der Schleife hin nutzen wir die beiden Makros *MAX* und *MIN*, um die Bounding-Box zu erweitern. Nach der Schleife setzen wir die Aabb in der Variablen *m_sAabb*. Fix wird noch der Mittelpunkt berechnet, und fertig ist die Animation.

8.4 Updaten und Nutzen des Modells

Wir haben jetzt alle Daten des Modells geladen, vorbereitet und können die Positionen der einzelnen Vertices auf den Punkt genau innerhalb der Frames berechnen. Was wir jetzt noch brauchen, sind kleinere Methoden, um das Modell auf dem aktuellen Stand zu halten und natürlich zu rendern. Ein paar kleinere Methoden zur Kontrolle des Modells dürfen natürlich auch nicht fehlen.

Aktueller Stand (Update)

Das Wichtigste für unser Modell ist die Zeit, und diese übergeben wir der Methode. Der Aufruf in unserem Hauptprogramm sieht ungefähr so aus:

```
// tu etwas
m_pModel->Update( fDeltaTime );
m_pModel->Render();
```

Es wird also das Delta an Zeit zwischen dem letzten Aufruf und jetzt übergeben. Hier seht ihr die kleine Methode:

```
// -----
// Name: Update( float fTime )
// Info: Update des Modells
//
// fTime      = Deltatime
//
// Return     = (HRESULT)   Status
// -----
HRESULT CZFXModel::Update( float fTime )
{
    // Zeit setzen
    m_fTime = fTime;

    // Animation durchführen
    return Animation();
}
// -----
```



Die Animation wird pro Frame durch die Methode *Animation()* vorbereitet, so dass wir unser Modell immer sofort rendern könnten, wenn wir wollen.

Präsentationstermin (Render)

Der wirklich einzige Beweggrund, warum wir das Ganze hier machen, ist diese Methode – unser Render-Aufruf. Da wir uns vorher die Mühe gemacht haben, alle Daten nach ihren Materialien zu sortieren, können wir jetzt beim Render-Aufruf diese Vorarbeit in echte Performance ummünzen.

```
// -----
// Name: Render( void )
// Info: Rendere das Modell
//
// Return     = (HRESULT)   Status
// -----
HRESULT CZFXModel::Render( void )
{
    // Variablen Init
    UINT      uiCounter = 0;           // Zähler

    // Culling setzen
    m_pRenderDevice->SetBackfaceCulling(RS_CULL_CCW);
```



```

// Vertex-Buffer rendern
for( uiCounter = 0; uiCounter <
    m_sHeader.uiNumMaterials; uiCounter++ )
    if( FAILED( m_pRenderDevice->
        GetVertexManager()->Render(
            VID_CA,
            m_sHeader.uiNumVertices,
            m_puiNumIndices[ uiCounter ],
            m_pVertices,
            (PWORD)m_ppIndices[ uiCounter ],
            m_puiSkinBuffer[ uiCounter ]
        ) ) )
        LOG( 1, true, "ERROR Failed to Render VB:
            %d [%d]", m_puiSkinBuffer[
                uiCounter ], uiCounter );

// Anderen Kram rendern?
if( m_bRenderBones )    RenderBones(); // Bones
if( m_bRenderNormals )  RenderNormals();// Normals
return S_OK;
}
// -----

```

Culling Als Erstes wird das Backface-Culling auf *CounterClockWise* gestellt – nicht, dass wir auf einmal das Innenleben des Modells sehen :-). Danach erfolgt der Render-Aufruf pro Material. Der Render-Methode der Engine übergeben wir das Vertexformat *VID_CA*, die Anzahl der Vertices gesamt, die Anzahl der Indices für dieses Material, einen Zeiger auf die (Arbeits-)Vertices, einen Zeiger auf die Indices zu den Vertices und einen Zeiger auf die aktuelle SkinID. Sollte das fehlschlagen, melden wir das mit einem Fehler.

Wenn dann noch die Bones oder die Normalenvektoren gerendert werden sollen, prüfen wir deren Flags und springen in die einzelnen Render-Methoden hinein.

Bis auf die Knochen runter (RenderBones)

Was wirklich interessant zu beobachten ist, sind die Bones – also eigentlich nur die Verbindung zwischen den Joints in Aktion. Die Bones sind einfach kleine Triangels, die jeweils von einem Joint zum nächsten Joint gerendert werden. Dies verdeutlicht Abbildung 8.7.



Abbildung 8.7:
Der Zombie geht in Richtung seines Opfers. Sehr schön sind hier die mitgerenderten Bones im Modell zu erkennen. Interessant ist auch der Hilfsbone zwischen den Beinen, auf dem das Modell wie eine Marionette steckt. Mit Hilfe dieses Bones kann man das gesamte Modell in einer schönen Animationssequenz nach hinten schleudern und umfallen lassen. (Quelle für das Modell: <http://www.psionic-3d.co.uk>)

```
// -----
// Name: RenderBones( void )
// Info: Render die Bones für Debug
//
// fTime      = Deltatime
//
// Return     = (HRESULT)   Status
// -----
HRESULT CZFXModel::RenderBones( void )
{
    // Variablen Init
    UINT      uiCounter = 0;           // Zähler
    LVERTEX   pLine[3];               // Joints
    WORD      pIndis[3] = { 0, 1, 2 }; // Indices
    DWORD     dwColor   = 0x00ffff;   // Farbe

    // Haben wir Bones?
    if( m_sHeader.uiNumJoints == 0 ) return S_OK;

    // Render-States
    m_pRenderDevice->SetBackfaceCulling(RS_CULL_NONE);
    m_pRenderDevice->SetDepthBufferMode(RS_DEPTH_NONE);

    // Rendere die Bones
    for( uiCounter = 0; uiCounter <
        m_sHeader.uiNumJoints; uiCounter++ )
    {
        // Erster Vertex
        pLine[0].x = m_pJoints[ uiCounter ].sMatrix._41;
        pLine[0].y = m_pJoints[ uiCounter ].sMatrix._42;
        pLine[0].z = m_pJoints[ uiCounter ].sMatrix._43;
```



```

pLine[0].Color = dwColor;

if( m_pJoints[ uiCounter ].wParentID != 255 )
{
    // Zweiter Vertex
    pLine[1].x = m_pJoints[ m_pJoints[
    uiCounter ].wParentID ].sMatrix._41;
    pLine[1].y = m_pJoints[ m_pJoints[
    uiCounter ].wParentID ].sMatrix._42;
    pLine[1].z = m_pJoints[ m_pJoints[
    uiCounter ].wParentID ].sMatrix._43;
    pLine[1].Color = dwColor;

    // Dritter Vertex
    pLine[2].x = pLine[1].x + 1.0f;
    pLine[2].y = pLine[1].y + 1.0f;
    pLine[2].z = pLine[1].z + 1.0f;
    pLine[2].Color = dwColor;

    // Render
    m_pRenderDevice->GetVertexManager()->
    Render(VID_UL, 3, 3, pLine, pIndis, 0);
}
// Render-States
m_pRenderDevice->
    SetDepthBufferMode(RS_DEPTH_READWRITE);
m_pRenderDevice->
    SetBackfaceCulling(RS_CULL_CCW);
return S_OK;
}
// -----

```

Am Anfang prüfen wir wie gewohnt, ob Joints vorhanden sind. Danach müssen wir die Engine einstellen, dass wir ohne Culling und ohne Depth-Buffer rendern wollen. Ansonsten würden wir nicht sehr viel von den Bones sehen. Danach wandern wir einfach von Joint zu Joint und erstellen die notwendigen drei Vertices für unser Triangel. Dabei kommen zwei jeweils vom Joint und seinem Parent-Joint, und der dritte wird von uns erstellt, indem wir einfach alle Koordinaten vom aktuellen Joint + 1.0f nehmen.

Ist der Bone, also das Triangel erstellt, können wir es dann auch rendern. Da wir ordentlich sind, setzen wir noch die Engine auf Standard-Settings zurück.

Voll normal (RenderNormals)

Damit ist kein Film gemeint, sondern die Normalenvektoren der Vertices, die gerendert werden sollen. Da wir für jeden Vertex einen Normalenvektor

haben, können wir einfach vom Ursprung des Vertex in Richtung des Normalenvektors eine Linie rendern, und das ist dann unser Normalenvektor. Klingt einfach, ist einfach und sieht fluffig aus.

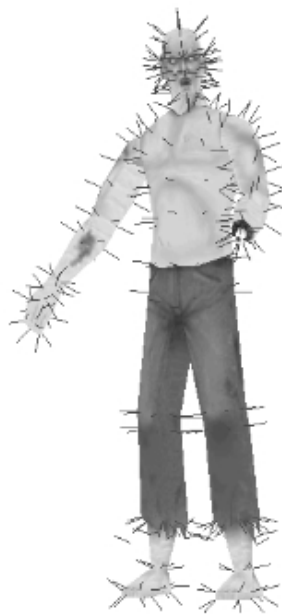


Abbildung 8.8:

Ob sich unser Zombie so wohlfühlt? Ich glaube ja :-), zumindest sieht er schön scheußlich aus. (Quelle für das Modell: <http://www.psionic3d.co.uk>)

Auf in den einfachen Quellcode:

```
// -----
// Name: RenderNormals( void )
// Info: Rendere die Normalenvektoren (Debug)
//
// fTime      = Deltatime
//
// Return      = (HRESULT)    Status
// -----
HRESULT CZFXModel::RenderNormals( void )
{
    ULONG    ulCounter = 0;
    float    fStart[3] = { 0,0,0 };
    float    fEnd[3]   = { 0,0,0 };
    ZFXCOLOR sColor    = { 1.0f, 0, 0, 0 };
    CVERTEX* pVertex   = NULL;

    // Bones rendern
    for( ulCounter = 0; ulCounter <
        m_sHeader.ulNumVertices; ulCounter++ )
    {
```



```

// aktuellen Vertex holen
pVertex = &m_pVertices[ uCounter ];

// Startpunkt setzen
memcpy( fStart, &pVertex->x,sizeof(float) * 3 );

// Endpunkt setzen
fEnd[0] = fStart[0] + (pVertex->vcN[0] * 2.0f);
fEnd[1] = fStart[1] + (pVertex->vcN[1] * 2.0f);
fEnd[2] = fStart[2] + (pVertex->vcN[2] * 2.0f);

// Normalenvektor rendern
m_pRenderDevice->GetVertexManager()->RenderLine(
    fStart, fEnd, &sColor );
}
return S_OK;
}
// -----

```

Ja, wie schon angedroht ist, diese Methode erschreckend einfach. Wir laufen einfach durch jeden Vertex und nehmen den Vertex als Startposition und einen auf dem Normalenvektor entlang berechneten Punkt als Endpunkt der Linie, die wir als Normalenvektor rendern. So einfach kann es gehen.

8.5 Ein Blick zurück, zwei Schritt nach vorn

Skinned Mesh und mehr

Im Verlauf dieses Kapitels haben wir gelernt, wie man ein so genanntes *Skinned Mesh* implementieren kann. Wir können nun animierte Modelle laden, eine bestimmte Animation ablaufen und das Modell dabei rendern lassen. Auch wenn hier der Fokus auf animierten Charakteren lag, so sollte man sich natürlich nicht nur allein darauf beschränken. Mit exakt derselben Technik kann man auch animierte Maschinen, Türen, Zugbrücken oder was auch immer erstellen und diese in seinen Levels platzieren. Eine Animation würde dann beispielsweise für eine Maschine kontinuierlich laufen und bei einer Tür oder einer Zugbrücke durch einen Schalter aktiviert werden.

Demo-Applikation

Natürlich gibt es auf der CD-ROM zu diesem Buch auch eine Demo-Applikation, die ein Modell im CBF-Format lädt und dessen Animationssequenzen darstellt. Diesen Projektbereich solltet ihr nun einmal öffnen und euch das Programm genau anschauen. Dann wird euch die Verwendung der Klasse `CZFXModel` schon bald in Fleisch und Blut übergegangen sein.

Move on

Im nächsten Kapitel wird es nun etwas dröge. Da geht es nur wieder um eher langweiligen Interface-Kram, den wir aber dennoch brauchen. Wir werden als Nächstes lernen, wie wir den Input des Spielers ohne die WinAPI abfragen können, und ein Input-Device für unsere Engine definieren und dieses dann auf Basis von `DirectInput` implementieren.

Teil 3 Hilfsmodule für die Engine

Kapitel 9:	Eingabe-Interface der ZFXEngine	471
Kapitel 10:	Audio-Interface der ZFXEngine	505
Kapitel 11:	Netzwerk-Interface der ZFXEngine	523
Kapitel 12:	Timing und Bewegung in der ZFXEngine	589
Kapitel 13:	Scene-Management	605

9 Eingabe-Interface der ZFXEngine

»Schämen sollten sich die Menschen, die sich gedankenlos der Wunder der Wissenschaft und Technik bedienen und nicht mehr davon geistig erfasst haben als die Kuh von der Botanik der Pflanzen, die sie mit Wohlbehagen frisst.« (Albert Einstein)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Initialisierung von DirectInput
- ➔ Implementierung einer DLL zur Kapselung des Inputs
- ➔ Tastatur, Maus und Joystick abfragen

9.1 Kurz und schmerzlos

Schon Einstein sagte, dass wir uns schämen sollten, wenn wir uns die Technik nur zu Nutze machen, aber nicht versuchen zu verstehen, warum was wie funktioniert. Man könnte dagegenhalten, dass der Anwender eines Tools gar nicht wissen muss, warum und wie es funktioniert. Das Tool müsste ja nur so angelegt sein, dass man es intuitiv bedienen kann, was auch immer für ein Tool das ist. Da wir uns aber auf der technischen Seite der Tools befinden und auch an deren Entwicklung Anteil nehmen, müssen wir uns schon ein wenig mit der Technik auskennen, die wir zum Einsatz bringen.

Einstein

In diesem Kapitel lernen wir daher DirectInput kennen – die Komponente von DirectX, mit der man den Input des Users abfragen kann. Ich setze hier voraus, dass ihr bereits über ein wenig Wissen im Bereich DirectInput verfügt, und werde daher die Initialisierung und Abfrage der Eingabe-Objekte zwar komplett beschreiben, aber nicht ausführlich erläutern. Wir werden in diesem Kapitel mehr Wert darauf legen, wie wir diese Eingabe-Objekte sinnvoll in das Design unserer Engine einbringen können.

Altbekanntes Interface-Design

Engine-Aufbau

Genau wie im dritten Kapitel erstellen wir auch hier eine Interface-Struktur und eine Implementierung dieses Interfaces in einer DLL, die über eine statische LIB geladen werden kann. Der Grundaufbau des Projekts entspricht dem aus dem dritten Kapitel. Das Projekt mit der statischen Bibliothek nennen wir `ZFXInput`, das Projekt mit der dynamischen Bibliothek erhält die Bezeichnung `ZFXDI` für `DirectInput`. Die folgenden Auflistungen zeigen die Dateien, die wir jeweils in den Projekten anlegen.

Dateien für `ZFXInput`:

- ➔ `ZFXInput.h` und `ZFXInput.cpp`
- ➔ `ZFXInputDevice.h`
- ➔ `ZFX.h`
- ➔ `ZFX3D.h`

Die ersten beiden Dateien beinhalten die Implementierung der statischen Bibliothek, und `ZFXInputDevice.h` ist die Definition des Interfaces. Der Header `ZFX.h` ist unsere große Wühlkiste für alle global exponierten Definitionen (z.B. Fehlermeldungen), Strukturen usw. `ZFX3D.h` ist der Header unserer 3D-Mathe-Bibliothek. Diese benötigen wir in diesem Kapitel zwar nicht, aber in `ZFX.h` verwenden einige Strukturen auch Klassen unserer 3D-Mathe-Bibliothek, daher binden wir den Header hier mit ein. Den Code der statischen Bibliothek werden wir hier nicht durchgehen, da wir dieses System der DLL-Anbindung bereits im dritten Kapitel ausführlich besprochen haben. Den kompletten Quellcode findet ihr aber natürlich auf der CD-ROM zu diesem Buch.

Dateien für `ZFXDI`:

- ➔ `ZFXDI.def`, `ZFXDI.h` und `ZFXDI.cpp`
- ➔ `ZFXKeyboard.cpp`
- ➔ `ZFXMouse.cpp`
- ➔ `ZFXJoystick.cpp`

Die ersten drei Dateien beinhalten hauptsächlich die Implementierung des Interfaces `ZFXInputDevice.h`. Das schauen wir uns in diesem Kapitel auch detailliert an. Die folgenden drei Dateien zeigen aber, dass wir für Tastaturen, Mäuse und Joysticks eigene Klassen haben werden. Auch dazu kommen wir weiter unten noch ausführlich.

Interface-Definition für eine Eingabe-Klasse

Auch hier verwenden wir wieder die Methode der Definition eines Interfaces, also einer rein virtuellen Klasse, um die Eingabe und deren Auswertung in der Engine zu kapseln. Dazu implementieren wir das Interface als Klasse in einer DLL und verwenden DirectInput Version 8¹ dafür. Durch unser Interface/DLL-System kann man diese Implementierung aber beliebig durch eine anderer Version austauschen, falls man eine andere API verwenden möchte. Als Erstes definieren wir einen Enumerationstyp für die drei Eingabegeräte, die wir unterstützen werden. Dann definieren wir gleich das Interface mit den Methoden, die wir von einer Input-Klasse erwarten.

*Baukasten-
System*

```
// in ZFX.h
typedef enum ZFXINPUTDEV_TYPE {
    IDV_KEYBOARD,
    IDV_JOYSTICK,
    IDV_MOUSE
} ZFXINPUTDEV;

// in ZFXInputDevice.h
class ZFXInputDevice {
protected:
    HWND      m_hWndMain;
    HINSTANCE m_hDLL;
    bool      m_bRunning;

public:
    ZFXInputDevice(void) {};
    virtual ~ZFXInputDevice(void) {};

    virtual HRESULT Init(HWND, RECT, bool)=0;
    virtual void Release(void)=0;
    virtual bool IsRunning(void)=0;
    virtual bool HasJoystick(char*)=0;
    virtual HRESULT Update(void)=0;

    // funktioniert nur mit Maus/Joystick
    virtual HRESULT GetPosition(ZFXINPUTDEV,
                               POINT*)=0;

    // funktioniert mit Tastatur, Maus und Joystick
    virtual bool IsPressed(ZFXINPUTDEV, UINT)=0;

    // funktioniert mit Tastatur, Maus und Joystick
    virtual bool IsReleased(ZFXINPUTDEV, UINT)=0;
}; // class
```



¹ Wir arbeiten zwar mit DirectX Version 9, aber in dieser Version wurde das DirectInput-Interface nicht verändert und besteht weiterhin in der alten Version 8.

Interface-Design

Hervorzuheben ist hier, dass unser Interface von der Struktur her einfach nur Eingaben kennt und nicht nach außen hin zwischen Tastatur, Maus und Joystick unterscheidet. Die letzten drei Funktionen im Interface dienen zum Beispiel dazu, die Zustände der verfügbaren Eingabegeräte abzufragen, beispielsweise der Tastaturtasten, Maustasten oder Joystick-Tasten. Die Unterscheidung, von welchem Gerät man Eingaben abfragen möchte, wird nicht in verschiedenen Funktionen, sondern in der Parameterliste über den Enumerationstyp `ZFXINPUTDEV`. Dieser Aufbau hält unser Interface dadurch schlank, dass der Anwender nur mit wenigen Methoden arbeiten muss.

Intern wird die Implementierung natürlich zwischen Tastatur, Maus und Joystick unterscheiden und dafür sogar verschiedene Klassen verwenden. Aber das interessiert denjenigen nicht, der nur mit der *ZFXEngine* arbeiten möchte. Den Anwender unserer DLL interessiert ja letzten Endes nicht, ob es da irgendwo ein Maus- oder Joystick-Objekt gibt. Er muss nur abfragen können, ob irgendeine Eingabeaktion wie z.B. ein Mausklick oder das Drücken eines Joystick-Buttons stattgefunden hat.

9.2 Basisklasse für DirectInput-Devices

Tastatur = Maus
= Joystick

Eine Maus ist eine Maus ist eine Maus. Oder etwa doch nicht? In seiner Komponente `DirectInput` kennt `DirectX` zunächst nur das Interface `IDirectInputDevice8`, das ein Eingabegerät darstellt. Ob dies nun eine Tastatur, eine Maus oder ein Joystick ist, interessiert zunächst recht wenig. Die Initialisierung und Freigabe eines solchen Eingabegeräts mit `DirectInput` läuft für alle Arten von Geräten gleich ab, abgesehen von genau zwei Funktionsparametern. Daher kapseln wir diese Funktionalität in einer Basis-klasse für Eingabegeräte namens `ZFXDDevice`.



```
class ZFXDDevice {
public:
    ZFXDDevice(void) { /*nix*/; }
    ~ZFXDDevice(void) { /*nix*/; }

    // Basis-Methoden
    virtual void Create(LPDIRECTINPUT8, HWND, FILE*);
    virtual void Release(void);
    virtual HRESULT CrankUp(REFGUID rguid,
                           LPCDDATAFORMAT lpdf);

    // Accessor-Methoden
    virtual void GetPosition(POINT *pPoint)
        { (*pPoint).x = m_lX; (*pPoint).y = m_lY; }

    // Virtuelle Methoden zur Vererbung
    virtual HRESULT Init(void)=0;
    virtual HRESULT Update(void)=0;
```

```
protected:
    virtual HRESULT GetData(ZFINPUTDEV Type,
                           void *pData,
                           DWORD *dwNum);

    LPDIRECTINPUTDEVICE8 m_pDevice;
    LPDIRECTINPUT8       m_pDI;
    HWND                 m_hWnd;
    long                 m_lX;
    long                 m_lY;
    FILE                 *m_pLog;
}; // class
```

Den Konstruktor und Destruktor dieser Klasse benötigen wir nicht, da wir ja keine Instanz dieser Klasse wirklich anlegen. Wir leiten später jeweils eine Klasse für Tastaturen, für Joysticks und für Mäuse von dieser Basisklasse ab. Diese erhalten dann valide Konstruktoren und Destruktoren. Die Funktionen `Create()`, `CrankUp()` und `Release()` sind jedoch für alle drei Eingabegerätetypen mehr oder weniger identisch, daher implementieren wir sie in der Basisklasse. Über `GetPosition()` geben wir dem Anwender die Attribute `m_lX` und `m_lY`, die bei einer Maus die Mauskoordinaten und bei einem Joystick die Werte auf der X- und Y-Achse angeben. Bei Tastaturen haben diese Attribute keine Bedeutung.

Basisklasse

Die Funktion `GetData()` dient dazu, die aktuellen Zustände eines DirectInput-Device-Objekts abzufragen und in bestimmten Strukturen zu speichern. Auch diese Funktion ist für Tastatur und Joystick fast identisch und für Mäuse recht ähnlich. Daher implementieren wir diese Funktion auch in der Basisklasse. Im Folgenden schauen wir uns die Implementierung an, bevor wir zu den abgeleiteten Klassen kommen.

Erstellen und Freigeben des Objekts

Wie schon so oft bei unseren bisherigen Implementierungen ist das Erstellen und Freigeben bestimmter Objekte die einfachste aller Aufgaben. Wir müssen dabei unbedingt daran denken, alle Pointer-Attribute mit `NULL` zu initialisieren. Tun wir dies nicht, so ist das im Debug-Kompilat nicht so tragisch, weil der Compiler dies dann automatisch tut. Aber wenn man eine Release-Version kompiliert, zwingt man das Programm quasi zum Absturz, wenn man auf einen Pointer zugreift, der weder auf validen Speicher zeigt noch ordentlich *ausgenullt* wurde.

*Erstellung einer
Instanz*



```
void ZFXDIDevice::Create(LPDIRECTINPUT8 pDI, HWND hWnd,
                        FILE* pLog) {
    m_pLog    = pLog;
    m_hWnd    = hWnd;
    m_pDI     = pDI;
    m_pDevice = NULL;
}
```

DirectInput- Hauptobjekt

Der Create()-Funktion müssen wir auf alle Fälle einen Pointer auf ein bereits initialisiertes IDirectInput8-Objekt übergeben. Dieses Objekt brauchen wir, um DirectInput-Device-Objekte erstellen zu können, und wir legen es entsprechend in der Implementierung unseres Interfaces in der Klasse ZFXDI an. Für den Moment akzeptieren wir einfach, dass das Objekt an dieser Stelle bereits betriebsbereit ist.

Unacquire

In der Release()-Funktion der Klasse tun wir nichts weiter, als das eventuell initialisierte IDirectInputDevice8-Objekt freizugeben. Wichtig ist dabei, dass wir auf alle Fälle zuerst noch die DirectInput-Methode IDirectInputDevice8::Unacquire aufrufen. Diese sorgt dafür, dass wir unseren Anspruch auf ein Eingabegerät wie beispielsweise die Maus aufgeben, so dass die anderen Anwendungen, die unter Windows laufen, auch wissen, dass sie nun keine Ressourcen mit uns zu teilen haben.



```
void ZFXDIDevice::Release(void) {
    if (m_pDevice) {
        m_pDevice->Unacquire();
        m_pDevice->Release();
        m_pDevice = NULL;
    }
    // Release
}
```

Inbetriebnahme

Eins, zwei, drei, und das Device eilt herbei

Jetzt können wir für das ZFXDI-Attribut vom Typ IDirectInputDevice8* Gefechtsbereitschaft herstellen. Unabhängig davon, ob das Interface eine Tastatur, eine Maus oder einen Joystick für uns anbinden soll, sind dazu nur drei Funktionen von DirectInput notwendig. Um was für ein Eingabegerät es sich dabei handelt, bestimmen die Parameter bei den ersten beiden Funktionen:

```
IDirectInput8::CreateDevice(
    REFGUID rguid,
    LPDIRECTINPUTDEVICE8 *ppDevice,
    LPUNKNOWN *pUnkOuter);

IDirectInputDevice8::SetDataFormat(
    LPCDIDATAFORMAT lpdf);
```



```
IDirectInputDevice8::SetCooperativeLevel(
    HWND hwnd,
    DWORD dwFlags);
```

Bei der ersten Funktion geben wir als ersten Parameter eine Referenz auf einen *Global Unique Identifier* an. Dabei handelt es sich um eine 128 Bit große Zahl, die ein Eingabegerät eindeutig identifiziert. Diese ID eines Eingabegeräts kann man durch Enumeration aller angeschlossenen Geräte feststellen. Für die Maus und die Tastatur ist das jedoch nicht nötig, da DirectInput hier mit GUID_SysKeyboard und GUID_SysMouse bereits Werte für die Default-Tastatur und die Default-Maus anbietet, die unter Windows aktiv sind. Der zweite Parameter der Funktion ist ein Zeiger auf die Adresse, an der das neu erstellte Device gespeichert werden soll, und den dritten Parameter benötigen wir nicht.

CreateDevice

Damit wir später die Werte eines Eingabegeräts korrekt abfragen können, müssen wir DirectInput auch mitteilen, in welchem Format die Daten für das Gerät vorliegen werden. Dazu kann eine Applikation eine eigene Variable der Struktur DIDATAFORMAT einstellen und verwenden. Allerdings bietet DirectInput hier bereits vordefinierte globale Objekte an, wie c_dfDIKeyboard, c_dfDIMouse und c_dfDIJoystick, die wir auch verwenden werden.

SetDataFormat

Für die Kooperation mit Windows müssen wir noch festlegen, zu welchem Fenster wir den Input abfragen wollen. Dazu geben wir im ersten Parameter der dritten Funktion das entsprechende Handle an. Im zweiten Parameter können wir über verschiedene Flags die Art der Kooperation festlegen. Eine Anwendung kann beispielsweise Exklusiv-Rechte auf bestimmte Eingabegeräte verlangen, so dass andere Anwendungen diese nicht verwenden können. Eine etwas kooperativere Politik macht jedoch in der Regel mehr Sinn, daher verwenden wir die beiden Flags DISCL_FOREGROUND und DISCL_NONEXCLUSIVE. So erhalten wir kooperativen Zugriff auf die Eingabegeräte, während das zugehörige Fenster im Vordergrund aktiv ist.

*SetCooperative-
Level*

Das war also der Schnellkurs in der Initialisierung von DirectInput. Und hier ist die konkrete Funktion, die das für uns bewerkstelligt:

```
HRESULT ZFXDDevice::CrankUp(REFGUID rguid,
                             LPCDIDATAFORMAT pdf) {
    DWORD dwFlags = DISCL_FOREGROUND |
                    DISCL_NONEXCLUSIVE;

    // falls es bereits existiert, zerstöre es
    if (m_pDevice) {
        m_pDevice->Unacquire();
        m_pDevice->Release();
        m_pDevice = NULL;
    }
```



```

// 1. Schritt: Erstelle das Device
if ( FAILED(m_pDI->CreateDevice(rguid, &m_pDevice,
                                NULL)))

    return ZFX_FAIL;

// Definiere das richtige Datenformat
if ( FAILED(m_pDevice->SetDataFormat(pdf)))
    return ZFX_FAIL;

// Kooperationslevel mit Windows festlegen
if ( FAILED(m_pDevice->SetCooperativeLevel(m_hWnd,
                                           dwFlags)))

    return ZFX_FAIL;

return ZFX_OK;
} // CrankUp

```

*Tastatur, Maus
und Joystick
kompatibel*

Diese Funktion kann damit alle Arten von Eingabegeräten erstellen. Dazu müssen wir nur die entsprechende GUID und das Datenformat über die Parameter angeben. Das tun wir nachher bei der konkreten Initialisierung der abgeleiteten Klassen für Tastaturen, Mäuse und Joysticks. Aber durch diese Funktion kann man sehr schön demonstrieren, wie man sich Basis-klassen zunutze machen kann, die identische Arbeitsabläufe für ähnliche Objekte kapseln.

Abfrage des Inputs

Analogien

Ähnlich verhält es sich bei der Abfrage des Zustands oder der Daten eines Eingabegeräts. Auch hier muss man bei DirectInput, vereinfacht gesagt, lediglich anhand der Parameter unterscheiden, ob es sich um eine Tastatur, eine Maus oder einen Joystick handelt, aber ansonsten das gleiche Abfrage-Schema verwenden. Daher können wir diese Abfrage auch gut in der Basis-klasse kapseln, um uns redundante Implementierungen in den abgeleiteten Klassen zu sparen.

*DirectInput-
Device-Abfrage-
Funktionen*

Um ein Eingabegerät abzufragen, bietet DirectInput zwei Funktionen an, weil man ein Eingabegerät mit DirectInput in zwei verschiedenen Modi betreiben kann: zum einen als gepuffertes Gerät (*buffered data*) und zum anderen natürlich auch als ungepuffertes Gerät (*immediate data*). Wenn man einen Puffer für ein Gerät verwendet, speichert DirectInput sozusagen Events in dem zugehörigen Datenpuffer, die man dann alle abfragen kann. In diesem Puffer werden dann so lange Events gespeichert, bis wir sie abfragen und den Puffer dadurch leeren. Auf diese Weise verpassen wir auf keinen Fall irgendeine Aktion eines Eingabegeräts, auch wenn wir ein Gerät in einem Frame einmal nicht abfragen. Bei ungepufferten Eingabegeräten hingegen fragen wir lediglich States ab, also den Zustand, in dem sich ein Gerät

aktuell befindet. So können wir nicht herausfinden, ob seit der letzten Abfrage eine Zustandsänderung eingetreten ist, die aber inzwischen schon wieder obsolet ist. Dadurch kann man bestimmte Eingaben *verpassen*.

Wir verwenden hier nur für die Maus eine gepufferte Eingabe, um auch diese Technik zu demonstrieren. Für den Joystick und die Tastatur reicht uns eine ungepufferte Eingabe. Da wir die Eingabegeräte sowieso in jedem Frame abfragen, ist es unwahrscheinlich, dass wir eine Eingabe verpassen. Das würde beispielsweise nur dann geschehen, wenn der Zeitraum zwischen zwei Abfragen eines Eingabegeräts (also die Zeit zwischen zwei Frames) länger ist, als die Eingabe an dem Gerät erfolgt, beispielsweise wenn der Anwender eine Taste eine halbe Sekunde lang drückt, wir aber nur einmal in der Sekunde die Tastatur abfragen und das auch genau dann, wenn die Taste gerade nicht gedrückt ist. Das ist verhältnismäßig unwahrscheinlich – eine flüssige Frame-Rate vorausgesetzt –, und daher reicht ungepufferte Eingabe aus.

*Heute gibt's
Mäuse-Puffer*

Nun aber zu den beiden oben angesprochenen Funktionen; Wenn wir mit einem gepufferten Gerät arbeiten, dann können wir von diesem die Daten seines Puffers abfragen. Wenn wir mit einem ungepufferten Gerät arbeiten, dann können wir nur die States, also die aktuellen Zustände des Geräts, abfragen. Dazu bietet DirectInput die folgenden beiden Funktionen:

*GetDeviceData
und GetDevice-
State*

```
IDirectInputDevice8::GetDeviceData(  
    DWORD    cbObjectData,  
    LPDIDEVICEOBJECTDATA rgdod,  
    LPDWORD  pdwInOut,  
    DWORD    dwFlags);
```

```
IDirectInputDevice8::GetDeviceState();
```

Der erste Parameter der ersten Funktion gibt die Größe der Datenstruktur im zweiten Parameter an, wobei es sich um ein Array von `DIDEVICEOBJECTDATA`-Objekten handelt. In jedem Element dieses Arrays wird dann je ein Event aus dem Puffer des Geräts gespeichert. Der dritte Parameter gibt zunächst an, wie viele Elemente das Array enthält. Die Funktion verändert den Wert dieser Variablen dann aber entsprechend in die Anzahl der Events, die im Puffer vorhanden waren. Als Flag im letzten Parameter kann man `DIGDD_PEEK` angeben. Dadurch werden die Daten aus dem Puffer nicht gelöscht. Dann muss man aber mit einem Überlaufen des Puffers rechnen, und deswegen wird man diese Option in der Regel nicht verwenden.

Puffer-Daten

Die zweite Funktion ist naturgemäß einfacher strukturiert als die Funktion für gepufferte Daten. Hier müssen wir im ersten Parameter lediglich die Größe der Struktur des zweiten Parameters angeben. Im zweiten Parameter gibt man dann eine Struktur an, in der die Funktion die States des Eingabegeräts abspeichert. Für Tastaturen ist dies ein einfaches `char`-Array und für Joysticks bietet DirectInput eine entsprechende Struktur an.

Device-States

Schauen wir uns die Funktion erst einmal an. Dann reden wir darüber, warum sie noch ein wenig komplexer ausfällt, als man das zunächst denken mag.



```
HRESULT ZFXDDevice::GetData(ZFXINPUTDEV Type,
                           void *pData,
                           DWORD *pdwNum) {

    HRESULT hr=ZFX_FAIL;
    size_t size=0;

    // Haben wir eine Maus oder Keyboard/Joystick?
    if (Type == IDV_MOUSE) {
        size = sizeof(DIDeviceOBJECTDATA);

        hr=m_pDevice->GetDeviceData(size,
                                     (DIDeviceOBJECTDATA*)
                                     pData, pdwNum, 0);
    }
    else {
        if (Type==IDV_KEYBOARD) size = sizeof(char)*256;
        else size = sizeof(DIJOYSTATE);

        hr = m_pDevice->GetDeviceState(size, pData);
    }

    // Ist die Abfrage fehlgeschlagen?
    if (FAILED(hr)) {
        // Ist das Device überhaupt akquiriert?
        if ( (hr==DIERR_NOTACQUIRED) ||
              (hr==DIERR_INPUTLOST) ) {
            hr = m_pDevice->Acquire();
            while (hr==DIERR_INPUTLOST)
                hr = m_pDevice->Acquire();

            // Eine andere Applikation hat grad Vorrang!
            if (hr==DIERR_OTHERAPPHASPRIO) return ZFX_OK;

            // Wir haben es wieder, also Daten abfragen!
            if (SUCCEEDED(hr)) {
                if (Type == IDV_MOUSE)
                    hr=m_pDevice->GetDeviceData(size,
                                                  (DIDeviceOBJECTDATA*)
                                                  pData, pdwNum, 0);
                else
                    hr = m_pDevice->GetDeviceState(size,
                                                    pData);
            }
            // ein anderer Fehler
            if (FAILED(hr)) return ZFX_FAIL;
        }
    }
}
```

```
// ein anderer Fehler  
else return ZFX_FAIL;  
}  
return ZFX_OK;  
} // GetData
```

Zuerst müssen wir die Funktion danach auftrennen, ob wir die Maus oder Tastatur/Joystick abfragen. Das ist deshalb so, weil wir die Maus im gepufferten Modus verwenden, Tastatur und Joystick jedoch nicht. Daher müssen wir zum Abfragen entsprechend die beiden verschiedenen `DirectInput`-Funktionen verwenden. Wichtig sind hierbei jedoch auch die Sicherheitsüberprüfungen, die wir vornehmen. Um ein Eingabegerät mit `DirectInput` verwenden zu können, muss man auf alle Fälle zuerst einmal noch die parameterlose Funktion `IDirectInputDevice8::Acquire` aufgerufen haben. Erst dann ist ein Device betriebsbereit. Erhalten wir bei der Abfrage der Daten den Fehler `DIERR_NOTACQUIRED`, dann ist ein Gerät eben noch nicht oder nicht mehr akquiriert, aus welchem Grund auch immer. Erhalten wir die Fehlermeldung `DIERR_INPUTLOST`, dann haben wir ein zuvor akquiriertes Device verloren, beispielsweise wenn eine andere Applikation es zwischenzeitlich akquiriert hat. In beiden Fällen versuchen wir, das Device in einer Schleife wieder zu akquirieren. Erhalten wir jedoch den Fehler `DIERR_OTHERAPPHASPRIO`, dann hat gerade eine andere Applikation Priorität bei der Verwendung dieses Eingabegeräts. An dieser Stelle ist es wichtig, keinen Fehler zurückzugeben. Diese Meldung ist kein wirklicher Fehler im Ablauf unseres Programms, sondern eine korrekte Prioritätszuordnung durch Windows. Wir müssen die Akquisition einfach zu einem späteren Zeitpunkt erneut versuchen.

Security-Checks

Haben wir es jedoch geschafft, das Device nun wieder zu akquirieren, dann versuchen wir nochmals die Abfrage der Daten bzw. des States. Schlägt das wieder fehl, so haben wir es wirklich mit einem schweren Fehler zu tun und melden dies entsprechend zurück.

Damit haben wir die Grundfunktionalität für die Kapselung eines `DirectInput-Device`-Objekts in unserer Basisklasse komplett. Jetzt kommen wir zu den abgeleiteten Klassen. Hier arbeiten wir wirklich konkret mit einer Tastatur, einer Maus und einem Joystick. Aber erschreckt euch bitte nicht, wenn der Code der jeweiligen abgeleiteten Klassen nun nur noch minimal ist.

9.3 Ran an die Tasten

Nach der ganzen mühsamen Vorarbeit ist die eigentliche Implementierung einer Klasse für die Tastatur nur noch reine Formsache. Es sind zwar ein paar Details bei der Abfrage einzelner Tasten zu berücksichtigen, aber unser Arbeitspferd `ZFXDIDevice` nimmt uns bereits viel der zu erledigenden Arbeit ab.

ZFXKeyboard-Klasse

Methoden

Als Erstes leiten wir natürlich eine eigene Klasse für Tastaturen von unserer Basisklasse ab. Diese Basisklasse schreibt uns über rein virtuelle Funktionsdefinitionen vor, dass wir die Funktionen `Init()` und `Update()` für jede Ableitung definieren müssen. Zusätzlich spendieren wir der Klasse noch ein paar Attribute und die beiden Funktionen `IsPressed()` und `IsReleased()`, mit denen wir abfragen werden, ob eine Taste gerade gedrückt ist oder gerade losgelassen wurde.



```
class ZFXKeyboard : public ZFXDDevice {
public:
    ZFXKeyboard(LPDIRECTINPUT8, HWND, FILE*);
    ~ZFXKeyboard(void);

    HRESULT Init(void);
    HRESULT Update(void);

    bool    IsPressed(UINT nID);
    bool    IsReleased(UINT nID);

private:
    char    m_Keys[256];
    char    m_KeysOld[256];
}; // class
```

Attribute

Bei den Attributen haben wir zwei Arrays, in denen wir den Zustand der Tasten speichern: ein Array für den aktuellen Zustand und ein Array für den Zustand vor der aktuellen Zustandsabfrage. Durch einen Vergleich der beiden Arrays können wir leicht bestimmen, ob eine Taste im letzten Frame gedrückt war und nun nicht mehr gedrückt ist. Ergo muss sie gerade eben losgelassen worden sein.

Initialisierung und Freigabe

Die Initialisierung und die Freigabe einer Instanz unser Tastatur-Klasse erfolgt komplett über die Methoden der Basisklasse. Daher haben wir im Konstruktor und im Destruktor nichts weiter zu tun, als die entsprechenden Funktionen aufzurufen.



```
ZFXKeyboard::ZFXKeyboard(LPDIRECTINPUT8 pDI, HWND hWnd,
                          FILE* pLog)
{ Create(pDI, hWnd, pLog); }

ZFXKeyboard::~ZFXKeyboard(void)
{ Release(); }
```

Allerdings gibt es noch die kleine `Init()`-Funktion, die wir implementieren müssen. In dieser tut sich schon ein wenig mehr, denn im Konstruktor machen wir ja nichts weiter, als die Attribute auf Startwerte zu stellen.

```
HRESULT ZFXKeyboard::Init(void) {
    if (FAILED(CrankUp(GUID_SysKeyboard,
                      &c_dfDIKeyboard)))
        return ZFX_FAIL;

    // Strukturen durchnullen
    memset(m_Keys, 0, sizeof(m_Keys));
    memset(m_KeysOld, 0, sizeof(m_KeysOld));

    // Gerät akquirieren
    m_pDevice->Acquire();
    return ZFX_OK;
} // Init
```



Den Löwenanteil der Arbeit nimmt uns hier natürlich die Funktion `ZFXDDevice::CrankUp` ab, denn dazu war sie ja auch gedacht. Man beachte die hier übergebenen Parameter, die das Gerät als Tastatur identifizieren. Zusätzlich leeren wir noch die beiden Arrays der Klasse. Abschließend akquirieren wir das Device ordnungsgemäß und haben die Tastatur damit vollständig initialisiert und für die aktive Nutzung bereit gemacht.

Crank-Up

Update

Das Update der Tastatur, das wir in jedem Frame aufrufen müssen, ist vergleichsweise lächerlich einfach. Bevor wir uns jetzt aber fragen, warum wir es dann nicht komplett über die Basisklasse erledigen, sei erwähnt, dass sich Tastatur, Maus und Joystick hier ein wenig unterscheiden und dass die letzten beiden ein klein wenig mehr Arbeit erfordern. Daher verbleibt diese Implementierung in den jeweils abgeleiteten Klassen.

```
HRESULT ZFXKeyboard::Update(void) {
    memcpy(m_KeysOld, m_Keys, sizeof(m_Keys));

    // frage den Zustand der Tastatur ab
    if (FAILED(GetData(IDV_KEYBOARD, &m_Keys[0], NULL)))
        return ZFX_FAIL;
    return ZFX_OK;
} // Update
```



Zunächst kopieren wir den bisherigen Zustand, um die Informationen aus dem letzten Frame zu erhalten. Dann holen wir uns über die Funktion `ZFXDDevice::GetData` den aktuellen Zustand der Tastatur. And that's it.

Abfrage des Inputs

Nachdem wir den Zustand der Tastatur in unseren Arrays gespeichert haben, ist es ein Leichtes herauszufinden, ob eine bestimmte Taste gedrückt ist oder soeben losgelassen wurde. Für DirectInput sind im Header `dinput.h` des DirectX SDK verschiedene Hexadezimalwerte definiert, die bestimmten Scan-Codes auf der Tastatur entsprechen. Von diesen Definitionen habe ich die wichtigsten in den Header `ZFXInputDevice.h` zu unserer Interface-Definition kopiert. Hier ein paar Beispiele:

```
#define ZVK_ESCAPE      0x01
#define ZVK_TAB         0x0F
#define ZVK_SPACE       0x39
#define ZVK_RETURN      0x1C
```

Die zugehörigen Nummern sind nicht viel mehr als die Indices in das Array mit dem Tastatur-Zustand, den wir von DirectInput für die Tastatur abgefragt haben. In diesem Element des Arrays überprüfen wir nun, ob dort das hohe Bit des Wertes gesetzt ist. Sollte dies der Fall sein, dann ist die Taste gerade gedrückt.



```
bool ZFXKeyboard::IsPressed(UINT nID) {
    if (m_Keys[nID] & 0x80)
        return true;
    return false;
} // IsPressed

bool ZFXKeyboard::IsReleased(UINT nID) {
    if ( (m_KeysOld[nID]&0x80) && !(m_Keys[nID]&0x80) )
        return true;
    return false;
} // IsPressed
```

Wie man in der zweiten Funktion sehen kann, arbeiten wir bei der Abfrage, ob eine Taste gerade losgelassen wurde, mit einem Vergleich des aktuellen und des vorherigen Zustands. Eine Taste wurde entsprechend gerade losgelassen, wenn sie im Array der Abfrage aus dem vorherigen Aufruf des Zustands der Tastatur gedrückt war, jetzt aber nicht mehr gedrückt ist. Dies setzt natürlich voraus, dass wir den Zustand auch regelmäßig abfragen, aber davon ist auszugehen.

Damit haben wir nun auch die Implementierung unserer Tastatur-Klasse so weit komplett und wenden uns den kleinen Nagetieren zu, die mittlerweile der beste Freund des Computers geworden sind.

9.4 Die Rattenfänger von Redmond

Zugegeben, es handelt sich hier um Mäuse und nicht um Ratten. Dass heutzutage fast immer eine Maus neben einem Computer liegt, mag ganz gewöhnlich erscheinen. Vor aber noch gar nicht allzu langer Zeit war eine Maus etwas Ungewöhnliches. Als textbasierte Betriebssysteme, wie beispielsweise Microsoft DOS, noch die Welt regierten, gab es nur die traute Dreisamkeit aus PC, Anwender und Tastatur. Mittlerweile gehört eine komfortable Eingabe aber bei einem grafischen Betriebssystem zum bequemen, nicht wegzudenkenden Standard, erlöst es den Anwender doch von der Qual, sich alle möglichen Befehle und Anweisungen hart anzutrainieren und diese auswendig zu lernen. Obwohl einige fossile Computer-Freaks diesen Zeiten nachtrauern, ist es doch eine nicht zu unterschätzende Entwicklung, dass man einen Computer mehr und mehr intuitiv bedienen kann. Der Umgang mit einer Maus will zwar erst gelernt sein, aber wenn man das Prinzip der Maus als Verlängerung der Hand in den Computer hinein erst einmal verstanden hat, dann ist klar, dass sie eine nicht zu unterschätzende Revolution in der Bedienungsfreundlichkeit der modernen PCs gebracht hat.

*Als die Becher
noch aus Holz
waren ...*

Rede ich zu viel? Das kann sein, aber ich finde es wichtig aufzuzeigen, dass durchaus nicht alles, was der heutigen Generation so natürlich erscheint, auch schon wirklich immer so da gewesen ist. Eine Maus ist sicherlich ein guter Anfang, aber sie sollte bei weitem nicht das Ende der Entwicklung sein. Das sollte man nie aus dem Auge verlieren.

ZFXMouse-Klasse

Um die Maus in den Griff zu bekommen, brauchte man früher eine Flöte. In Zeiten wie diesen reicht es aber, dass wir DirectInput zur Verfügung haben. Wer so unmusikalisch ist wie ich es bin, der wird das sehr zu schätzen wissen. Tatsächlich müssen wir für die Maus nur ein paar Zeilen Code implementieren. Wir beginnen wieder mit der Klassen-Definition.

*Spiel mir das Lied
vom DirectInput*

```
class ZFXMouse : public ZFXDDevice {
public:
    ZFXMouse(LPDIRECTINPUT8, HWND, FILE*);
    ~ZFXMouse(void);

    HRESULT Init(void);
    HRESULT Update(void);

    void SetCage(RECT rcCage) { m_rcCage = rcCage; }

    bool IsPressed(UINT nBtn)
    { if (nBtn<3) return m_bPressed[nBtn];
      return false; }
```



```

bool IsReleased(UINT nBtn)
{ if (nBtn<3) return m_bReleased[nBtn];
  return false; }

private:
    HANDLE                m_hEvent;
    bool                  m_bPressed[3];
    bool                  m_bReleased[3];
}; // class

```

Methoden

Die Abfrage des Maus-Zustandes – oder präziser formuliert: der Maustasten – erledigen wir hier in der Klassen-Definition. Wie wir jedoch die Werte für das Maustasten-Array setzen, das sehen wir gleich in der `Update()`-Funktion. Die Position der Maus haben wir ja bereits über die Funktionen in der Klassendefinition der Basisklasse an den Aufrufer weitergegeben. Wichtig ist noch das Setzen von `m_rcCage`, das wir auch in der `Update()`-Funktion erklären.

Initialisierung und Freigabe

Der Konstruktor und der Destruktor der Maus sind genauso langweilig wie die entsprechenden Funktionen bei der Tastatur. Aber genau das macht ja den Sinn der vererbenden Basisklasse aus, dass sie alle Methoden deklariert die für die verschiedenen denkbaren Ableitungen gleichsam vorhanden sein müssen.



```

ZFXMouse::ZFXMouse(LPDIRECTINPUT8 pDI, HWND hWnd,
                   FILE* pLog)
{ Create(pDI, hWnd, pLog); }

```

```

ZFXMouse::~ZFXMouse(void)
{ Release(); }

```

Aber bei der anschließenden Initialisierung der Maus haben wir schon ein kleines bisschen mehr Arbeit, als das bei der Tastatur der Fall war. Neben der Funktion `ZFXDI::CrankUp` der Basisklasse müssen wir insbesondere noch den Puffer für die Mausdaten erzeugen. Dazu verwenden wir die `DirectInput`-Struktur `DIPROPDWORD`, in der wir bestimmte Eigenschaften für ein `DirectInput-Device` einstellen können. Hier müssen wir die Größe des Puffers angeben und über die Funktion `IDirectInputDevice8::SetProperty` diese Einstellungen für das Eingabegerät aktivieren.

```

#define BUFFER_SIZE 16

HRESULT ZFXMouse::Init(void) {
    // Arrays ausnullen
    memset(m_bPressed, 0, sizeof(bool)*3);
    memset(m_bReleased, 0, sizeof(bool)*3);
    m_lX = m_lY = 0;

    if (FAILED(CrankUp(GUID_SysMouse, &c_dfDIMouse)))
        return ZFX_FAIL;

    // Event-Benachrichtigung
    if (!(m_hEvent = CreateEvent(NULL, FALSE, FALSE,
                                NULL)))
        return ZFX_FAIL;

    if (FAILED(m_pDevice->SetEventNotification(
                                                m_hEvent)))
        return ZFX_FAIL;

    // Mäuse-Puffer aufbauen
    DIPROPDWORD dipdw;
    dipdw.diph.dwSize      = sizeof(DIPROPDWORD);
    dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);
    dipdw.diph.dwObj       = 0;
    dipdw.diph.dwHow       = DIPH_DEVICE;
    dipdw.dwData           = BUFFER_SIZE;

    if (FAILED(m_pDevice->SetProperty(DIPROP_BUFFERSIZE,
                                    &dipdw.diph)))
        return ZFX_FAIL;

    m_pDevice->Acquire();
    return ZFX_OK;
} // Init

```

Abschließend dürfen wir natürlich auch hier nicht vergessen, das Eingabegerät, das die Maus repräsentiert, durch die Funktion `Acquire()` zu aktivieren. Nun ist die gepufferte Maus betriebsbereit.

*Akquirieren der
Maus*

Update

Die letzte Funktion, die wir in unserer Maus-Klasse brauchen, ist `ZFXMouse::Update`. Über die Funktion `ZFXDI::GetData` der Basisklasse können wir ja, wie weiter oben gezeigt, die Daten des Puffers der Maus über `DirectInput` abfragen, und zwar mittels eines Arrays der Struktur `DIDEVICEOBJECTDATA` in der Größe des Puffers. Nachdem wir das gemacht haben, müssen wir die Daten aber noch auswerten. Im dritten Parameter der Funktion können wir einen Pointer auf eine `DWORD`-Variable übergeben. Beim Aufruf der Funktion



muss dort gespeichert sein, wie groß der Puffer ist. Nach dem Aufruf der Funktion ist dort gespeichert, wie viele Events im Puffer vorhanden sind. Über genau diese Anzahl müssen wir dann eine Schleife laufen lassen, um alle Events im Puffer über die jeweiligen Array-Einträge abzuarbeiten.

Im Feld `dwOfs` der Struktur ist dann eine der vordefinierten Maus-Konstanten von `DirectInput` gespeichert, die wir abfragen können:

```
typedef enum {
    DIMOFS_BUTTON0,
    DIMOFS_BUTTON1,
    DIMOFS_BUTTON2,
    DIMOFS_BUTTON3,
    DIMOFS_BUTTON4,
    DIMOFS_BUTTON5,
    DIMOFS_BUTTON6,
    DIMOFS_BUTTON7,
    DIMOFS_X,
    DIMOFS_Y,
    DIMOFS_Z
} Mouse Device;
```

Die Bezeichner sind selbsterklärend, daher nehmen wir sie einfach so hin und schreiben gleich die Funktion:



```
HRESULT ZFXMouse::Update(void) {
    DIDEVICEOBJECTDATA od[BUFFER_SIZE];
    DWORD dwNumElem = BUFFER_SIZE;

    // lies die Daten des Maus-Puffers
    if (FAILED(GetData(IDV_MOUSE, &od[0], &dwNumElem)))
        return ZFX_FAIL;

    m_bReleased[0] = m_bReleased[1] = m_bReleased[2] =
        false;

    // jetzt haben wir 'dwNumElem' Maus-Events
    for (DWORD i=0; i<dwNumElem; i++) {
        switch (od[i].dwOfs) {
            // BEWEGUNG
            case DIMOFS_X: {
                m_lX += od[i].dwData;
                if (m_lX < m_rcCage.left)
                    m_lX = m_rcCage.left;
                else if (m_lX > m_rcCage.right)
                    m_lX = m_rcCage.right;
            } break;

            case DIMOFS_Y: {
```

```

        m_lY += od[i].dwData;
        if (m_lY < m_rcCage.top)
            m_lY = m_rcCage.top;
        else if (m_lY > m_rcCage.bottom)
            m_lY = m_rcCage.bottom;
    } break;

// MAUS-TASTEN
case DIMOFS_BUTTON0: {
    if (od[i].dwData & 0x80) {
        m_bPressed[0] = true;
    }
    else {
        if (m_bPressed[0])
            m_bReleased[0] = true;
        m_bPressed[0] = false;
    }
    } break;

case DIMOFS_BUTTON1: {
    if (od[i].dwData & 0x80)
        m_bPressed[1] = true;
    else {
        if (m_bPressed[1])
            m_bReleased[1] = true;
        m_bPressed[1] = false;
    }
    } break;

case DIMOFS_BUTTON2: {
    if (od[i].dwData & 0x80)
        m_bPressed[2] = true;
    else {
        if (m_bPressed[2])
            m_bReleased[2] = true;
        m_bPressed[2] = false;
    }
    } break;
}; // switch
} // for
return ZFX_OK;
} // Update

```

Man beachte, dass wir die Maus darauf beschränken, sich nur innerhalb eines gewissen Käfigs zu bewegen. Diesen geben wir in Form des Attributs `m_rcCage` der Klasse vom Typ `RECT` an. Die Koordinaten der Maus sind nun auf den Bereich beschränkt, den das Rechteck vorgibt. Eine `DirectInput-Maus` sendet beispielsweise immer dann das Event `DIMOFS_X`, wenn man die Maus nach rechts oder links bewegt. Dabei kennt `DirectInput` keine Beschränkung der Koordinaten auf einen bestimmten Wert. Ist der Maus-

Maus im Käfig

zeiger beispielsweise ganz am rechten Rand und bewegt der Benutzer die Maus trotzdem weiter nach rechts, so sendet die Maus auch weiterhin die entsprechende Nachricht. Hier muss der Programmierer aufpassen, dass er die Koordinaten der Maus dennoch im gültigen Bereich der Anwendung hält.

9.5 Kein Spaß ohne Joystick

Ohne viele Worte möchte ich nun auch den Joystick abhandeln. Dessen Initialisierung ist fast so erschreckend langweilig wie die der Tastatur und der Maus. Ich kann kurz mit dem Schreckenswörtchen Enumeration drohen, aber selbst diese ist lächerlich einfach. Also: Ran an den Speck bzw. den Joystick.

ZFXJoystick-Klasse

Natürlich brauchen wir auch für den Joystick eine eigene Klasse. Auffällig ist hierbei nur, dass diese Klasse ein paar Extrawürste hat, verglichen mit der Maus und Tastatur. Die Funktion `JoystickFound()` dient dazu, dem Aufrufer bekannt zu machen, ob überhaupt ein Joystick vorhanden ist. Kein PC kommt ohne Tastatur oder Maus aus, aber ein Joystick muss ja nicht unbedingt vorhanden sein. Das herauszufinden ist die Aufgabe der Funktion `EnumJoyCallback()`. Last but not least können wir über die Funktion `GetName()` den Namen des Joysticks erhalten, sofern einer vorhanden ist.



```
class ZFXJoystick : public ZFXDDevice {
public:
    ZFXJoystick(LPDIRECTINPUT8, HWND, FILE*);
    ~ZFXJoystick(void);

    HRESULT Init(void);
    HRESULT Update(void);

    bool JoystickFound(void) { return m_bJoyFound; }
    BOOL EnumJoyCallback(const DIDEVICEINSTANCE *pI);
    void GetName(char *pJoyName)
        { memcpy(pJoyName, m_Name, sizeof(char)*256); }

    bool IsPressed(UINT nBtn)
        { if (nBtn < m_dwNumBtns) return m_bPressed[nBtn];
          return false; }

    bool IsReleased(UINT nBtn)
        { if (nBtn < m_dwNumBtns) return m_bReleased[nBtn];
          return false; }

private:
```

```

GUID          m_guid;
char          m_Name[256];
bool          m_bJoyFound;
bool          m_bPressed[12];
DWORD        m_dwNumBtns;
}; // class

```

Bemerkenswert ist hier noch das Attribut `m_guid`. Im Gegensatz zu den Standard-GUIDs der System-Tastatur und der System-Maus haben wir so etwas für den Joystick nicht. Durch die Enumeration der angeschlossenen Joysticks können wir diese aber ermitteln.

GUID

Initialisierung und Freigabe

Nein, spannender wird es auch hier erst mal nicht. Der Konstruktor und der Destruktor sind langweilig wie eh und je bei diesen Ableitungen:

```

ZFXJoystick::ZFXJoystick(LPDIRECTINPUT8 pDI, HWND hWnd,
                        FILE* pLog)
{ Create(pDI, hWnd, pLog); }

```

```

ZFXJoystick::~ZFXJoystick()
{ Release(); }

```

Die Initialisierung eines Joysticks ist endlich einmal mit etwas Spannung verbunden. Es gibt hier die Möglichkeit, dass mehrere Joysticks oder auch gar keiner an den Computer angeschlossen sind. Daher müssen wir die verfügbaren enumerieren. Eine Callback-Funktion in einer Klasse unterzubringen ist aber, wie wir schon im dritten Kapitel bei der Enumeration der Grafikkarten gesehen haben, nicht so ganz einfach. Daher gehen wir hier denselben Umweg über eine globale Variable der Klasse `ZFXJoystick`, über die wir die Callback-Funktion in der Klasse aufrufen können.

Enumeration

Für DirectInput existiert in der Welt der Eingabegeräte neben Tastaturen und Mäusen nichts weiter als so genannte Game-Controller. In diese Kategorie fallen sowohl Joysticks als auch Gamepads. Beide Arten von Eingabegeräten werden entsprechend analog behandelt.

Um angeschlossene Joysticks zu enumerieren, bietet DirectInput die Funktion `IDirectInput8::EnumDevices`. Im zweiten Parameter geben wir dabei die Callback-Funktion an, die für jedes gefundene Device angesprochen wird. Den Aufruf zu `ZDXDI::CrankUp` erledigen wir dann in der Callback-Funktion unserer Klasse. Wir erzeugen also vollkommen unwählerisch aus dem ersten gefundenen Joystick ein aktives Device. Danach müssen wir die Eigenschaften für den Joystick einstellen, beispielsweise den Wertebereich auf den Achsen.

Initialisierung





```

ZFXJoystick *g_pThis=NULL;
BOOL CALLBACK gEnumJoyCallback(
    const DIDEVICEINSTANCE* pInst,
    void* pUserData)
{ return g_pThis->EnumJoyCallback(pInst); }

HRESULT ZFXJoystick::Init(void) {
    DIPROPRANGE diprg;
    DIDEVCAPS diCaps;

    // einige Inits
    memset(m_bPressed, 0, sizeof(m_bPressed));
    memset(m_bReleased, 0, sizeof(m_bReleased));
    m_bJoyFound = false;
    m_lX = m_lY = 0;
    g_pThis = this;

    // enumriere angeschlossene Joysticks
    m_pDI->EnumDevices(DI8DEVCLASS_GAMECTRL,
        (LPDIENUMDEVICESCALLBACK)
        gEnumJoyCallback,
        &m_guid, DIEDFL_ATTACHEDONLY);

    // Keinen gefunden?
    if (!m_bJoyFound)
        return ZFX_FAIL;

    // Letzte Einstellungen
    diprg.diph.dwSize = sizeof(DIPROPRANGE);
    diprg.diph.dwHeaderSize = sizeof(DIPROPHEADER);
    diprg.diph.dwHow = DIPH_BYOFFSET;
    diprg.lMin = -1000;
    diprg.lMax = +1000;

    diprg.diph.dwObj = DIJOFS_X;
    m_pDevice->SetProperty(DIPROP_RANGE, &diprg.diph);

    diprg.diph.dwObj = DIJOFS_Y;
    m_pDevice->SetProperty(DIPROP_RANGE, &diprg.diph);

    // Anzahl der Feuerknöpfe
    if (SUCCEEDED(m_pDevice->GetCapabilities(&diCaps)))
        m_dwNumBtns = diCaps.dwButtons;
    else
        m_dwNumBtns = 4;
    return ZFX_OK;
} // Init

```


Wer bei der Initialisierung von Joysticks ein wenig Nachholbedarf hat, der sollte sich das entsprechende Tutorial in der DirectX SDK-Dokumentation durchlesen oder anhand der hier gezeigten Funktionen deren Sinn in der DirectInput-Referenz nachschlagen.

Wissenslücke?

Der folgende Code ist die Callback-Funktion, die wir für gefundene Joysticks anspringen. Wir versuchen sofort, einen gefundenen Joystick gefechtsbereit zu machen. Glückt dieser Versuch, so beenden wir die Enumeration und sind zufrieden. Anderenfalls machen wir mit der Enumeration weiter, bis wir den nächsten Joystick finden oder bis es keinen weiteren mehr gibt.

```
BOOL ZFXJoystick::EnumJoyCallback(
    const DIDEVICEINSTANCE *pInst) {
    // versuche, den gefundenen Joystick zu starten
    if (SUCCEEDED(CrankUp(pInst->guidInstance,
        &c_dfDIJoystick))) {
        m_bJoyFound = true;
        strcpy(m_Name, (char*)pInst->tszProductName);
        return DIENUM_STOP;
    }
    return DIENUM_CONTINUE;
} // EnumJoyCallback
```



Update

Den Zustand eines Joysticks abfragen zu wollen ist eine triviale Aufgabe. Zuerst müssen wir dazu die für Joysticks besondere DirectInput-Funktion `IDirectInputDevice8::Poll` aufrufen. Damit fordert DirectInput das Device auf, seine aktuellsten Daten preiszugeben. Einige Joysticks und Gamepads melden bei dieser Funktion jedoch einen Fehler, weil sie diesen Aufruf nicht nötig haben. Diesen Fehler kann man getrost ignorieren.

Poll

Danach können wir über `ZFXDI::GetData` wie gewohnt den Zustand des Joysticks abfragen – diesmal allerdings über die Struktur `DIJOYSTATE`. DirectInput betankt diese Struktur dann mit den Zuständen der Feuerknöpfe des Joysticks und mit der Stellung des Steuerknüppels. Die entsprechenden Werte speichern wir in die Attribute der Klasse um, um sie auf Wunsch angeben zu können.

```
HRESULT ZFXJoystick::Update(void) {
    DIJOYSTATE js;

    // Poll auf den Joystick
    m_pDevice->Poll();

    // Hole die Daten vom Joystick
    if (FAILED(GetData(IDV_JOYSTICK, &js, NULL)))
        return ZFX_FAIL;
```



```

// Joystick-Buttons
for (DWORD i=0; i<m_dwNumBtns; i++) {
    m_bReleased[i] = false;
    if (js.rgbButtons[i] & 0x80)
        m_bPressed[i] = true;
    else {
        if (m_bPressed[i]) m_bReleased[i] = true;
        m_bPressed[i] = false;
    }
}

// Position des Sticks
m_lX = js.lX;
m_lY = js.lY;

return ZFX_OK;
} // Update

```

Damit sind wir schon fast durch, was unsere Eingabegeräte angeht. Nun fehlt uns nur noch die wirkliche Implementierung des Interfaces. Dabei werden wir natürlich die eben entwickelten Klassen entsprechend verwenden.

9.6 Implementierung des Interfaces

Bei der tatsächlichen Implementierung des Interfaces `ZFXInputDevice` in unserer DLL als Klasse `ZFXDI` gibt es keine Überraschungen. Die Definition der Klasse sieht genauso aus, wie es das Interface vorsieht. Wir benötigen keine zusätzlichen Funktionen. Allerdings haben wir noch ein paar zusätzliche Attribute: zum einen das `DirectInput`-Hauptobjekt `m_pDI` und zum anderen je ein Attribut für die Tastatur, die Maus und den Joystick.



```

class ZFXDI : public ZFXInputDevice {
public:
    ZFXDI(HINSTANCE hDLL);
    ~ZFXDI(void);

    HRESULT Init(HWND, RECT, bool);

    void Release(void);
    bool IsRunning(void) { return m_bRunning; }
    bool HasJoystick(char *pJoyName);

    HRESULT Update(void);

    bool IsPressed(ZFXINPUTDEV idType, UINT nID);
    bool IsReleased(ZFXINPUTDEV idType, UINT nID);
    HRESULT GetPosition(ZFXINPUTDEV idType,
        POINT *pPt);

```

```
private:
    LPDIRECTINPUT8 m_pDI;
    ZFXKeyboard    *m_pKB;
    ZFXMouse       *m_pMouse;
    ZFXJoystick    *m_pJoy;
}; // class
```

Initialisierung und Freigabe

Der Konstruktor unserer neuen Klasse besteht nur aus dem üblichen Standard: aus dem Ausnullen der Pointer-Attribute. Im Destruktor fällt ein kleines bisschen mehr Arbeit an, denn hier müssen wir insbesondere unsere drei Eingabegeräte und das DirectInput-Hauptobjekt freigeben.

*Konstruktion und
Destruktion*

```
ZFXDI::ZFXDI(HINSTANCE hDLL) {
    m_hDLL      = hDLL;
    m_pDI       = NULL;
    m_pLog      = NULL;
    m_bRunning  = false;
    m_pKB       = NULL;
    m_pMouse    = NULL;
    m_pJoy      = NULL;
}
```

```
ZFXDI::~ZFXDI() {
    Release();
}
```

```
void ZFXDI::Release() {
    if (m_pKB) {
        delete m_pKB;
        m_pKB = NULL;
    }
    if (m_pMouse) {
        delete m_pMouse;
        m_pMouse = NULL;
    }
    if (m_pJoy) {
        delete m_pJoy;
        m_pJoy = NULL;
    }
    if (m_pDI) {
        m_pDI->Release();
        m_pDI = NULL;
    }
}
```



Initialisierung

Nachdem sich der Anwender unserer DLL ein `ZFXInputDevice`-Objekt hat erzeugen lassen, muss er wie gehabt die `Init()`-Funktion dieses Objekts aufrufen, um es gefechtsbereit zu machen. In der Implementierung dieser Funktion erzeugen wir zunächst das `DirectInput`-Hauptobjekt. War das erfolgreich, so konstruieren wir unsere drei Objekte für die Eingabegeräte und initialisieren sie dann eines nach dem anderen. Schlägt das bei der Tastatur oder der Maus fehl, so melden wir einen schweren Fehler an den Aufrufer zurück. Schlägt das für den Joystick fehl, so kann es auch daran liegen, dass gar kein Joystick an den Computer angeschlossen ist. In diesem Fall ist das nicht tragisch, da ein Joystick immer nur optional sein sollte. Folglich löschen wir das Objekt für den Joystick wieder, setzen aber dennoch eine *Alles-Okay*-Meldung an den Aufrufer ab.



```
HRESULT ZFXDI::Init(HWND hWnd, RECT rcMouseCage,
                    bool bSaveLog) {
    HRESULT hr;
    m_hWndMain = hWnd;

    // erzeuge DirectInput-Hauptobjekt
    if (FAILED (hr = DirectInput8Create(m_hDLL,
                                        DIRECTINPUT_VERSION,
                                        IID_IDirectInput8,
                                        (void**)&m_pDI,
                                        NULL)) )
        return ZFX_FAIL;

    // erzeuge alle InputDevice Objekte
    m_pKB   = new ZFXKeyboard(m_pDI, hWnd, m_pLog);
    m_pMouse = new ZFXMouse(m_pDI, hWnd, m_pLog);
    m_pJoy   = new ZFXJoystick(m_pDI, hWnd, m_pLog);

    // initialisiere alle Input-Device Objekte
    if (FAILED( m_pKB->Init() )) {
        if (m_pKB) delete m_pKB;
        m_pKB = NULL;
        return ZFX_FAIL;
    }

    if (FAILED( m_pMouse->Init() )) {
        if (m_pMouse) delete m_pMouse;
        m_pMouse = NULL;
        return ZFX_FAIL;
    }
    m_pMouse->SetCage(rcMouseCage);

    if (FAILED( m_pJoy->Init() )) {
        if (m_pJoy) delete m_pJoy;
        m_pJoy = NULL;
    }
}
```

```
m_bRunning = true;
return ZFX_OK;
} // Init
```

Wie ihr hier seht, benötigt die Funktion neben dem Handle auf das Hauptfenster der Anwendung, in der wir die Eingabegeräte nutzen wollen, als Parameter noch ein RECT-Objekt. Dabei handelt es sich um den Gültigkeitsbereich für die Maus-Koordinaten. Ein solches Rechteck kann man bei einer Desktop-Auflösung von 1024x768 beispielsweise wie folgt initialisieren:

Maus-Käfig

```
RECT rcCage = { 0, 0, 1023, 767 };
```

Anhand diverser Parameter bezüglich eines mysteriösen Log-Files erkennt man auch hier, dass ich ein paar Dinge aus den Funktionen herausgeschnitten habe. Wenn ihr euch den Quelltext auf der CD-ROM genau anschaut, werdet ihr dort auch Funktionen und Aufrufe für das Ausgeben von Fehlermeldungen in eine Log-Datei finden. Der besseren Übersichtlichkeit halber habe ich diese für den Abdruck des Codes hier im Buch weggekürzt. In einem echten Projekt sollte man auf so einen eingebauten kleinen Runtime-Debugger aber nicht verzichten.



Update

Hier gibt es nicht viel zu sagen. Die gesamte Arbeit steckt ja in der Klasse ZFXDIDevice und deren Ableitungen. Aus der Klasse ZFXDI heraus müssen wir diese dann nur noch aufrufen.

```
HRESULT ZFXDI::Update(void) {
    HRESULT hr;

    if (!IsRunning()) return ZFX_FAIL;

    if (m_pKB) {
        if ( FAILED( hr=m_pKB->Update() ) )
            return hr;
    }
    if (m_pMouse) {
        if ( FAILED( hr=m_pMouse->Update() ) )
            return hr;
    }
    if (m_pJoy) {
        if ( FAILED( hr=m_pJoy->Update() ) )
            return hr;
    }
    return ZFX_OK;
} // Update
```



Abfrage der Daten

*Joystick
vorhanden?*

Um nun nach einem Update an die Daten eines Eingabegeräts heranzukommen, haben wir einen Satz von vier Funktionen, mit denen wir alles abdecken, was sich der Aufrufer wünschen kann. Obwohl wir das Vorhandensein eines Joysticks bereits intern in der DLL managen, könnte es für den Aufrufer von Interesse sein, ob er einen Joystick zur Verwendung exponieren kann oder nicht.



```
bool ZFXDI::HasJoystick(char *pJoyName) {
    if (m_pJoy) {
        if (pJoyName) m_pJoy->GetName(pJoyName);
        return true;
    }
    return false;
} // HasJoystick
```

*Maus- und
Joystick-Position*

Für die Maus und den Joystick ist es dann wichtig, an die Positionsdaten heranzukommen. Für die Maus ist das die Position des Mauszeigers nach den letzten Bewegungen des Benutzers an der Maus selbst. Für den Joystick können wir auch eine zweidimensionale Position zurückgeben, die die aktuelle Stellung des Sticks auf seinen beiden Achsen darstellt. Je mehr diese Werte nach oben oder unten von 0 abweichen, desto stärker ist der Joystick auf der entsprechenden Achse bewegt worden.



```
HRESULT ZFXDI::GetPosition(ZFXINPUTDEV idType,
                           POINT *pPt) {
    if (idType == IDV_MOUSE) {
        m_pMouse->GetPosition(pPt);
        return ZFX_OK;
    }
    else if (idType==IDV_JOYSTICK) {
        if (m_pJoy)
            m_pJoy->GetPosition(pPt);
        else {
            (*pPt).x = 0;
            (*pPt).y = 0;
        }
        return ZFX_OK;
    }
    else return ZFX_INVALIDPARAM;
} // GetPosition
```

Buttons

Zu guter Letzt haben wir noch die Buttons. Die Tastatur hat natürlich die Tasten, die Maus hat die Maustasten, von denen wir bis zu drei unterstützen, und der Joystick hat Feuerknöpfe, von denen wir bis zu zwölf unterstützen. Die folgenden beiden Funktionen dienen nun dazu, den Zustand eines bestimmten Buttons der Tastatur, Maus oder Joysticks abzufragen.

Die Funktionalität der Abfrage steckt dabei in der abgeleiteten Klasse für die Eingabegeräte, hier unterscheiden wir nur nach dem Objekt, an das wir den Aufruf weiterleiten müssen.

```
bool ZFXDI::IsPressed(ZFXINPUTDEV idType, UINT nBtn) {
    if (idType == IDV_MOUSE)
        return m_pMouse->IsPressed(nBtn);
    else if (idType==IDV_KEYBOARD)
        return m_pKB->IsPressed(nBtn);
    else if ( (idType==IDV_JOYSTICK) && (m_pJoy) )
        return m_pJoy->IsPressed(nBtn);
    else
        return false;
} // Pressed
```

```
bool ZFXDI::IsReleased(ZFXINPUTDEV idType, UINT nBtn) {
    if (idType == IDV_MOUSE)
        return m_pMouse->IsReleased(nBtn);
    else if (idType==IDV_KEYBOARD)
        return m_pKB->IsReleased(nBtn);
    else if ( (idType==IDV_JOYSTICK) && (m_pJoy) )
        return m_pJoy->IsReleased(nBtn);
    else
        return false;
} // Released
```

Mit dieser nun kompletten DLL können wir uns daran machen, eine kleine Demo-Applikation zu schreiben, um die Funktionalität unserer DLL zu evaluieren.

Fertig!

9.7 Demo-Applikation zur Anwendung der DLL

Unsere kleine Demo-Applikation besteht nur aus einem kleinen Fenster mit ein wenig Text, um sie so einfach wie möglich zu halten. Sie kann jedoch auf Tastatur-, Maus- und Joystick-Ereignisse reagieren, entweder durch Ausgabe von Zahlen, eines Textes oder einer Message-Box. Das kleine Ding kann also die gesamte Mächtigkeit unserer neuen Input-DLL demonstrieren und ist gleichzeitig ein Beweis für die Simplizität der Programmierung mit unserer *ZFXEngine*. Neben einer Standard-WinMain()-Funktion und einer minimalen Windows-Prozedur sind lediglich ein paar Zeilen Quellcode notwendig, um die gesamte Vielfalt von Input-Möglichkeiten über die physikalischen Standard-Eingabegeräte Tastatur, Maus und Joystick abzudecken.

Funktionalität

Viele Änderungen am bisherigen Code der Demo-Applikation gibt es nicht. Das heißt, wir verwenden hier als Grundlage die Demo-Applikation aus dem sechsten Kapitel. Anstelle des *ZFXRenderDevice*-Objekts verwenden wir

Neue Startup-Funktion

hier das `ZFXInputDevice`. Den kompletten Code findet ihr natürlich auf der CD-ROM. Hier zeige ich nur kurz die Initialisierung unseres Input-Device-Objekts. Man kann schön erkennen, dass sich auch das Input-Device an dasselbe Schema hält wie das Render-Device. Zuerst erstellen wir das Hauptobjekt, dann erzeugen wir das Device und lassen uns einen Pointer darauf geben. Abschließend rufen wir die Initialisierung des Devices auf und können es danach verwenden.



```
HRESULT ProgramStartup(void) {
    RECT rcCage = { 0, 0, 0, 0 };

    HDC hdc = GetDC(NULL);
    rcCage.right = GetDeviceCaps(hdc, HORZRES) - 1;
    rcCage.bottom = GetDeviceCaps(hdc, VERTRES) - 1;
    ReleaseDC(NULL, hdc);

    // erzeuge das ZFXInput-Hauptobjekt
    g_pInput = new ZFXInput(g_hInst);

    // erzeuge das ZFXInputDevice-Objekt
    if (FAILED( g_pInput->CreateDevice() ))
        return ZFX_FAIL;

    // hole einen Pointer auf das Device
    g_pDevice = g_pInput->GetDevice();
    if(g_pDevice == NULL) return E_FAIL;

    // initialisiere das Device
    if (FAILED(g_pDevice->Init(g_hWnd, rcCage, true)))
        return ZFX_FAIL;

    return ZFX_OK;
} // ProgramStartup
```

Maus-Käfig und Joystick-Flattern

Als Maus-Käfig legen wir hier die Abmessungen des Desktops zugrunde. Kleinere Werte als 0 lassen wir als Koordinaten für den Mauszeiger nicht zu, ebenso wenig wie Koordinaten, die größer sind als der entsprechende Wert der Desktop-Abmessung. Bei der Demo-Applikation gibt es aber ein paar kleine Schönheitsfehler: Zum einen stimmen die angezeigten Mauskoordinaten nicht mit dem Mauszeiger überein, den Windows darstellt. Die von DirectInput gelieferten Koordinaten entsprechen also nicht direkt den Koordinaten, die Windows für die Maus verwendet. Das hat verschiedene Gründe, beispielsweise die unter Windows konfigurierbare Mausgeschwindigkeit. Wenn man also eine Windows-Applikation programmiert, die im Fenster-Modus auf dem Desktop läuft und im Clientbereich nicht nur Grafikausgabe (z.B. über ein `ZFXRenderDevice`-Objekt), sondern auch Windows-API-Elemente wie Buttons usw. enthält, dann sollte man auch auf die Input-Verarbeitung von Windows zurückgreifen, was die Maus und die Mausposition angeht.

Ein anderes scheinbares Problem ist ein Flattern der Koordinaten des Joysticks. Das hängt damit zusammen, dass der Joystick selbst im Ruhezustand nicht hundertprozentig kalibriert ist. Vielmehr sendet er auch dann noch minimale Signale. Wenn man also die Bewegung des Joysticks abfragt, sollte man die Position des Joysticks immer nur dann als Veränderung vom Ruhezustand ansehen, wenn ein entsprechend großer absoluter Wert gesendet wird. Aber das kann man anhand der Ausgabe dieser Applikation auch gut erkennen – es sei denn, man hat einen sehr guten und un-abgenutzten Joystick, der entsprechend wenig flattert.

In der Funktion `ProgramTick()` der Demo-Applikation werden verschiedene Input-Events abgefragt und entsprechend im Fenster der Applikation angezeigt. Das möchte ich jetzt hier nicht alles auflisten, da es eine vielfache Wiederholung redundanten Codes wäre. Die Applikation erkennt sowohl das *Gedrücktsein* als auch das Loslassen von zwei Maustasten und vier Joystick-Feuertasten. Für diese gibt es jeweils eine numerische Anzeige, die auf 0 steht, wenn die zugehörige Taste nicht gedrückt ist, und die auf 1 steht, wenn die Taste gedrückt ist. Beim Loslassen einer Maus- oder Joystick-Taste reagiert das Programm mit einer Message-Box darauf. Der Code dazu sieht z.B. für eine Maustaste wie folgt aus:

Abgefragte Inputs

```
#define MSGBX(a) MessageBox(NULL, a,
                          "ZFXEngine report",
                          MB_OK|MB_ICONINFORMATION)

if (g_pDevice->IsPressed(IDV_MOUSE, 0)) {
    SetWindowText(hMouseBtn0, "1");
}
else {
    SetWindowText(hMouseBtn0, "0");
    if (g_pDevice->IsReleased(IDV_MOUSE, 0))
        MSGBX ("mouse button 0 released");
}
```



Zu den weiteren abgefragten Inputs gehören die Tasten , ,  und . Die ersten beiden lösen eine Reaktion aus, sobald sie gedrückt werden, die letzten beiden tun dies, sobald sie losgelassen werden. Im Code sieht das wie folgt aus:

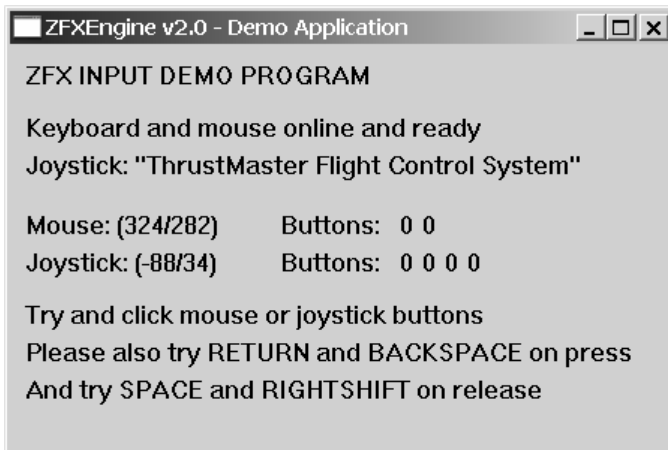
```
if (g_pDevice->IsPressed(IDV_KEYBOARD, ZVK_RETURN))
    MSGBX("RETURN key was pressed");
if (g_pDevice->IsPressed(IDV_KEYBOARD, ZVK_BACK))
    MSGBX("BACKSPACE key was pressed");
if (g_pDevice->IsReleased(IDV_KEYBOARD, ZVK_SPACE))
    MSGBX("SPACEBAR was released");
if (g_pDevice->IsReleased(IDV_KEYBOARD, ZVK_RSHIFT))
    MSGBX("RIGHTSHIFT key was released");
```



Screenshot

Damit ihr einen besseren Eindruck von der Demo-Applikation erhaltet, zeige ich mit der Abbildung 9.1 einen Screenshot, bevor ihr dann selbst die CD-ROM ins Laufwerk schiebt und das Programm endlich testet.

Abbildung 9.1:
Screenshot der
Demo-Applikation



9.8 Ein Blick zurück, zwei Schritt nach vorn

Resümee

Das nenne ich einen Schnelldurchlauf in Sachen DirectInput! Auf gerade mal knapp über 30 Seiten haben wir nicht nur die Tastatur-, Maus- und Joystick-Verwendung abgearbeitet, sondern auch noch unser Grundwissen über DirectInput aufgefrischt und, was das wichtigste ist, den gesamten Code perfekt in unsere Engine-Struktur integriert.

ZFXEngine

Durch das neue `ZFXInputDevice`-Objekt haben wir neben dem `ZFXRenderDevice`-Objekt nun unsere zweite Komponente in der *ZFXEngine*. Beide Objekte verwenden dasselbe Design und können daher durch vollkommen analoge Schritte in einer Applikation verwendbar gemacht werden. Naturgemäß ist natürlich die 3D-Engine weit komplexer als eine Input-Engine. Auch diese Tatsache haben wir schön in unserem Design spiegeln können. Schließlich kommt unsere Input-Engine mit einem Satz von gerade mal einem guten halben Dutzend Funktionen aus, die dem Anwender alle Wünsche erfüllen, die er an eine Input-Engine stellen kann.

*Was es da noch
gibt!*

Natürlich kann man auch die Komplexität der Input-Engine um ein Vielfaches erweitern. Wir haben hier insbesondere zwei wichtige Themen weglassen: zum einen die Behandlung von Force-Feedback-Geräten. Dabei handelt es sich um Joysticks oder Lenkräder, die durch eingebaute Motoren einen Gegendruck zu den Bewegungen des Spielers aufbauen können. Durch die Steuerung dieser Motoren können wir Programmierer beispielsweise Turbulenzen für ein Flugzeug simulieren oder eine holprige Straße für ein Auto. DirectInput bietet natürlich auch für Force-Feedback-Joysticks Unter-

stützung an. Dieses Thema habe ich jedoch ausgelassen, weil mir der entsprechende Platz nicht mehr zur Verfügung stand. Aber mit dem hier erworbenen Wissen und der SDK-Dokumentation sollte es euch nun ohne weiteres möglich sein, die Klasse `ZFXJoystick` optional entsprechend zu erweitern.

Zum anderen haben wir hier ein fixiertes Set von Eingabe-Möglichkeiten. In beinahe jedem aktuellen Computerspiel kann der Spieler aber die Tasten beliebigen Aktionen im Spiel zuordnen, so dass man die gesamte Steuerung des Spiels über Inputs selbst nach eigenen Wünschen konfigurieren kann. `DirectInput` ermöglicht dies beispielsweise über das Action-Mapping. Dabei handelt es sich aber um recht komplexe Funktionen, um den gesamten Input frei konfigurierbar zu machen. Ebenfalls aufgrund der Kürze des Themas Input, auf das wir hier explizit keinen Schwerpunkt legen, behandeln wir diese Option hier auch nicht weiter.

Flexibilität

In diesem Kapitel haben wir der Grundstruktur unserer Game-Engine einen großen Brocken hinzugefügt. In den folgenden Kapiteln wird es darum gehen, den Funktionsumfang unserer Engine noch so weit auszubauen, dass wir sie am Ende dieses Buches wirklich als Game-Engine bezeichnen können. Im nächsten Kapitel entwickeln wir daher eine Audio-Engine für die `ZFXEngine`, mit der wir Soundeffekte und Musikstücke abspielen können.

Ausblick

Nun ist es also an der Zeit, den richtigen Sitz der Sound-Karte zu überprüfen und den Lautstärke-Regler auf 20 zu drehen.

10 Audio-Interface der ZFXEngine

» Wer ganz Ohr ist, hört nicht.« (Moritz Heimann)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Initialisierung von DirectMusic
- ➔ Implementierung einer DLL zur Kapselung der Audio-Ausgabe
- ➔ Laden und Abspielen von Sound-Effekten
- ➔ 3D-Sound-Quellen und Position des Zuhörers

10.1 Kurz und schmerzlos

Um uns wertvolle Seiten für wichtigere Dinge freizuhalten, fasse ich mich so kurz wie möglich. In diesem Kapitel werdet ihr (hoffentlich) viele Parallelen zu vorherigen Kapiteln entdecken. Der Aufbau des hier entwickelten Codes entspricht in seinen Grundzügen dem Aufbau unseres `ZFXRenderDevice`-Objekts und dem des `ZFXInputDevice`-Objekt. Konsequenterweise entwickeln wir hier das so genannte `ZFXAudioDevice`-Objekt, das natürlich in unserer ZFXEngine dazu dienen wird, akustische Signale über die Lautsprecher des Computers auszugeben. Damit ist dieses Objekt dann die dritte von den insgesamt vier DLL-Komponenten, die den Kern unserer Game-Engine bilden.

*Eine weitere
Engine-
Komponente*

Wer mit dem Aufbau unserer Engine noch nicht wirklich vertraut ist, sollte diesbezüglich in Kapitel 2 nachschlagen. Und allen, die ihr Wissen über die Implementierung eines Interfaces in einer DLL, die über eine statische LIB geladen wird, auffrischen möchten, kann ich nur das dritte Kapitel ans Herz legen.



An dieser Stelle verweise ich wieder einmal auf die wirklich gute Dokumentation des DirectX SDK. Im Verlauf dieses Kapitels werden wir die DirectMusic-Komponente von DirectX verwenden, um unser Audio-Interface zu implementieren. Im DirectX SDK findet ihr diverse Beispiele und Erklärungen zur Verwendung von DirectMusic, daher werde ich die notwendigen

SDK-Doku

Erklärungen auf ein Minimum beschränken. Ich setze hier voraus, dass ihr bereits einige Erfahrung im Umgang mit DirectX gesammelt habt und daher der Implementierung hier ohne Probleme folgen könnt. Wer hier noch Nachholbedarf verspürt, dem empfehle ich die DirectMusic-Tutorials der DirectX SDK-Dokumentation.

Altbekanntes Interface-Design

Encore une fois

Genau wie bei den vorangegangenen Projekten erstellen wir auch hier eine Interface-Struktur und eine Implementierung dieses Interfaces in einer DLL, die über eine statische LIB geladen werden kann. Der Grundaufbau des Projekts ist ganz analog zu den bisherigen beiden Beispielen (Renderer und Input). Das Projekt mit der statischen Bibliothek nennen wir ZFXAudio, das Projekt mit der dynamischen Bibliothek erhält die Bezeichnung ZFXDA für DirectAudio. Die folgenden Aufzählungen zeigen die Dateien, die wir jeweils in den Projekten anlegen.



In DirectX gibt es die beiden Komponenten DirectSound und DirectMusic, wobei DirectMusic aber durch eigene Methoden auch auf DirectSound-Interface-Objekte zugreifen kann. Daher verwendet man hier die Bezeichnung DirectAudio, obwohl es keine gleichnamige Komponente im DirectX SDK gibt – jedenfalls noch nicht.

Dateien für ZFXAudio:

- ➔ ZFXAudio.h und ZFXAudio.cpp
- ➔ ZFXAudioDevice.h
- ➔ ZFX.h
- ➔ ZFX3D.h

Wie gehabt stehen die ersten beiden Dateien für die Implementierung der statischen Bibliothek bereit. Der Header ZFXAudioDevice.h enthält wie gewohnt die Definition des Interfaces. Ebenfalls wieder mit von der Partie ist der Allround-Header ZFX.h unserer Engine, mit all seinen Definitionen, Datenstrukturen usw. Daher benötigen wir auch wieder den Header ZFX3D.h unserer 3D-Mathe-Bibliothek, da einige Strukturen im Allround-Header auch Vektoren und Ähnliches aus der Mathe-Lib beinhalten. Allerdings haben wir hier auch die Situation vorliegen, dass wir zumindest den Vektortyp auch für unsere Audio-DLL benötigen werden, nämlich dann, wenn wir unseren Sound dreidimensional machen werden. Aber dazu später mehr. Die Implementierung der statischen Bibliothek zeige ich an dieser Stelle nicht noch einmal, da sich an ihr außer dem Namen des Interfaces und der zu ladenden DLL nichts ändert. Den kompletten Quellcode findet ihr aber natürlich auf der CD-ROM zu diesem Buch.

Dateien für ZFXDA:

- ➔ ZFXDA.def
- ➔ ZFXDA.h und ZFXDA.cpp

Das eigentliche DLL-Projekt ist hier in aller Kürze abgehandelt. Wie wir gleich sehen werden, benötigen wir nicht viel Code, um Sound laden und abspielen zu können. Auch dreidimensionaler Sound lässt sich über DirectMusic beinahe unheimlich einfach implementieren. Allerdings muss ich dazu sagen, dass wir uns hier auf das Nötigste beschränken. Bei einer kompletten Game-Engine würden sicherlich noch zusätzliche Gimmicks in der Audio-DLL implementiert werden.

Interface-Definition für eine Audio-Klasse

Bevor wir uns aber nun in die unendlichen Weiten von DirectMusic stürzen, folgen wir unserer angenehmen Angewohnheit, unsere Arbeit im Vorfeld zu strukturieren. An dieser Stelle fragen wir uns also zunächst: Was soll unsere Audio-Engine überhaupt alles leisten, und wie soll sie zu bedienen sein? Genau diese Fragestellung wird uns zu der Definition des entsprechenden Interfaces führen. Bevor wir also anfangen, uns auf DirectMusic-Interfaces zu stürzen und sie wild zu umkapseln, gehen wir lieber den umgekehrten Weg. Schließlich ist es nicht unsere Absicht, den schönsten Layer um DirectX-Komponenten zu legen, sondern wir wollen eine einfach zu verwendende, aber umfangreiche Engine entwickeln. Wie so ein Interface für den Sound sinnvollerweise aussehen könnte, das sehen wir hier:

Los geht's

```
class ZFXAudioDevice {
protected:
    HWND      m_hWndMain;      // Hauptfenster
    HINSTANCE m_hDLL;          // DLL-Handle
    bool       m_bRunning;      // nach Init

public:
    ZFXAudioDevice(void) {};
    virtual ~ZFXAudioDevice(void) {};

    virtual HRESULT Init(HWND, const char*, bool)=0;
    virtual void Release(void)=0;
    virtual bool IsRunning(void)=0;

    // Stoppt gesamte Audio-Ausgabe
    virtual void StopAll(void)=0;

    // Lädt einen Sound aus einer Datei
    virtual HRESULT LoadSound(const char*, UINT*)=0;
```



```

// Spielt einen bestimmten Sound
virtual void PlaySound(UINT, bool bLoop)=0;

// Stoppt einen bestimmten Sound
virtual void StopSound(UINT)=0;

// Zuhörer-Parameter
virtual void SetListener(ZFXVector vPos,
                        ZFXVector vDir,
                        ZFXVector vUp,
                        ZFXVector vV)=0;

// Parameter der Soundquelle
virtual void SetSoundPosition(ZFXVector, UINT)=0;
virtual void SetSoundDirection(ZFXVector,
                               ZFXVector vV,
                               UINT)=0;

virtual void SetSoundMaxDist(float, UINT)=0;
}; // class

```

KISS Wie ihr seht, sollte das Audio-Interface nicht viel mehr bieten als die Initialisierung, den Shutdown sowie natürlich das Laden und Abspielen von Sounds. Nach außen hin haben wir also scheinbar eine recht dünne Funktionalität der DLL, doch das ist ja auch der Sinn der Übung. Unsere Engine soll ja Wasser in den Händen des Spiele-Programmierers sein, der normalerweise nicht viel mehr Funktionalität braucht.

:-) TIPP

Das Motto des ehemals sowjetischen Flugzeug-Konstruktionsbüros MIG lautete: »Ein System, das nicht in ein Flugzeug eingebaut ist, kann garantiert nicht ausfallen!« Ähnliches gilt auch für die Programmierung komplexer Software. Je weniger Funktionen die Software bietet, desto weniger Bugs kann sie haben.

Natürlich sinkt damit insgesamt auch die Funktionalität, die die Software (oder eine MIG) zu bieten hat. Hier gilt es also, die Gratwanderung zwischen den beiden konkurrierenden Zielen größtmögliche Einfachheit und größtmögliche Funktionalität zu wagen. Dies bezeichnet man gemeinhin auch als KISS-Prinzip und meint damit: Keep It Simple, Stupid.

10.2 Implementierung des Interfaces

Nachdem wir also ein schlankes Interface definiert haben, ist nun der große Moment gekommen, die tatsächliche Implementierung in Angriff zu nehmen. Dabei werden wir jetzt DirectMusic *on-the-fly* lernen bzw. bereits vorhandenes Grundwissen über diese DirectX-Komponente vertiefen. Ich zeige hier zwar alle verwendeten Funktionen und erkläre deren Sinn, sofern sie nicht selbstverständlich sind, aber ich werde nicht einzeln die Prototypen

und alle möglichen Optionen an Parametern durchgehen. Dazu könnt ihr notfalls die DirectX-Dokumentation heranziehen.

ZFXAudio Klasse

Da wir es hier mit Audio zu tun haben, kommen wir garantiert im Verlauf unserer Implementierung an die Stelle, an der wir es mit verschiedenen Sound-Files zu tun haben werden. Daher bauen wir jetzt schon einmal vor und erzeugen eine Struktur, in der wir so einen Sound speichern werden. Dabei lernen wir auch gleich ein wenig mehr über DirectMusic.

*Sounds in der
Engine speichern*

```
typedef struct ZFXSOUND_TYPE {
    char                *chName;
    bool                bChanged;
    IDirectMusicSegment8 *pSegment;
    IDirectMusicAudioPath8 *p3DPath;
    IDirectSound3DBuffer8 *p3DBuffer;
} ZFXSOUND;
```



Die ersten beiden Felder unserer Struktur sind wohl selbsterklärend. Zum einen haben wir hier den Dateinamen eines Sounds gespeichert, um ihn identifizieren zu können. Zum anderen haben wir einen `bool`-Parameter, der anzeigt, ob sich an dem Sound etwas geändert hat. Dies benötigen wir in der Regel nur für 3D-Sounds, wenn diese ihre Position, Geschwindigkeit oder Ausrichtung verändern. Die folgenden drei Felder sind Interface-Objekte von DirectMusic bzw. DirectSound. Dazu holen wir nun ein wenig weiter aus. Bei DirectMusic geht quasi nichts ohne einen so genannten *Audiopath*. Dieser managt sozusagen den Fluss von Audiodaten durch DirectMusic und ist in jeder DirectMusic-Applikation mindestens einmal implizit vorhanden, nämlich in Form des Default-Audiopath-Objekts. Ein echter Musiker kann nun allerlei Schweinereien mit so einem Audiopath machen, aber wir beschränken uns hier auf die einfachste Sound-Ausgabe. Für diese benötigen wir so einen Audiopath eigentlich nicht. »Eigentlich« sage bzw. schreibe ich deshalb, weil wir ihn eben doch einmal für jeden Sound benötigen, und zwar dann, wenn wir einen DirectSound-3D-Buffer für einen Sound haben möchten.

*DirectMusic,
Baby, DirectMusic*

Damit sind wir schon beim 3D-Sound. Glücklicherweise erledigt DirectSound alles für uns, was mit der Berechnung des Sounds zu tun hat. Uns obliegt nur die Aufgabe, dem Sound seine Position und Orientierung in der 3D-Welt mitzuteilen. Dazu brauchen wir einen Interface-Pointer vom Typ `IDirectSound3DBuffer8` auf jeden unserer Sounds, und an diesen kommen wir nun über den Audiopath des Sounds heran.

*DirectSound und
3D*

Doch bisher haben wir den Sound ja noch nirgendwo gespeichert, nicht wahr? Dazu verwendet man in DirectMusic ein so genanntes *Segment*. Ein Segment ist sozusagen die Struktur, in der DirectMusic die Daten des Sounds speichert. Wenn wir einen Sound laden, dann laden wir ihn direkt aus der Datei in ein

*DirectMusic
Segmente*

solches Segment. Mehr brauchen wir darüber nicht zu wissen. Wer schon einmal mit DirectSound gearbeitet hat, der kennt eventuell das Konzept des primären und der sekundären Sound-Buffer-Objekte. Mit den DirectMusic-Segmenten verhält es sich genauso. Man kann ein solches Segment direkt als primäres Segment über DirectMusic abspielen lassen. Von diesen Segmenten gibt es immer nur eins, und das bedeutet, es läuft dann ausschließlich dieses eine Segment. Arbeiten wir mit mehreren Sounds, dann geben wir diese beim Abspielen als sekundäre Segmente an. DirectMusic nimmt dann das sekundäre Segment und mischt es automatisch in das primäre Segment. So kann man beliebig viele Sounds gleichzeitig abspielen lassen.

Doch nun genug der Worte. Hier ist die tatsächliche Klassen-Definition, mit der wir unser ZFXAudioDevice-Interface implementieren:



```
class ZFXDA : public ZFXAudioDevice {
public:
    ZFXDA(HINSTANCE hDLL);
    ~ZFXDA(void);

    HRESULT Init(HWND, const char*, bool);

    // Interface-Funktionen
    void    Release(void);
    bool    IsRunning(void) { return m_bRunning; }

    void    SetListener(ZFXVector, ZFXVector,
                      ZFXVector, ZFXVector);

    HRESULT LoadSound(const char*, UINT*);
    void    PlaySound(UINT nID, bool);
    void    StopSound(UINT nID);
    void    StopAll(void) { if (m_pPerformance)
                          m_pPerformance->Stop(NULL, NULL, 0, 0); }

    void    SetSoundPosition(ZFXVector, UINT);
    void    SetSoundMaxDist(float, UINT);
    void    SetSoundDirection(ZFXVector, ZFXVector,
                          UINT);

private:
    IDirectMusicLoader8      *m_pLoader;
    IDirectMusicPerformance8 *m_pPerformance;
    IDirectSound3DListener8  *m_pListener;
    IDirectMusicAudioPath8   *m_pDAPath;
    DS3DLISTENER             m_dsListener;
    DS3DBUFFER               m_dsBuffer;
    ZFXSOUND                 *m_pSounds;
    UINT                     m_NumSounds;
}; // class
```

In dieser Klassen-Deklaration kommen wir tatsächlich mit den Funktionen aus, die das Interface vorschreibt. Wir benötigen lediglich einen ganzen Satz an zusätzlichen Attributen, die – abgesehen von den beiden letztgenannten – allesamt aus DirectMusic und DirectSound stammen. Den Audiopath kennen wir ja bereits. Für das Hauptobjekt benötigen wir zwar nicht wirklich den Audiopath, aber für den Fall, dass man die Funktionalität der Audio-DLL später noch erweitern möchte, ist es sehr wahrscheinlich, dass man den Default-Audiopath benötigt. Für diesen Fall zeige ich gleich, wie man den Default-Audiopath erhält.

Attribute

Dann haben wir einen DirectMusic-*Loader*, der logischerweise dazu dienen wird, Sound-Daten zu laden. Dazu kommt ein DirectMusic-Performance-Objekt, das sozusagen das DirectMusic-Arbeitspferd ist, da es – anders, als wir es beispielsweise bei DirectInput und Direct3D gesehen haben – kein IDirectMusicDevice8-Interface oder Ähnliches gibt. Zu guter Letzt benötigen wir als Attribut auch noch ein DirectSound-*Listener*-Objekt, wenn wir 3D-Sound spielen möchten.

Die verbleibenden beiden Attribute sind zwei Strukturen von DirectSound, daher das Präfix DS3D. Diese brauchen wir, um die Parameter für den 3D-Sound zu setzen. Damit haben wir alle notwendigen Attribute zusammen und schreiten zur Konstruktion und Destruktion von Instanzen dieser Klasse.

Konstruktion und Destruktion

```
ZFXDA::ZFXDA(HINSTANCE hDLL) {
    m_hDLL      = hDLL;
    m_ploader    = NULL;
    m_pListener  = NULL;
    m_pPerformance = NULL;
    m_pSounds    = NULL;
    m_pLog       = NULL;
    m_bRunning   = false;

    // Strukturen initialisieren
    m_dsListener.dwSize = sizeof(DS3DLISTENER);
    m_dsBuffer.dwSize = sizeof(DS3DBUFFER);
} // constructor

ZFXDA::~ZFXDA() {
    Release();
} // destructor
```



Hier gilt es lediglich zu beachten, dass das Feld `dwSize` der beiden DirectSound-Strukturen mit der korrekten Größe initialisiert sein muss. Gehen wir nun gleich weiter zur Initialisierung und der Freigabe einer Instanz.

Initialisierung und Freigabe

Initialisierung und COM

Unglücklicherweise gibt es für DirectMusic keine eigene `Create ... ()`-Funktion so wie bei den anderen DirectX-Komponenten. An dieser Stelle ist die DirectX-Architektur ein wenig inkonsistent. Daher müssen wir hier einfach das zu Fuß machen, was die `Create ... ()`-Funktionen der anderen Komponenten sonst für uns erledigen: Wir müssen COM initialisieren.



Die Bezeichnung COM steht für Component Object Model und bezeichnet eine Spezifikation für die Implementierung von Software-Modulen. Die ersten Objekte, die die COM-Spezifikationen verwendeten, waren die OLE-Steuerelemente (auch OCX) und natürlich auch deren Derivat, die ActiveX-Komponenten. Auch die einzelnen DirectX-Komponenten sind unter Verwendung der COM-Spezifikationen implementiert worden.

Das klingt aber schlimmer, als es ist. Für die Initialisierung von COM benötigen wir lediglich die parameterlose Funktion `CoInitialize()` der WinAPI. Ist das geschafft, dann können wir über die Funktion `CoCreateInstance()` eine Instanz eines jeden beliebigen COM-Interfaces erzeugen. Und da DirectX streng nach dem COM-Prinzip aufgebaut ist, gilt das natürlich auch für DirectX-Interfaces wie beispielsweise ein *Loader*- oder ein *Performance*-Objekt von DirectMusic. Genau diese beiden müssen wir dann nämlich auch erzeugen, wozu wir der Funktion die entsprechenden Identifier und Interface-Namen angeben. Bevor wir jetzt noch die weiteren Schritte besprechen, schauen wir uns erst einmal den Code dazu an und klären danach, welche weiteren Schritte zur Initialisierung von DirectMusic notwendig sind.



```
HRESULT ZFXDA::Init(HWND hWnd, const char *chPath,
                    bool bSaveLog) {
    HRESULT hr;
    WCHAR   wPath[MAX_PATH];

    m_hWndMain = hWnd;
    g_bLF      = bSaveLog;

    // COM initialisieren
    CoInitialize(NULL);

    // Erzeuge eine Instanz des Loader-Objekts
    hr = CoCreateInstance(CLSID_DirectMusicLoader,
                        NULL,
                        CLSCTX_INPROC,
                        IID_IDirectMusicLoader8,
                        (void**)&m_pLoader);
    if (FAILED(hr)) return ZFX_FAIL;
```

```

// Erzeuge eine Instanz des Performance-Objekts
hr = CoCreateInstance(
    CLSID_DirectMusicPerformance, // Class-ID
    NULL,                        // Aggregat
    CLSCTX_INPROC,               // Kontext
    IID_IDirectMusicPerformance8, // Referenz-ID
    (void**)&m_pPerformance);    // Zieladresse
if (FAILED(hr)) return ZFX_FAIL;

// Setze den Default-Pfad für Sound-Dateien
if (MultiByteToWideChar(CP_ACP, 0, chPath, -1, wPath,
    MAX_PATH)==0)
    return ZFX_FAIL;

if ( FAILED ( hr = m_pLoader->SetSearchDirectory(
    GUID_DirectMusicAllTypes,
    wPath, false)))
    return ZFX_FAIL;

// Initialisiere das Performance-Objekt
if ( FAILED ( hr = m_pPerformance->InitAudio(
    NULL, NULL, hWnd,
    DMUS_ATH_SHARED_STEREOPLUSREVERB,
    64, DMUS_AUDIOF_ALL, NULL)))
    return ZFX_FAIL;

// Hole einen Pointer auf den Default-Audiopath
if (FAILED(m_pPerformance->GetDefaultAudioPath(
    &m_pDAPath)))
    return ZFX_FAIL;

// Hole einen Pointer auf das Listener-Objekt im Pfad
if (FAILED(m_pDAPath->GetObjectInPath(
    0, DMUS_PATH_PRIMARY_BUFFER,
    0, GUID_NULL, 0,
    IID_IDirectSound3DListener8,
    (void**)&m_pListener)))
    return ZFX_FAIL;

m_bRunning = true;
return ZFX_OK;
} // Init

```

Nachdem wir die beiden notwendigen DirectMusic-Objekte erhalten haben, setzen wir sogleich über die entsprechende Interface-Funktion `IDirectMusicLoader8::SetSearchDirectory` den Default-Pfad für zu ladende Sound-Dateien auf den entsprechenden Parameter, den die Initialisierungsfunktion beim Aufruf erhalten hat. Allerdings ist DirectMusic darauf ausgelegt, mit `WCHAR`- (Wide Char-)Strings zu arbeiten, daher müssen wir den normalen `char`-

Suchpfad setzen

String zuvor mittels der WinAPI-Funktion `MultiByteToWideChar()` entsprechend umwandeln.

**Hauptobjekt
initialisieren**

Damit wir aber `DirectMusic` richtig in Betrieb nehmen können – bisher haben wir ja nur Pointer auf die entsprechenden Interface-Objekte generiert – müssen wir zuerst noch die folgende Methode namens `IDirectMusicPerformance8::InitAudio` aufrufen. Hierbei handelt es sich um eine Standardinitialisierung wie beispielsweise im `DirectMusic-Tutorial` der SDK-Dokumentation, daher gehe ich nicht näher auf die einzelnen Parameter ein, als es bereits durch die Kommentare im Quelltext geschehen ist.

Default-Audiopath

Nach der erfolgreichen Initialisierung unseres `Performance`-Objekts rufen wir `IDirectMusicPerformance8::GetDefaultAudioPath` auf, um den `Default-Audiopath` abzufragen. Diesen benötigen wir in unserem Beispiel nur dazu, um ein `Listener`-Objekt über die Funktion `IDirectMusicAudioPath8::GetObjectInPath` zu erhalten. Diese Funktion sollten wir uns gut merken, denn über sie können wir aus einem `Audiopath` verschiedene andere Objekte abfragen. Wir erinnern uns ja noch dunkel, dass der `Audiopath` sozusagen der Weg ist, den ein Sound bei seiner Reise vom RAM durch `DirectMusic` über die Soundkarte zurücklegen muss, um zu unseren Ohren zu gelangen. In diesem Path finden sich daher auch alle die Objekte von `DirectMusic` und `DirectSound`, die dazu benötigt werden. Das `Listener`-Objekt behandeln wir später noch ausführlich, aber natürlich dient es dazu, die Position des Hörers in der 3D-Welt festzulegen, da wir ja 3D-Sounds verwenden.

Freigabe

So viel zur Initialisierung. Wenn wir eine Instanz unserer Klasse wieder freigeben wollen, rufen wir, wie im Destruktor gezeigt, die Funktion `ZFXDA::Release` auf. Diese gibt einfach alle zwischendurch initialisierten Objekte wieder frei, um keine Speicherlecks entstehen zu lassen. Die meisten dieser Objekte haben wir eben in der Initialisierungsfunktion bereits gesehen, aber die Liste `m_pSounds` mit unseren ganzen Sounds wird erst im Laufe der Anwendung der DLL aufgebaut.



```
void ZFXDA::Release() {
    if (m_pSounds) {
        for (UINT i=0; i<m_NumSounds; i++) {
            if (m_pSounds[i].pSegment) {
                m_pSounds[i].pSegment->Unload(
                    m_pPerformance);
                m_pSounds[i].pSegment->Release();
                m_pSounds[i].pSegment = NULL;
                delete [] m_pSounds[i].chName;
                m_pSounds[i].chName = NULL;
            }
        }
        free(m_pSounds);
    }
}
```

```

if (m_pLoader) {
    m_pLoader->Release();
    m_pLoader = NULL;
}
if (m_pListener) {
    m_pListener->Release();
    m_pListener = NULL;
}
if (m_pPerformance) {
    m_pPerformance->Stop(NULL, NULL, 0, 0);
    m_pPerformance->CloseDown();
    m_pPerformance->Release();
    m_pPerformance = NULL;
}

// COM herunterfahren
if (m_bRunning) CoUninitialize();
m_bRunning = false;
} // release

```

Bevor wir ein DirectMusic-Performance-Objekt freigeben dürfen, müssen wir jedoch auf alle Fälle alle gerade laufenden Sounds stoppen. Danach folgt ein `IDirectMusicPerformance8::CloseDown`-Aufruf, um alle internen Prozesse des Performance-Objekts zu stoppen und die entsprechenden Objekte freizugeben. Zuvor müssen wir aber selbst alle Objekte freigeben haben, die wir von dem Performance-Objekt abgefragt haben. Last but not least dürfen wir auch nicht vergessen, das COM-System der WinAPI herunterzufahren. Dies erledigt die parameterlose Funktion `CoUninitialize()` für uns.

Aufräumen

Das Laden und Abspielen von Sounds

Das Laden von Sound-Dateien aus allen gängigen Sound-Formaten wie beispielsweise Wave ist mit DirectMusic ein absolutes Kinderspiel. Wozu haben wir schließlich ein DirectMusic-Loader-Objekt? Und tatsächlich, dieses hat die Funktion `LoadObjectFromFile()`, mit der wir diese Funktionalität erhalten, ohne eine einzige Zeile Code zu schreiben. Da wir aber eine schöne Kapselung um DirectMusic herum anlegen, haben wir in unserer eigenen Ladefunktion dieser Klasse noch ein wenig mehr Arbeit. Als Erstes beginnen wir damit, den Dateinamen des zu ladenden Sounds in einen Wide-Char-String zu konvertieren. Danach durchlaufen wir den Pointer `m_pSounds` und schauen nach, ob wir den Sound nicht schon geladen haben. Das schont, genauso wie bei Materialien und Texturen, eine ganze Menge Ressourcen.

Ein Kinderspiel

Der Zeiger `m_pSounds` ist aber beim Laden des ersten Sounds noch nicht initialisiert. Wir verwenden hier dasselbe Prinzip wie auch schon bei den Texturen und Materialien. Das heißt, wir allokatieren immer Speicher für 50 neue Elemente in diesem Pointer, wenn unser Zähler anzeigt, dass wir nicht mehr genügend Platz für den nächsten zu ladenden Sound haben. Dann kopieren

m_pSounds

wir den Namen der zu ladenden Datei in den entsprechenden Slot und laden die Sound-Datei.

*Instrumente,
Pfade und 3D-
Buffer*

Danach sind noch drei weitere Schritte notwendig, damit wir mit diesem Sound auch nach unserem Gutdünken im 3D-Raum arbeiten können. Zuerst müssen wir in einem ersten Schritt die DirectMusic-Funktion `IDirectMusicSegment8::Download` aufrufen. Diese lädt alle verwendeten Instrumente aus dem Performance-Objekt für dieses Segment herunter. Dazu muss man wissen, dass es ganz grundsätzlich zwei verschiedene Methoden gibt, um Sound-Dateien zu speichern: zum einen durch digitale Samples und zum anderen instrumentenbasiert. Bei der erstgenannten Methode speichert man einfach die komplette Welle (z.B. beim *Wave*-Format), die durch den Sound erzeugt wird, digital ab. Dies verbraucht sehr viel Speicher, benötigt aber später keine Interpretation oder eine Berechnung der Daten zur Laufzeit. Bei der zweiten Methode speichert man einfach die Instrumente, mit denen der Sound erstellt worden ist, auch ab, und zwar zusammen mit den entsprechenden Nachrichten, welches Instrument zu welchem Zeitpunkt was für eine Aktion auszuführen hat. Das ist so, als würde man nicht den Sound selbst speichern, sondern lediglich den Komponisten bzw. seine Kommandos an die einzelnen Instrumente oder sein Notenblatt. Damit hat man wesentlich weniger Daten zu speichern, muss jedoch zur Laufzeit des Sounds noch Aktionen durchführen. Dann muss man nämlich jedem Instrument die entsprechenden Befehle zukommen lassen, wann es wie einzusetzen hat usw.

DLS-Format

Das *Downloadable Sounds*-Format (DLS), das auch DirectMusic verwendet, kombiniert beide Methoden. Hierbei werden zur Laufzeit die entsprechenden Instrumentendaten generiert, und bei DirectMusic müssen wir diese durch den Aufruf von `Download()` für unser Segment holen. Dies ist nur einmalig nötig. Normalerweise wären wir jetzt mit dem Laden eines Sounds fertig, wenn da nicht 3D wäre. Für gewöhnlich reicht es uns, wenn wir in unserer Audio-Applikation einen einzigen Audiopath haben. Wenn wir jedoch 3D-Sounds verwenden, müssen wir für jeden einzelnen 3D-Sound einen eigenen Audiopath generieren. Im dritten und letzten Schritt, der nun noch verbleibt, holen wir uns aus diesem speziellen Audiopath des 3D-Sounds einen 3D-Sound-Buffer. Diesen benötigen wir, um die Position und die anderen Parameter des Sounds jederzeit setzen zu können. Aus diesem Grund brauchen wir auch für 3D-Sounds einen eigenen Audiopath. Ansonsten können wir nicht unterschiedliche Parameter (z.B. Positionen) für die Sounds festlegen.

So, jetzt habe ich schon wieder viel zu viel erzählt. Hier ist die komplette Funktion:



```
HRESULT ZFXDA::LoadSound(const char *chName, UINT *nID)
{
    WCHAR    wName[MAX_PATH];
    HRESULT  hr;
```



```

if (MultiByteToWideChar(CP_ACP, 0, chName, -1, wName,
                        MAX_PATH)==0)
    return ZFX_FAIL;

// Haben wir diesen Sound schon geladen?
for (UINT i=0; i<m_NumSounds; i++) {
    if (strcmp(chName, m_pSounds[i].chName)==0) {
        *nID = i;
        return ZFX_OK;
    }
} // for

// 50 neue Memory-Slots für Sounds
if ( (m_NumSounds%50) == 0 ) {
    int n = (m_NumSounds+50)*sizeof(ZFXSOUND);
    m_pSounds = (ZFXSOUND*)realloc(m_pSounds, n);
    if (!m_pSounds) return ZFX_OUTOFMEMORY;
}

// temporärer Zeiger für das Segment
IDirectMusicSegment8 *pSeg =
    m_pSounds[m_NumSounds].pSegment;

m_pSounds[m_NumSounds].chName =
    new char[strlen(chName)+1];
memcpy(m_pSounds[m_NumSounds].chName, chName,
    strlen(chName)+1);

m_pSounds[m_NumSounds].bChanged = false;

// lade die Datei
if (FAILED(hr=m_pLoader->LoadObjectFromFile(
    CLSID_DirectMusicSegment, // Klasse
    IID_IDirectMusicSegment8, // Interface-Typ
    wName, // Dateiname
    (void**)&pSeg))) // Zieladresse
{
    if ( (hr==DMUS_E_LOADER_FAILEDOPEN) ||
        (hr==DMUS_E_LOADER_FAILEDCREATE) )
        return ZFX_FILENOTFOUND;
    else if (hr==DMUS_E_LOADER_FORMATNOTSUPPORTED)
        return ZFX_INVALIDPARAM;
    else if (hr==E_OUTOFMEMORY)
        return ZFX_OUTOFMEMORY;
    return ZFX_FAIL;
}

// mache einen Download der Instrumente
if (FAILED(pSeg->Download(m_pPerformance))) {
    pSeg->Release();
}

```

```

        pSeg = NULL;
        return ZFX_FAIL;
    }

    // erzeuge einen Audiopath für den Sound
    m_pPerformance->CreateStandardAudioPath(
        DMUS_ATH_DYNAMIC_3D,
        64, TRUE,
        &m_pSounds[m_NumSounds].p3DPath);

    m_pSounds[m_NumSounds].p3DPath->GetObjectInPath(
        DMUS_PCHANNEL_ALL,          // Performance-Kanal
        DMUS_PATH_BUFFER,          // Stage im Audiopfad
        0,                          // Index im Pfad
        GUID_NULL,                  // Klasse
        0,                          // Index
        IID_IDirectSound3DBuffer, // Typ
        (void**)                     // Zieladresse
        &m_pSounds[m_NumSounds].p3DBuffer);

    m_NumSounds++;
    return ZFX_OK;
} // LoadSound

```

Fertig! Nun haben wir einen Sound in ein freies Feld unseres Arrays `m_pSounds` gespeichert und können ihn endlich abspielen. Dies ist erfreulicherweise recht einfach. Diesmal zeige ich aber zuerst die Funktion, und erkläre dann noch ein wenig mehr dazu.



```

void ZFXDA::PlaySound(UINT nID, bool bLoop) {
    if (nID >= m_NumSounds)
        return;

    // irgendwelche neuen Änderungen?
    if (m_pSounds[nID].bChanged) {
        m_pListener->CommitDeferredSettings();
        m_pSounds[nID].bChanged = false;
    }

    if (bLoop)
        m_pSounds[nID].pSegment->SetRepeats(
            DMUS_SEG_REPEAT_INFINITE);

    // als sekundären Buffer abspielen
    m_pPerformance->PlaySegment(m_pSounds[nID].pSegment,
        DMUS_SEGF_DEFAULT |
        DMUS_SEGF_SECONDARY,
        0, 0);
} // PlaySound

```

Die Funktion kann auf Wunsch eine Wiederholung für den Sound setzen. Dieser wird dann so lange in einer Schleife gespielt, bis man ihn wieder stoppt. Dazu verwenden wir bei Bedarf die Funktion `IDirectMusicSegment8::SetRepeats`. Um den Sound dann abzuspielen `IDirectMusicPerformance8::PlaySegment`. Dabei geben wir im ersten Parameter das abzuspielende Segment für die Performance an, im zweiten Parameter ein paar Flags, und die letzten beiden Parameter ignorieren wir. Für unsere einfachen Sound-Anwendungen reichen die hier gezeigten Flags aus. Das erste Flag zeigt an, dass wir die Flags und Parameter des Segments selbst verwenden wollen beispielsweise seinen eigenen Audiopath und nicht den Default-Audiopath. Das zweite Flag zeichnet das Segment als Secondary-Segment aus. Dies ist notwendig, weil mehrere Secondary-Segmente (im Gegensatz zu Primary-Segmenten) gleichzeitig laufen können. Diese werden dann von DirectMusic automatisch in das einzige existierende Primary-Segment gemischt.

Wiederholung und Abspielen

Im dem oben gezeigten Code steht auch noch die Methode `IDirectSound3DLister8::CommitDeferredSettings`, und dazu müssen wir ein wenig vorgreifen. Wenn wir die Parameter eines 3D-Sounds ändern, also beispielsweise seine Position und Geschwindigkeit, so muss DirectMusic bzw. DirectSound den Sound auf der CPU neu berechnen und entsprechend anpassen. Dies ist eine relativ zeitaufwändige Angelegenheit und sollte daher nur dann gemacht werden, wenn man alle gewünschten Änderungen am Sound eingestellt hat. Wenn man neue Werte für die Parameter eines Sounds oder auch des Listeners selbst setzt, so kann man per Flag angeben, ob diese sofort berechnet oder erst mal nur vorgemerkt werden sollen. Hat man die Änderungen nur vorgemerkt, so werden sie erst aktiv, wenn wir die hier gezeigte Funktion aufrufen. Dazu dient auch das `bool`-Feld in der `ZFXSOUND`-Struktur. Wenn wir die Parameter eines Sounds ändern, dann setzen wir dieses Feld auf `true` und wissen damit, dass wir die Änderungen erst noch aktivieren müssen, wenn wir den Sound abspielen wollen.

Deferred Settings

Wo wir einen Sound starten können, da müssen wir ihn auch wieder stoppen können – beispielsweise dann, wenn wir einen Sound als Schleife spielen oder wenn die Hintergrundmusik eines Menüs anhalten soll, weil nun ein Level beginnt.

Stoppen

```
void ZFXDA::StopSound(UINT nID) {
    if (nID >= m_NumSounds)
        return;

    m_pPerformance->Stop(m_pSounds[nID].pSegment,
                        0, 0, 0);
} // StopSound
```



Nachdem das auch geklärt ist, kommen wir nun zu dem dreidimensionalen Aspekt der Sounds.

Zuhörer und Quelle für 3D-Sound

Den Zuhörer positionieren

Mit 3D kennen wir uns ja nun schon ein wenig aus, daher werden wir auch keine Probleme haben, das Konzept von 3D-Sounds zu verstehen. DirectMusic und DirectSound berechnen für uns alles Notwendige, um 3D-Sound abspielen zu können. Wir müssen dabei nur zwei Dinge tun: Zum einen müssen wir den beiden mitteilen, wo sich der Zuhörer im 3D-Raum befindet, und zum anderen müssen wir jeder 3D-Soundquelle ebenso mitteilen, wo sie sich im 3D-Raum befindet. Neben der Position des Zuhörers und der Sound-Quelle sind aber noch andere Daten notwendig, beispielsweise die Ausrichtung und die Geschwindigkeit. Die folgende Funktion dient dazu, diese Parameter für den Zuhörer zu setzen. In unserem Fall sind das die Daten des Spielers bzw. der Kamera im 3D-Raum.



```
size_t g_szoFF3 = sizeof(float)*3;

void ZFXDA::SetListener(ZFXVector vPos, ZFXVector vDir,
                       ZFXVector vUp, ZFXVector vSpeed)
{
    m_pListener->GetAllParameters(&m_dsListener);

    memcpy(&m_dsListener.vPosition, &vPos, g_szoFF3);
    memcpy(&m_dsListener.vOrientFront, &vDir, g_szoFF3);
    memcpy(&m_dsListener.vOrientTop, &vUp, g_szoFF3);
    memcpy(&m_dsListener.vVelocity, &vSpeed, g_szoFF3);

    if (m_pListener)
        m_pListener->SetAllParameters(&m_dsListener,
                                     DS3D_IMMEDIATE);
} // SetListener
```

Zuerst fragen wir mit der entsprechenden DirectMusic-Funktion `IDirectMusic3DListener8::GetAllParameters` die aktuellen Daten des Zuhörers ab. Dabei werden diese im Attribut `m_dsListener` unserer Klasse, also in der Struktur vom Typ `DS3DLISTENER` gespeichert. Dann ändern wir die entsprechenden Parameter in dieser Struktur und schreiben sie mit der zugehörigen Set-Funktion wieder in unser Listener-Objekt zurück. Dabei verwenden wir das Flag `DS3D_IMMEDIATE`, womit die Änderungen sofort von DirectMusic berechnet werden.

Sound-Quellen positionieren

Da wir davon ausgehen, dass wir den Zuhörer in jedem Frame einmal neu platzieren müssen, sofern wir eine bewegte Kamera verwenden, können wir die Änderungen auch sofort wirksam machen. Anders bei den 3D-Sounds: Hier haben wir, wie man bereits im Interface sehen konnte, drei Methoden, um Eigenschaften einer Sound-Quelle zu ändern: zuerst natürlich die Position, dann die Richtung und Geschwindigkeit und zu guter Letzt die Reichweite des Sounds. Wenigstens die ersten beiden Methoden werden mit Sicherheit öfter aufgerufen, sowohl beide in einem Frame nacheinander als

auch mal die eine und mal die andere. In diesem Fall wäre es töricht, beide Male DirectMusic alle Änderungen durchrechnen zu lassen. Daher verwenden wir hier als Flag `DS3D_DEFERRED`, womit die Änderungen erst dann aktiv werden, wenn wir die Funktion `CommitDeferredSettings()` aufrufen, die wir in der Funktion `ZFXDA::PlaySound` bei Bedarf aufrufen.

Hier sind nun die Funktionen, mit denen wir die entsprechenden Parameter eines 3D-Sounds ändern können:

```
void ZFXDA::SetSoundPosition(ZFXVector vPos, UINT nID) {
    IDirectSound3DBuffer8 *p3DBuffer;
    if (nID >= m_NumSounds) return;

    p3DBuffer = m_pSounds[m_NumSounds].p3DBuffer;
    m_pSounds[m_NumSounds].bChanged = true;

    p3DBuffer->GetAllParameters(&m_dsBuffer);

    m_dsBuffer.dwMode = DS3DMODE_NORMAL;
    memcpy(&m_dsBuffer.vPosition, &vPos, g_szoff3);

    p3DBuffer->SetAllParameters(&m_dsBuffer,
                               DS3D_DEFERRED);
} // SetSoundPosition
```

```
void ZFXDA::SetSoundDirection(ZFXVector vDir,
                              ZFXVector vV, UINT nID) {
    IDirectSound3DBuffer8 *p3DBuffer;
    if (nID >= m_NumSounds) return;

    p3DBuffer = m_pSounds[m_NumSounds].p3DBuffer;
    m_pSounds[m_NumSounds].bChanged = true;

    p3DBuffer->GetAllParameters(&m_dsBuffer);

    m_dsBuffer.dwMode = DS3DMODE_NORMAL;
    memcpy(&m_dsBuffer.vVelocity, &vV, g_szoff3);
    memcpy(&m_dsBuffer.vConeOrientation, &vDir, g_szoff3);

    p3DBuffer->SetAllParameters(&m_dsBuffer,
                               DS3D_DEFERRED);
} // SetSoundDistance
```

```
void ZFXDA::SetSoundMaxDist(float fDis, UINT nID) {
    if (nID >= m_NumSounds) return;
    IDirectSound3DBuffer8 *p3DBuffer;
    if (nID >= m_NumSounds) return;
```



```
p3DBuffer = m_pSounds[m_NumSounds].p3DBuffer;  
m_pSounds[m_NumSounds].bChanged = true;  
  
p3DBuffer->GetAllParameters(&m_dsBuffer);  
  
m_dsBuffer.dwMode = DS3DMODE_NORMAL;  
m_dsBuffer.flMaxDistance = fDis;  
  
p3DBuffer->SetAllParameters(&m_dsBuffer,  
                           DS3D_DEFERRED);  
} // SetSoundMaxDist
```

10.3 Demo-Applikation zur Anwendung der DLL

Kurz und knackig

Das kleine Demo-Programm zu diesem Kapitel ist kurz und knackig. Ihr könnt dort ganz einfach über den Open-File-Dialog der WinAPI eine Sound-Datei auswählen, die daraufhin in einer Schleife abgespielt wird. Eine reine Sound-Demo ist einfach langweilig, solange man noch keine Grafik hat, die den Sound unterstützt. Aber wir werden am Ende des Buches erleben, wie der Sound in einem richtigen Spiel eingesetzt werden kann.

10.4 Ein Blick zurück, zwei Schritt nach vorn

Eine weitere Komponente

Im Verlauf dieses Kapitels haben wir gesehen, wie man der *ZFXEngine* eine weitere Komponente API-unabhängig hinzufügen kann. Unsere Engine nähert sich damit einem Zustand an, in dem wir sie wirklich für multimediale Applikationen einsetzen können. Wir haben nun die Möglichkeit, Grafik auszugeben. Wir können den Input des Anwenders verarbeiten und sind damit wirklich interaktiv. Nun können wir auch noch Sound-Effekte und Musik-Stücke erklingen lassen und haben damit eigentlich unsere Multimedia-Pflicht und -Schuldigkeit getan.

Netzwerke

Doch fehlt uns noch eine bedeutsame Komponente, die wir im nächsten Kapitel entwickeln werden. Dabei handelt es sich um die Integration der Netzwerk-Funktionalität in unsere Engine.

Denn was wäre ein echtes Multimedia-Spiel heutzutage, wenn man es nicht mit jedem beliebigen Menschen auf der Welt über das Internet spielen könnte? Oder eben mit dem Kumpel von nebenan über das hauseigene LAN? Bevor wir uns also noch lange von dem Audio-Kram aufhalten lassen, blättern wir gleich weiter zum nächsten Kapitel.

11 Netzwerk-Interface der ZFXEngine

»Stelle mich nicht auf die Probe wie einen schwächlichen Knaben
oder ein Weib, das nichts versteht von den Werken des Krieges.«
(Hektor, Homers Ilias)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Verschiedene Arten von Netzwerkspielen
- ➔ Technische Gestaltung von Netzwerken
- ➔ Peer-to-Peer- und Client-Server-Architekturen
- ➔ Implementierung einer Netzwerk-DLL für die ZFXEngine
- ➔ Anwendung der DLL für Chat- und Dateitransfer-Applikationen

11.1 Netzwerk-Spiele

Heutzutage wird kaum noch ein Spiel ohne Netzwerk-Support auf den Markt gebracht. Dabei beschränken sich die Hersteller nicht darauf, dass ein zweiter Rechner im gleichen lokalen Netz, also zum Beispiel bei uns zu Hause, mit dem anderen Rechner im Netz spielen kann, sondern weltweit über eine Verbindung im Internet. Diese Internetverbindung ermöglicht in sehr vielen Fällen das Spielen mit Tausenden Gleichgesinnten in einer virtuellen Welt.

*Weltweite
Vernetzung auch
bei Spielen*

Ein solches Spiel kann einfach ein Deathmatch in einem First-Person-Shooter sein, wo es nur darum geht, so viele *Fraggs*, also Abschüsse, wie möglich zu erzielen. Die Messlatte hat momentan bei den Spielen *Ultima Online*, *Diablo 2* und demnächst bei *Starwars Galaxies* ein Ende. Hier kann man große Welten gemeinsam mit Freunden erkunden, erobern, verteidigen, aufbauen usw. Man kann fast sagen, dass ein Spieler dort virtuell leben kann. Teilweise bekommen virtuelle Gegenstände in den Spielen einen echten realen Preis. So wurde der erste Level-70-Kämpfer in *Diablo 2* für über 5.000 US-\$ auf Ebay verkauft, und für eine Überweisung von virtuellen 100.000 Credits im Spiel *Star Wars Galaxies* werden bei Ebay Gebote von bis zu 100 € gemacht.

Fragger

Schauen wir uns die einzelnen Möglichkeiten, die uns durch das Netzwerk und das Internet gegeben wurden, einmal genauer an.

Session-basiert

Zug um Zug

Hier hat jeder beteiligte Spieler einen Client auf dem PC, mit dem er seinen letzten bekannten Status verwalten kann. Typisch für diese Kategorie von Spielen waren Aufbau- und Strategie-Spiele wie z.B. *GalaxyWars* oder noch einfacher Schach. Der Spieler hat mit seinem Client den nächsten Zug bestimmt, indem er in seinem Client zum Beispiel seine Flotten bewegt, Rohstoffe zugeteilt und Produktionen bestimmt hat. War der Zug abgeschlossen, so wurde das Informationspaket mit den neuen Zugdaten an eine E-Mail-Adresse oder direkt an einen Server versendet. Dieser hat so lange gewartet, bis entweder seine maximale Wartezeit pro Runde vorbei war oder alle Spieler ihren Zug eingesandt hatten. Dann hat der Server all die Züge ausgewertet und den Spielern über das gleiche Medium die neuen Daten zurückgesandt, in denen alle Züge aller anderen Spieler berücksichtigt worden waren. So ein Zug konnte dann schon mal eine Woche oder länger an Zeit kosten, die Spiele dauerten auch oft länger als ein Jahr.

Die Vorläufer von diesen Spielen waren die Post-Rollenspiele, wo auf den Befehlsbögen die Befehle dem Spielleiter per Post zugesandt wurden. Hier waren maximal ein bis zwei Züge pro Monat möglich.

Keine persistenten Daten

Ein anderes Merkmal der meisten session-basierten Spiele ist, dass sie keine persistenten Daten auf dem Server und auch nicht auf den Clients gespeichert haben, wenn man von den Highscore-Listen einmal absieht. Der Spieler startet das Spiel, ohne dass dieses einen Bezug zu seinen vorherigen Sessions hat. Es werden keine Spielstände wie beispielsweise Lebensenergie, Inventar, Waffen und Munition gespeichert. Alle Spieler haben zu Beginn einer solchen Session dieselben Voraussetzungen.

Deathmatch

Jeder gegen jeden

In der Regel hostet ein Server ein Deathmatch-Szenario, also eine Karte oder einen Level. Die Spieler klinken sich mit ihren Spielen, die die Clients darstellen, auf dem Server ein und sind als »reale« Spieler in dem Level für alle anderen sichtbar. Wenn eine Spielfigur nicht durch einen realen Spieler gesteuert wird, werkelt hier eine KI (künstliche Intelligenz) in Form eines Bots im Hintergrund. Jetzt geht es nur darum, so viele andere Spieler so oft wie möglich zu erledigen. Einmal jemanden zu erledigen heißt im Jargon ein *Frag*, und die Kill-Rate heißt daher *Fragrate*. Das Spiel ist entweder nach einer definierten Zeit zu Ende oder nach einer definierten Anzahl von notwendigen Frags. Ein Deathmatch ist also ein Spiel jeder gegen jeden, in dem kein tieferes Ziel als das *Fraggen* verfolgt wird. *Quake 3* von *id Software*, *Unreal* von *Epic* und *Half-life* von *Valve* sind die bekanntesten Spiele dieser Kategorie.

Team und Coop

Bei Team- und Coop-Spielen steht natürlich auch ein Server im Hintergrund, meistens im Internet, auf dem sich die Spieler wie oben bei Deathmatch-Spielen beschreiben, einklinken. Der Unterschied ist, dass hier in Teams gespielt wird oder sogar kooperativ ein Level gemeinsam bestritten werden kann. Das beste Beispiel dafür ist wohl *Battlefield 1942* von EA Games. Hier kann ein Spieler zum Beispiel einen Panzer fahren, dessen Kanone oder das MG bedienen, zu Fuß im Gelände kämpfen oder gar Flugzeuge und Schiffe benutzen, um den Feind zu fraggen. Bei Team-Deathmatch-Spielen gibt es in der Regel zwei Teams, die jeweils zusammen spielen und das andere Team zum Feind haben.

Eine unschöne Erscheinung bei dieser Art von Spielen sind die so genannten *T'kers*, die *Team Killers*. Dabei handelt es sich um verwirrte Menschen, die aus Dummheit oder bösartiger Absicht heraus die Spieler aus dem eigenen Team fraggen. Das kann sehr nervig sein, wenn jemand das mit Absicht betreibt, und so bieten viele Netzwerk-Spiele auch die Option an, das *Friendly Fire* zu deaktivieren. Dann hat das Waffenfeuer des eigenen Teams keine so hohe oder gar keine Auswirkung auf die Teammitglieder. Dabei geht natürlich viel vom Realismus des Spiels verloren. Eine andere Variante sind automatische *Kicks*. Das bedeutet, ein Server schmeißt alle Spieler einfach aus dem Spiel, die eine gewissen Anzahl an Team-Kills auf dem Gewissen haben. Einmal ist keinmal und kann im Kampfgetümmel durchaus vorkommen. Aber bei oben erwähntem *Battlefield 1942* ist es ein oft vorkommendes Problem, dass Spieler zu T'kers werden, um an bestimmte Fahrzeuge zu kommen, die andere Mitglieder des Teams vor ihnen erreichen würden oder dass sie diese zerstören (und mit ihnen den Fahrzeugführer), um einen Respawn (neues Auftauchen) des Fahrzeugs zu erzwingen.

*T'kers, Friendly
Fire und Kicks*

Capture the Flag

In diesem Spielmodus hat jedes Team eine oder mehrere Flaggen, und die Spieler, die sich auf dem Server eingeloggt haben, verteilen sich auf zwei oder mehr Teams. Jetzt versuchen die Teammitglieder, die Flagge von den anderen zu klauen und in ihr heimisches Lager zu schaffen. Dafür gibt es dann Punkte. Wer in einer bestimmten Zeit die meisten Punkte hat, der hat gewonnen. *Team Fortress* von Valve, das auf der *Half-life*-Engine basiert, ist ein klassisches Beispiel. Es gibt auch diverse Abwandlungen dieses Spielprinzips, wie beispielsweise *Capture the Hill* im Spiel *Delta Force* von Nova Logic. Dabei muss ein Team eine bestimmte Stellung feindfrei halten und das Team, das die Stellung für die längste Zeitspanne besetzt hielt, gewinnt das Spiel.

Go for the Flag!

Persistente Welten

Speicherung der Daten auf dem Server

Ein echtes *virtuelles* Zuhause bekommen die Spieler in persistenten Welten. Hier verwalten viele Datenbankserver all die Spieler, die sich in dieser Welt angemeldet haben. Diese Spiele werden in der Regel monatlich mit einem Pauschalbetrag bezahlt. Sie sind immer online, weltweit erreichbar, und man kann dort mit vielen Gleichgesinnten online spielen. Hier werden ganze Städte und Kontinente von »realen« Spielern aus der ganzen Welt bevölkert. Es werden entweder gemeinsam Abenteuer erlebt, oder es wird auch gegeneinander gekämpft. Es gibt wie im realen Leben politische und hierarchische Systeme, in denen sich der Spieler hocharbeitet.

Leveln

Solche Spiele bieten in der Regel ein ausgeklügeltes System aus Fähigkeiten und Eigenschaften des Characters, die ein Spieler durch das gezielte Spielen des Spiels verbessern kann. Dies nennt man im Jargon auch *leveln*. Dadurch verbessert der Spieler kontinuierlich seinen Charakter, um besser in der Welt klarzukommen, seine Waffen und Ausrüstungsgegenstände effektiver einsetzen zu können und schwerere Missionen bestehen zu können. Solche Welten werden in allen Zeitaltern und Dimensionen gespielt, ob nun im tiefsten Urschleim der Erde oder in der hypermodernsten und fernsten Galaxie, die man sich ausdenken kann. Es ist einfach alles möglich. Dadurch, dass die Objekte »real« in der Welt genutzt werden, können sie teilweise auch real zum Beispiel über Ebay gekauft und gehandelt werden. Ich habe selbst schon mehrere magische Gegenstände auf Ebay ersteigert, das Geld überwiesen und mich mit dem Verkäufer zu einer definierten Zeit an einem bestimmten Ort zur Übergabe getroffen. Das ist hier gang und gäbe. Schaut einfach mal auf Ebay vorbei, was es in dieser Richtung alles gibt.

Ausflüge in die reale Welt und zurück

Beendet ein Spieler sein Spiel in dieser virtuellen Welt, um eine Auszeit in der realen Welt zu nehmen, weil er beispielsweise das Geld für den monatlichen Unkostenbeitrag für die Online-Welt verdienen muss, dann bleiben die Eigenschaften und Ausrüstungsgegenstände seines Characters natürlich erhalten. Der Server speichert diese und andere relevante Daten, wie beispielsweise den Gesundheitszustand der Figur und ihr virtuelles Geld, in eine Datenbank. Wenn der Spieler die virtuelle Welt wieder betritt, dann ist er an genau der Stelle, an der er das Spiel beendet hat und im selben Zustand. Die virtuelle Welt hingegen kann sich durchaus verändert haben. Sie ist nämlich 24 Stunden am Tag online und wird von anderen Spielern weltweit gespielt. Während ein Spieler also offline ist, dreht sich die virtuelle Welt trotzdem weiter. Dies ist eines der Probleme, die solche Spiele aufwerfen. Wer ein Vielspieler ist und sehr viel Zeit mit dem Spiel verbringen möchte und auch kann, der ist bei den meisten Spielen klar im Vorteil, weil er innerhalb weniger Tage oder Wochen sehr viele Stunden online ist und seinen Character entsprechend schnell stark verbessern kann. Gelegenheitspieler und solche, die pro Woche nur ein paar Stunden spielen können oder wollen, brauchen entsprechend viele Wochen, um ein ähnliches Ergebnis zu

erzielen, und in der Regel sind solche persistenten Welten in den unteren Leveln der Characters einfach saulangweilig zu spielen.

RPG versus MMORPG

Im Single-Player-Modus, also bei Spielen, die nicht über ein Netzwerk mit mehreren gespielt werden können, nennt man diese Art von Spielen *Role Playing Games* oder kurz RPG. Da die weltweite Verfügbarkeit einer persistenten virtuellen Welt aber ganz andere Dimensionen zu diesem Spielprinzip hinzufügt, nannte man diese Spiele dann ganz modern *Massively Multi-player Online Role Playing Game* oder kurz MMORPG.

Akronyme

Nun ist das Handling von Millionen von Spielern, die einen international erfolgreichen Titel spielen, nicht gerade einfach, und je mehr Spieler auf einem Server sind, desto komplizierter wird das Ganze zu managen, wenn man eine vernünftige Performance erreichen möchte. Es ist daher oftmals nicht so, dass das Spiel auf einem gigantischen Server läuft, auf den die ganze Welt zugreifen kann. Es gibt hier verschiedene Strategien, aber in der Regel verwendet man einfach mehrere Instanzen des Spiels auf verschiedenen Servern, zwischen denen ein Spieler mit einem bestimmten Character nicht wechseln kann. So existiert das virtuelle Universum auf jedem Server in einer unabhängigen Kopie, die mit den anderen Servern nichts zu tun hat. Wenn man mit seinen Freunden zusammen in der virtuellen Welt eines Spiels aktiv sein möchte, so müssen alle denselben Server verwenden. Eine andere Strategie ist es, verschiedene Teile des Spiel wie beispielsweise verschiedene Planetensysteme auf verschiedene Server zu packen und dann den Character eines Spiels beim Übergang von einem Planetensystem zu einem anderen entsprechend von einem Server zu einem anderen umziehen zu lassen, inklusive aller Daten aus der Datenbank.

*Nicht wirklich
weltweit, aber
irgendwie schon*

LAG

Als letztes Fachwort fehlt in unserem Multiplayer-Jargon noch der Ausdruck LAG. Jeder der schon einmal ein Online-Spiel gespielt hat, wird vermutlich die Situation kennen, dass ein anderer Spieler über die Messaging-Funktion des Spiels »*goddam' fuckin' laaaaag*« geschrieben hat. Wenn die Verbindung vom Client zum Server nicht sonderlich schnell ist, dann spricht man von einem LAG und meint damit, dass die Berechnung des Spiels aufgrund der nur bruchstückhaft eintreffenden Daten vom Server extrem langsam ist. Scheinbar hat man eine sehr geringe Frame-Rate, und das Spiel ruckelt oder zeigt sogar für eine Sekunde oder noch länger ein Standbild. Die Frame-Rate ist jedoch in der Regel in Ordnung, denn das Rendern geschieht ja lokal auf dem Client. Doch die Daten zur Berechnung neuer Positionen vom Server kommen eben viel zu langsam herein, als dass die Darstellung in jedem Frame wirklich aktualisiert werden könnte. LAG stammt von dem Begriff *Latency*, der die Zeit bezeichnet, die ein Paket vom

Server zum Client benötigt. Verändern kann man das nur dann, wenn man selbst eine sehr langsame Internetverbindung hat. Oftmals liegt es aber auch an der Überlastung der Server, auf denen man gerade den Level spielt. Da lohnt es sich schon mal, einen Server zu suchen, der schneller ist.

11.2 Netzwerk-Architektur

Die Architektur für Netzwerke wird für gewöhnlich in zwei Kategorien unterteilt: in Peer-To-Peer und in Client-Server. Das hat nichts mit der physischen Verkabelung der EDV-Systeme zu tun, sondern damit, wie diese untereinander kommunizieren. Selbst auf einem einzigen Client können Peer-To-Peer- sowie Client-Server-Netze aufgebaut werden. Was das genau ist, schauen wir uns jetzt an...

Peer-to-Peer

Hier wird jeder Client mit jedem anderen Client verbunden. Das heißt nichts anderes, als dass jeder Client direkt mit jedem anderen Client frei kommunizieren kann. Es wird hierbei kein Server für die Kommunikation benötigt.

Vorteil:

- ➔ Ausfallsicherheit, ein Client kann einfach sich ausklinken, und die anderen können weitermachen.

Nachteile:

- ➔ Erhöhter Traffic, da ein Client zum Updaten seiner Position immer alle Clients informieren muss.
- ➔ Dadurch auf eine geringe Anzahl von Clients begrenzt.

Client-Server

Dedizierte Server

Ein einzelnes System wird als Server definiert. Das kann auch ein ganz normaler PC sein. Dieser hostet dann das Spiel, den Level oder die Welt. An diesem Server können sich die Clients anmelden und am Spiel teilnehmen. Wenn ein Server nur das Spiel hostet und keine Visualisierung hat, also nicht durch einen anderen Spieler zum Spielen genutzt wird, dann spricht man auch von einem dedizierten Server oder *dedicated Server*. An diesem Server muss sich jeder Client explizit anmelden und auch abmelden. Alle registrierten Clients werden mit den notwendigen Informationen zentral vom Server aus versorgt.

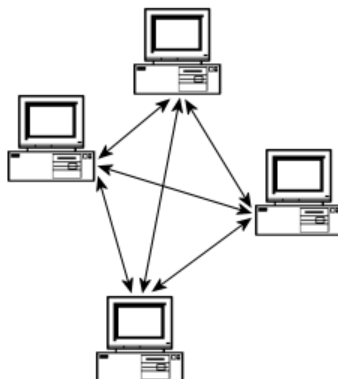


Abbildung 11.1:
Schematische Darstellung eines Peer-to-Peer-Netzwerks. Jeder einzelne Computer des Netzes ist mit jedem anderen direkt verbunden.

Dadurch können einfache Vorberechnungen, die für alle allgemein gültig sind, auf dem Server stattfinden und schneller an die Clients verteilt werden. Sämtliche Kommunikation in einem solchen Netzwerk läuft also über den Server. Hat ein Client eine Nachricht zu vermelden, beispielsweise, dass er einen anderen Client gefragt hat, dann sendet er diese an den Server und nicht an die Clients. Der Server entscheidet, wie mit dieser Nachricht weiter zu verfahren ist. Er wird normalerweise ihre Konsistenz prüfen und sie dann an alle Clients weiterverteilen.

In der Regel ist die Client-Server-Architektur auch die Technik, die überall angewendet wird. Wenn auf dem Server MMORPGs gehostet werden, dann werden von dort zum Beispiel die ganzen Datenbankserver angesteuert, in denen sich die Spieler bewegen. Das wäre über eine Peer-To-Peer-Architektur nicht möglich.

Vorteile:

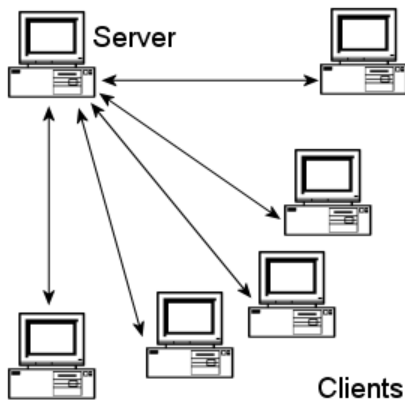
- ➔ Für große und komplexe Spiele ausgelegt (Datenbanken usw.)
- ➔ Schnellere Kommunikation mit den Clients
- ➔ Ein Client braucht nur noch mit einem Gegenpart zu kommunizieren, nämlich mit dem Server.
- ➔ Für Online-Welten geeignet

Nachteile:

- ➔ Single Point of Failure (zentraler Fehlerpunkt), d.h., wenn der Server ausfällt, kann keiner mehr spielen.
- ➔ Oftmals ist eigene Hardware für den Server nötig.

Abbildung 11.2:

Schematische Darstellung einer Client-Server-Architektur. Jeder Client-Computer im Netzwerk hat nur eine Verbindung im Netzwerk, nämlich die zu dem Server. Der Server hingegen hat zu jedem Client genau eine Verbindung.



11.3 Netzwerk-Technik

IPX, TCP, UDP und OSI

Die Systeme in einem Netzwerk unterhalten sich über Kupfer- oder Glasfaserkabel. Das ähnelt dem Telefonieren. Unsere Stimme wird so weit bearbeitet, dass sie zu der Telefonzentrale übertragen wird und von dort aus zu unserem gewünschten Ansprechpartner. Dass eine Applikation oder ein Spiel überhaupt eine Kommunikation zu einem anderen System aufnehmen kann, verdankt es dem Netzwerkprotokoll, das es nutzt. Es gibt verschiedene Netzwerkprotokolle, die bekanntesten sind sicherlich IPX/SPX, NETBEUI und natürlich TCP/IP das im Internet und dem World Wide Web verwendet wird. Etwas ältere Generationen von Fraggern, die beispielsweise *Doom 1* im Netzwerk gespielt haben, werden sich mit Tränen in den Augen daran zurückerinnern, wie die Endorphine und das Adrenalin den Körper durchströmten, wenn endlich IPX konfiguriert war, man den Computer des Kumpels über das serielle Kabel gefunden hatte und endlich zocken konnte. Es gibt aber auch noch viele weitere Protokolle. Als der absolute Standard hat sich das IP, also das *Internet Protocol*, als TCP und UDP durchgesetzt. Was genau TCP und UDP bedeutet, klären wir gleich. Zuerst sehen wir uns aber an, was OSI ist, denn das IP basiert auf dem OSI-Modell.

Das OSI-Modell

Das Akronym OSI steht für *Open System Interconnection*, ein Modell der Kommunikation, das in den späten 70er Jahren von der ISO (*International Organization for Standardization*) entwickelt wurde. Das OSI-Modell basiert auf sieben so genannten *Layern* (dt. Schichten), die aufeinander aufbauen. Eine Kommunikation fängt immer in der obersten Schicht bzw. auf dem obersten Layer an. Die Daten, die übertragen werden sollen, wandern dann durch alle Layer von oben nach unten. Sind sie am Ziel im Netzwerk

angekommen, dann wandern sie wieder von unten nach oben zur obersten Schicht. Dabei bearbeitet jeder Layer die Daten auf seine eigene Art. Eine wichtige Unterscheidung wird im Layer 4, der Transportschicht gemacht. Hier wird entschieden, ob das Paket als UDP (*User Datagram Protocol*, s.u.) oder als TCP (*Transmission Control Packet*, s.u.) versandt werden soll. Wir werden später die Unterschiede kennen lernen. Schauen wir uns zuerst das OSI-Modell in Abbildung 11.3an.

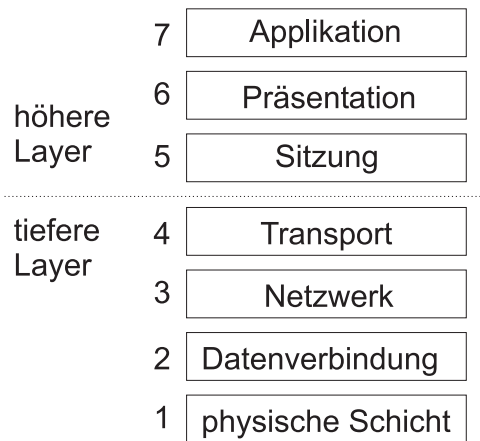


Abbildung 11.3:
Schematische Darstellung des OSI-Schichten-Modells. Die einzelnen Schichten werden im Text erläutert.

Layer 7: Anwendungsschicht

Hier sind die eigentlichen Nutzer der Netzwerkkommunikation angesiedelt. Das sind Programme wie unser E-Mail-Programm, ein FTP-Client oder einfach auch nur unser Internetbrowser. Hier geht es um jegliche Applikation, die ein Netzwerk-Protokoll verwenden möchte, um über ein Netzwerk zu kommunizieren.

Application Layer

Layer 6: Darstellungsschicht

Diese Schicht kümmert sich um eventuell komprimierte oder verschlüsselte Daten und definiert, wie die Daten für den Transfer aussehen sollen, daher auch der Name. Hier werden Daten also entsprechend verschlüsselt und entschlüsselt und auch komprimiert oder dekomprimiert, falls diese Features im Netzwerk genutzt werden sollen.

Presentation Layer

Layer 5: Sitzungsschicht

Hier wird bestimmt, wie eine Netzwerksitzung zwischen den beiden Verbindungspunkten, also diesem Client und einem Server, aufgebaut und beendet werden kann. Die Sitzung wird hier auch permanent synchron gehalten, so dass dem Computer durch diese Schicht klar wird, wann er Daten senden und wann er Daten empfangen kann.

Session Layer

Layer 4: Transportschicht

Transport Layer Abhängig von der gewählten Übertragungsmethode, UDP oder TCP, werden die Daten hier auf Fehler überprüft und in die passende Paketgröße optimiert. Zusätzlich wird überprüft, dass die Daten korrekt gesendet und empfangen werden.

Layer 3: Vermittlungsschicht

Network Layer Damit die Pakete auch in den komplexen Netzwerken zu ihrem Ziel finden, werden sie hier mit der Adresse ausgerüstet. Hier wird das so genannte IP (Internet Protocol) generiert.

Layer 2: Sicherungsschicht

Datalink Layer In dieser Schicht werden die Daten durch den Netzwerkkartentreiber weiterverarbeitet. Es findet eine weitere Fehlerprüfung der einzelnen Datenpakete statt. Zudem wird hier die physikalische Verbindung zur Kommunikation zwischen zwei Punkten hergestellt und der zu versendende Bitstream in kleine Häppchen zerlegt.

Layer 1: Bitübertragungssicht

Physical Layer In dieser Schicht werden die Daten in echte Lo/Hi-Signale auf dem Kabel umgewandelt, daher auch der Name. Die Bits werden hier tatsächlich physikalisch, also elektrisch übertragen.

Protokolle

Ohne Moos nichts Los, oder im Netzwerkbereich besser gesagt: »Ohne Protokoll nichts toll.«

Sternenflotten-Protokoll Protokolle dienen der geregelten Kommunikation zwischen zwei Gesprächspartnern. Wenn wir uns zum Beispiel mit einem Freund unterhalten, dann nutzen wir das Protokoll »deutsche Sprache«. Jeder wird sicherlich noch ein paar Grammatik-Regeln im Kopf haben, die genauestens definieren, wie wir wann welches Wort sprechen oder schreiben sollen. Und genauso wird es auch in der EDV gemacht. Das Protokoll gibt genauestens an, wann wir welches Byte senden oder lesen müssen, um die darin enthaltene Nachricht so zu versenden oder zu empfangen, dass wir mit einem anderen Computer, der dasselbe Protokoll verwendet, Nachrichten austauschen können.

Wie bereits weiter oben erwähnt wurde, gibt es einen Haufen verschiedener Protokolle, aber wir wollen uns hier nur die zwei interessantesten Ausprägungen anschauen, nämlich das Internet Protocol unter Verwendung von UDP und unter Verwendung von TCP.

UDP

Die Abkürzung UDP steht für *User Datagram Protocol*. UDP ist im Gegensatz zu TCP ein nicht gesichertes Protokoll. Zwischen den beiden Kommunikationspunkten besteht keine feste Verbindung, und der Zielpunkt muss folglich jedes Mal wieder neu angegeben werden. Ob die Daten am Zielpunkt angekommen sind oder nicht, wird nicht überprüft, daher wird es als ungesichertes Protokoll bezeichnet. Aufgrund der nicht implementierten Sicherheit ist es aber ziemlich schnell und effizient. Für uns Spiele-Entwickler ist das auch der Grund, warum es interessant ist. Das bisschen Sicherheit, das wir brauchen, können wir einfach selbst implementieren. Wenn man also bei einer Nachricht wirklich wissen muss, ob sie angekommen ist, so muss man diese Sicherheitsabfrage selbst in seine Netzwerk-Engine integrieren. Damit bremst man natürlich die Effizienz von UDP ein wenig aus. Interessant ist UDP aber auf alle Fälle dort, wo es uns egal sein kann, wenn mal die eine oder andere Nachricht verloren geht.

Express-Lieferung

TCP/IP

Die Abkürzung TCP bedeutet *Transmission Control Protocol*. TCP ist ein verbindungsorientiertes Protokoll. Somit können wir erst kommunizieren, wenn eine gesicherte Verbindung zu dem Zielpunkt aufgebaut ist. Sie muss nach der Kommunikation wieder explizit beendet werden. Folglich ist diese Verbindung persistent, bis sie wirklich geschlossen wird. Die Nachrichten werden gesichert übertragen. Das heißt, wenn ein Paket innerhalb einer definierten Zeit nicht am Zielpunkt angekommen ist, dann wird dieses Paket automatisch nochmals gesendet. Deshalb spricht man hier auch von einer garantierten Auslieferung des Pakets. Sollte das ursprüngliche Paket jetzt doch wider Erwarten am Zielpunkt ankommen und somit also zwei identische Pakete vorhanden sein, so wird eines gelöscht. Auch die Reihenfolge der Pakete wird überprüft, so kann z.B. das erste Paket erst nach dem letzten Paket eintreffen; die Daten werden dann geordnet. Wir können mit TCP also sicher sein, dass die Nachricht auch wirklich am Ziel ankommt. Das kostet natürlich Übertragungskapazität.

*Garantierte
Lieferung*

APIs

Damit wir die sieben Schichten des OSI-Modells nicht selbst programmieren bzw. implementieren müssen, obwohl das bestimmt viel Spaß bereiten würde, nutzen wir Netzwerk-APIs. Im Folgenden habe ich ein paar bekannte APIs aufgeführt, um euch einen kleinen Überblick darüber zu geben, was so auf dem Markt ist und genutzt werden kann.

Berkeley Sockets

Good old Berkeley Sockets... In dieser API wurde erstmals TCP/IP implementiert. 1986 wurde es in verschiedene Betriebssysteme, insbesondere in

Unix-Derivate, implementiert. Es diente 1991 in der Version 4.3 den Entwicklern der WinSock-API als Grundlage, und diese ist daher auch zu den Berkeley Sockets kompatibel. Auf Unix- und Linux-Maschinen sind die Berkeley Sockets immer noch der Standard für die Implementierung von Netzwerk-Applikationen.

Windows Sockets (WinSock)

WinSock ist die Netzwerk-API der Windows-Betriebssysteme und dort integraler Bestandteil mit der WinAPI. Aktuell ist hier die Version 2.0. Die Windows Sockets implementieren auch die Standards, nach denen sich die Berkeley Sockets richten – wenigstens mehr oder weniger. Ein entsprechend geschriebener Netzwerk-Code kann also unter Verwendung der Berkeley Sockets-API auch zügig nach Unix/Linux portiert werden. Aber die WinSock-API stellt dem Programmierer noch einige schöne zusätzliche Features zur Verfügung, die nicht über Berkeley Sockets umsetzbar sind. Als Stichwort sei hier nur der asynchrone Modus genannt, mit dem man Nachrichten direkt in die Nachrichten-Prozedur einer Windows-Applikation schieben kann. WinSock ist auch die API, mit der wir unsere ZFXNetwork-DLL erstellen werden. Die WinSock-API ist sehr effizient und auch einfach zu bedienen. Nur muss man wie bei allen Low-Level-APIs noch eine ganze Menge an eigenen Funktionen zur Kapselung programmieren, bis man sauber über das Netzwerk kommunizieren kann.

DirectPlay

Sehr funktional ...

Als Bestandteil von DirectX ist DirectPlay die von Microsoft entwickelte Spiele-Netzwerk-API. Sie ist im DirectX-Paket enthalten und stellt etliche Highlevel-Funktionalitäten wie beispielsweise eine Lobby zum Login in laufende Spiele, einen Message-Router und unterschiedliche Connectoren, zur Verfügung. DirectPlay ist eine sehr umfangreiche API und bietet uns wesentlich mehr Funktionalität, als wir in diesem Buch brauchen. Der größte Nachteil von DirectPlay, den man auch aus etlichen Gamasutra Post Mortem herauslesen kann, ist, dass man sich damit automatisch an DirectX-Plattformen bindet. Da man Server aber gern aus Stabilitätsgründen auf einer Unix- oder Linux-Maschine ohne grafisches Interface betreibt, sollte man – selbst wenn man die Clients zu einem Spiel mit Direct3D nur für Windows programmiert – den Netzwerk-Code auch unter Unix bzw. Linux zum Laufen bekommen können.

... but tied to windows

Wir werden hier die WinSock-API verwenden. Damit sind wir relativ dicht an der Berkeley-Spezifikation, und es sollte uns nicht allzu schwerfallen, unseren Netzwerk-Code nach Unix bzw. Linux zu portieren. Wenn man erst einmal auf der Ebene der Sockets gearbeitet hat, dann wird es einem nicht schwerfallen, sich in die Spezifikationen einzulesen. Die meisten Methoden von WinSock sind schließlich nach derselben Spezifikation umgesetzt.

11.4 Implementierung der Netzwerk-Bibliothek

Mittlerweile haben wir genug zum Thema Netzwerke und Netzwerkspiele gehört, um endlich selbst mit der Implementierung einer entsprechenden Bibliothek beginnen zu können. Diese führen wir auf Basis der Windows Sockets-API durch, um möglichst nah an der Hardware zu operieren. Letzten Endes ist die Netzwerkfunktionalität nicht viel mehr als das Versenden und das Empfangen von Daten. Und das lässt sich relativ einfach und schnell im Quellcode umsetzen. Natürlich verwenden wir auch hier wieder unser ...

Altbekanntes Interface-Design

Hier brauche ich nun eigentlich gar nichts mehr zu erklären, denn diesen Aufbau haben wir schon dreimal identisch für das Renderer-Device, das Input-Device und das Audio-Device besprochen. Unsere Netzwerk-Funktionalität stecken wir auch in eine DLL, mit der wir über ein Interface kommunizieren. So kann man die Implementierung beispielsweise durch eine DirectPlay-Komponente ersetzen. Wir erstellen im folgenden Abschnitt die DLL ZFXWS (für Windows Sockets), die das Interface ZFXNetworkDevice implementiert.

Die Dateien für ZFXNetwork (LIB):

- ➔ ZFXNetwork.h und ZFXNetwork.cpp
- ➔ ZFXNetworkDevice.h
- ➔ ZFX.h
- ➔ ZFX3D.h

Wie gehabt stehen die ersten beiden Dateien für die Implementierung der statischen Bibliothek bereit. Der Header ZFXNetworkDevice.h repräsentiert natürlich auch wieder das Interface. Erneut mit von der Partie ist der Allround-Header ZFX.h unserer Engine, ebenso wie der Header ZFX3D.h unserer 3D-Mathe-Bibliothek. Die Implementierung der statischen Bibliothek zeige ich auch hier nicht, denn an ihr ändern sich lediglich ein paar Variablennamen. Den kompletten Quellcode findet ihr auf der CD-ROM zu diesem Buch.

Dateien für ZFXWS (DLL):

- ➔ ZFXWS.def
- ➔ ZFXWS.h und ZFXWS.cpp
- ➔ ZFXSocketObject.h und ZFXSocketObject.cpp
- ➔ ZFXQueue.h und ZFXQueue.cpp

FIFO Neben den Dateien für die Implementierung des Interfaces haben wir hier noch zwei andere Dinge: Zum einen haben wir auch eine Klasse für ein Sockets-Objekt und zum anderen die so genannte *ZFXQueue*. Dabei handelt es sich um eine *FIFO-Schlange*, wobei FIFO für *First In First Out* steht. Das ist eine Datenstruktur analog einer verketteten Liste, bei der man jedoch neue Listenelemente immer nur am Ende anhängen und nicht mittendrin einfügen kann. Möchte man ein Element aus der Liste entnehmen, so kann man nur das jeweils erste Element entfernen. Dadurch ist sichergestellt, dass die Elemente der Liste genau in der Reihenfolge entnommen werden, wie sie dort eingefügt wurden. Den Sinn dieser Schlange werden wir später ausführlich besprechen.

Server versus Clients

Zwei in eins Das Prinzip eines Client-Server-Aufbaus in einem Netzwerk kennen wir schon von der Theorie her. Auf der Applikationsseite sieht es so aus, dass man für gewöhnlich getrennte Objekte für einen Client und einen Server implementiert. Wir werden jedoch beide Objekte sauber in einer Klasse zusammenfassen, da die Basisfunktionalität bei beiden mehr oder weniger identisch ist. Eine Applikation, die unsere Netzwerk-DLL nachher verwendet, kann sich dann zwei Instanzen der *ZFXNetwork*-Klasse ziehen und eine davon als Server und die andere als Client initialisieren und betreiben.

Dies ist aber nur nötig, wenn eine Applikation (oder eine Instanz einer Applikation) sowohl Client als auch Server sein soll. Wenn jemand beispielsweise eine Deathmatch-Session als Server hosten möchte, aber gleichzeitig auch an dem Spiel teilnehmen möchte, dann ist dies nötig. Alternativ könnte man das Programm auch auf einem Rechner nur als Server starten. Andere Spieler könnten sich dann dort einklinken und über diesen Server ihr Spiel spielen. Der Server selbst jedoch ermöglicht es nicht, im Spiel mitzumischen. Das macht dann Sinn, wenn man einen Stand-Alone-Server hat der seine ganze Rechenpower auf das *Serven* (also das Ausrichten des Spiels) konzentrieren soll und nicht auf das Rendern von Grafik.

Wie wir später im Beispiel sehen werden, kann man unsere DLL auch dazu nutzen, aus einer einzigen Applikation heraus einen Server und einen Client (oder beliebig viele) zu starten.

Pakete schnüren

Bei der Implementierung der Netzwerk-Funktionalität tun wir eigentlich nichts anderes, als die Möglichkeit zu schaffen, Datenpakete über ein Netzwerk zu versenden und zu empfangen. Das vernünftige Design der versendeten Pakete ist dabei von elementarer Wichtigkeit, denn der gesamte Netzwerk-Traffic – und damit die Geschwindigkeit des Spiels im Netzwerk – hängt linear mit der Menge der Daten zusammen, die verschickt werden.

Idealerweise sollten die versendeten Pakete daher so klein wie möglich sein. Diese These lässt sich jedoch so nicht pauschal halten. Denn wenn man viele Daten zu versenden hat, kann es auch effektiver sein, ein großes Paket zu schnüren als viele kleine zu verwenden. Daher sollten wir unser Paket so flexibel wie möglich halten. Unsere Netzwerk-Engine basiert auf der Verwendung der ZFXPACKAGE-Struktur.

```
typedef struct ZFXPACKAGE_TYPE {
    UINT    nLength;    // Länge von pData in Bytes
    UCHAR   nType;      // Nachrichten-Typ-ID
    UINT    nSender;    // Absender-ID (0 heißt Server)
    void *pData;        // Eigentliche Daten
} ZFXPACKAGE;
```

Paketgröße



Diese Struktur beginnt mit ein paar Feldern, die in jeder Art von Paket enthalten sein sollten. Am wichtigsten ist die Angabe der Größe des Pakets in nLength. Hier wird jedoch nur die Größe des Feldes pData angegeben, da die Größe der Struktur an sich als bekannt vorausgesetzt werden kann. Dazu enthält das Paket noch eine ID für den Typ der Nachricht. Hier kann jede Applikation eigene IDs vergeben, wobei gewisse Werte reserviert sind, da sie von unserer DLL verwendet werden. Tabelle 11.1 zeigt die Bedeutung der reservierten Werte. Im Feld nSender muss jeder Client seine ID eintragen, die ihm der Server bei seiner Anmeldung zugesandt hat. So weiß der Empfänger ankommender Nachrichten immer, woher das Paket kam. Der Server selbst hat die ID 0.

Felder der Struktur

Paket-ID	Bedeutung
0	Erste Nachricht vom Server an einen neuen Client. Das Feld pData enthält die ID, die der Server dem Client gegeben hat.
1	Nachricht vom Server an alle Clients, dass sich ein neuer Client angemeldet hat. Das Feld pData enthält in diesem Fall die ID, die der Server dem Client zugewiesen hat.
2	Nachricht vom Server an alle Clients, dass sich ein Client abgemeldet hat. Das Feld pData enthält die ID des abgemeldeten Clients.

Tabelle 11.1:
Reservierte Werte
für Paket-IDs

Im Feld pData stehen schließlich die eigentlichen Daten des Pakets. Hier alloziert man Speicher in der Menge nLength und füllt dort beliebige eigene Daten ein. Beispielsweise kann man für jeden Nachrichtentyp, den man definiert, eine entsprechende Struktur erstellen. Wenn in einer Applikation zum Beispiel ein Spieler eine Rakete abfeuert, so müsste man wenigstens die Startposition, den Geschwindigkeitsvektor und den Schadenswert der Rakete als Daten in ein Paket füllen und an den Server schicken.

Daten

Die Struktur `ZFXPACKAGE` ist damit sozusagen nur eine *Carrier-Nachricht*, die als Träger für die eigentliche Nachricht dient.

Warteschlangen

Nichts als Nachrichten

Damit sind wir schon mittendrin im Netzwerk-Geschäft. Rufen wir uns noch einmal in Erinnerung, dass im Netzwerk eigentlich nur eines passiert: das Versenden von Datenpaketen. Wenn ein Client etwas zu melden hat, dann sendet er eine Nachricht an den Server. Dieser empfängt die Nachricht und bearbeitet sie entsprechend. Wenn der Server etwas zu melden hat, dann sendet er seinerseits eine Nachricht an einen, mehrere oder alle Clients. Jeder Client muss also auch bereit sein, Nachrichten zu empfangen und zu bearbeiten. Das impliziert bereits, dass sowohl der Client als auch der Server eine Art Verwaltung für anfallende Nachrichten haben müssen. Aus diesem Grunde führen wir hier Warteschlangen für Nachrichten ein.

Abstraktions- ebene

Nun müssen wir uns noch darüber klar werden, auf welcher Abstraktionsebene sich unsere DLL später befinden wird. Wenn wir hier von Client und Server reden, so ist damit nur die rein technische Seite der Programmierung gemeint. Auch die Applikation, die später die DLL verwendet, um einen Client oder Server zu erzeugen, muss dort eine entsprechende Logik einbringen. Auf der Ebene der DLL ist wenig mehr möglich als das Versenden und Empfangen von Nachrichten. Wie eine Applikation aber auf empfangene Nachrichten reagiert, das bleibt ganz dem Programmierer der Applikation überlassen.

Postfach

Daher legen wir das Interface `ZFXNetworkDevice` entsprechend so an, dass eine Applikation, die ein solches Device als Client oder Server verwendet, sich dort die eingegangenen Nachrichten abholen muss. Das kann man sich wie die Windows-Nachrichten-Warteschlange vorstellen, die man in der Hauptschleife eines Windows-Programms abfragt. Immer wenn ein Client oder ein Server unserer DLL eine Nachricht empfängt, steckt er diese in sein Eingangs-Postfach. Dies ist natürlich, wer hätte es gedacht, ein Objekt vom Typ `ZFXQueue`, also eine FIFO-Warteschlange. Die Applikation ist nun dafür verantwortlich, dieses Postfach Nachricht für Nachricht zu leeren und angemessen auf die Nachrichten zu reagieren. Wenn eine Applikation jedoch über ihr `ZFXNetworkDevice`-Objekt eine Nachricht verschickt, wird diese direkt an den Server gesendet (oder vom Server entsprechend an bestimmte oder an alle Clients), ohne in einem Ausgangs-Postfach zwischengelagert zu werden. Dieses Postfach brauchen wir ja nur, weil wir keine Möglichkeit haben (bzw. keine implementieren), von der DLL direkt auf die Applikation zuzugreifen und die Nachrichten sofort beim Empfangen an die Applikation weiterzuleiten.

Hier seht ihr die Klasse `ZFXQueue`, die wir als einfaches Postfach verwenden, bzw. die Klasse `ZFXQueueElem`, deren Instanzen als Elemente in der Warteschlange verwendet werden:

```
class ZFXQueueElem {
public:
    ZFXQueueElem(const char *pData,
                  unsigned int nSize);
    ~ZFXQueueElem(void);

    ZFXQueueElem *m_pNext;
    char          *m_pData;
    unsigned int  m_nSize;
};

class ZFXQueue {
private:
    ZFXQueueElem *m_pHead;
    ZFXQueueElem *m_pTail;
    unsigned int  m_Count;

public:
    ZFXQueue(void);
    ~ZFXQueue(void);

    void Dequeue(void);
    void Enqueue(const void *pData,
                 unsigned int nSize);
    void Front(void *pData, bool bDequeue);

    unsigned int GetCount(void) { return m_Count; }

    unsigned int GetFrontSize(void)
    {
        if (m_pHead) return m_pHead->m_nSize;
        else return 0;
    }
};
```



Ein Element der Warteschlange hat lediglich einen Pointer auf die Daten sowie ein Attribut, das die Größe der Daten in Bytes angibt. Diese beiden Attribute haben eine ähnliche Funktion wie die Felder `nLength` und `pData` der Struktur `ZFXPACKAGE`. Wann immer also ein ankommendes Paket in die Warteschlange geschoben wird, wird das gesamte Paket in das Element kopiert und die Größe des Pakets gespeichert. Der Zeiger `m_pNext` eines Elements in der Schlange zeigt dann auf das nachstehende Element in der Schlange, sofern es eines gibt. Dieses Konzept kennen wir bereits von verketteten Listen.

*Elemente der
Warteschlange*

Methoden der Warteschlange

Die Warteschlange selbst ist recht einfach gehalten. Die Funktion `ZFXQueue::Enqueue` dient dazu, ein neues Element an das Ende der Warteschlange anzufügen.



```
void ZFXQueue::Enqueue(const void *pData,
                      unsigned int nSize) {
    ZFXQueueElem *pNew = new ZFXQueueElem(
        (const char*)pData, nSize);

    // noch kein Element vorhanden
    if (m_Count == 0) {
        m_pHead = pNew;
        m_pTail = pNew;
    }
    // bereits ein Element vorhanden
    else if (m_Count == 1) {
        m_pHead->m_pNext = pNew;
        m_pTail = pNew;
    }
    else {
        m_pTail->m_pNext = pNew;
        m_pTail = pNew;
    }
    m_Count++;
} // Enqueue
```

Genau das Gegenteil bewirkt die Funktion `ZFXQueue::Dequeue`, denn man verwendet diese Funktion dazu, um das erste Element aus der Warteschlange zu löschen:



```
void ZFXQueue::Dequeue(void) {
    ZFXQueueElem *pTemp;

    // ist bereits leer
    if (m_Count == 0) return;

    // hat genau noch ein Element
    else if (m_Count == 1) {
        delete m_pHead;
        m_pHead = NULL;
        m_pTail = NULL;
    }
    else {
        pTemp = m_pHead;
        m_pHead = m_pHead->m_pNext;
        delete pTemp;
    }
    m_Count--;
} // Dequeue
```


Bevor man ein Element aus der Schlange löscht, sollte man sich aber eventuell eine Kopie der Daten besorgen. Schließlich hebt man in der Warteschlange Pakete auf, um sie später zu bearbeiten. Die Funktion `ZFXQueue::Front` gibt dem Aufrufer eine Kopie des ersten Elements aus der Schlange und löscht dieses dann auf Wunsch auch gleich.

*Erstes Element
auslesen*

```
void ZFXQueue::Front(void *pData, bool bDequeue) {
    if (pData) {
        if (m_pHead) {
            memcpy(pData,
                m_pHead->m_pData,
                m_pHead->m_nSize);
        }
    }
    if (bDequeue) Dequeue();
} // Front
```



Aber Vorsicht: Wie ihr hier seht, erwartet die Funktion, dass man hinter der Adresse, auf die der Parameter `pData` zeigt, bereits mindestens die korrekte Menge an Speicher bereitgestellt hat. Damit dies möglich ist, gibt es die Funktion `ZFXQueue::GetFrontSize`, die den benötigten Wert auf Wunsch vorab liefern kann.

Das waren auch schon alle Methoden, die wir für eine einfache Warteschlange benötigen. Nun müssen wir uns lediglich noch die Klasse `ZFXQueueElem` anschauen. Diese besteht nur aus einem Konstruktor und einem Destruktor.

```
ZFXQueueElem::ZFXQueueElem(const char *pData,
    unsigned int nSize) {
    m_pData = NULL;
    m_pNext = NULL;
    m_pData = new char[nSize];
    m_nSize = nSize;
    memcpy(m_pData, pData, nSize);
} // constructor

/*-----*/

ZFXQueueElem::~ZFXQueueElem(void) {
    if (m_pData) {
        delete [] m_pData;
        m_pData = NULL;
    }
    m_pNext = NULL;
} // destructor
```



Wenn wir ein neues Element für die Liste erzeugen, allokatieren wir also einfach den entsprechenden Speicher und kopieren die Daten dorthin. Beim Löschen eines Elementes wird der Speicher dann wieder ordentlich aufgeräumt.

Fertiges Postfach

Mit diesen beiden Klassen haben wir nun ein Postfach in unserem Repertoire, in dem wir später unsere über das Netzwerk ankommenden Pakete ablegen werden, bis die Applikation sie abfragt. Jetzt kümmern wir uns um die so genannten *Socket-Objekte*, über die wir Daten senden um empfangen können.

Socket-Objekte

Client- und Server-Socket- Objekte

Die Basis einer jeden Kommunikation über das Netzwerk ist bei der Windows Sockets-API ein so genanntes Socket-Objekt. Diese Objekte kann man sich als Abschlusstücke einer imaginären Netzwerkleitung zwischen Server und Client vorstellen. Jeder Client verfügt über ein solches Socket-Objekt. Über dieses Socket-Objekt sendet er seine Daten an den Server, und über dieses Objekt empfängt er seine Daten vom Server. Der Server hingegen hat mehrere Socket-Objekte. Er benötigt ein so genanntes *Master-Socket-Objekt*, über das er sozusagen das Netzwerk überwacht. Immer dann, wenn sich ein neuer Client in das Netzwerk einklinkt, *hört* der Server das auf seinem Master-Socket. Zusätzlich hat der Server noch ein Socket-Objekt für jeden Client, der im Netzwerk bei ihm angemeldet ist. Wenn der Server Daten an einen Client senden will oder von diesem empfängt, geschieht das auf genau dem Socket, der sozusagen von dem Server zu dem entsprechenden Client führt.

Anmeldung

Das bedeutet: Wann immer sich ein Client bei einem Server anmeldet und von diesem akzeptiert wird, passieren zwei Dinge: Zum einen öffnet der Server einen neuen Socket, über den er exklusiv mit diesem Client kommuniziert, und zum anderen öffnet der Client ebenfalls einen Socket, der von ihm aus gesehen zu dem Socket führt, über den der Server mit ihm kommunizieren wird. Der Server hat damit also zu jedem einzelnen Client einen eigenen heißen Draht.

WinSock-API

Was genau so ein Socket ist und wie der Server einen entsprechenden Socket erkennt, das sind technische Fragen, die durch die WinSock-API behandelt werden und uns hier nicht weiter kümmern sollen. Im Folgenden entwickeln wir eine Klasse, die die wichtigsten Funktionen der WinSock-API kapselt, die eben jene Socket-Objekte betreffen. Damit ihr an dieser Stelle eine bessere Übersicht habt, welche Schritte jeweils notwendig sind, um einen Client oder einen Server zu erzeugen, schicke ich in Tabelle 11.2 eine kleine Aufstellung der notwendigen Aufrufe vorweg. Die zugehörigen Methoden entwickeln wir im Anschluss.

Funktion	Aufgabe	Server	Client
CreateSocket()	Initialisierung des Sockets	x	x
Bind()	Benennung des Server-Master-Sockets	x	
Listen()	Server horcht, ob Clients anklopfen	x	
Accept()	Server akzeptiert anklopfenden Client	x	
Connect()	Client klopft beim Server an		x
Send()	Daten über das Netzwerk senden	x	x
Receive()	Daten über das Netzwerk empfangen	x	x
Disconnect()	Socket freigeben	x	x

Tabelle 11.2:
Funktionen der
ZFXSocketObject-Klasse

ZFXSocketObject-Klasse

Die folgende Klasse dient primär dazu, die Arbeit mit dem Socket-Objekt der WinSock-API zu vereinfachen und gewisse gleich bleibende Arbeitsabläufe zu kapseln. Der Kern dieser Klasse ist damit das Objekt vom Typ SOCKET, das in der WinSock-API definiert ist und einen Socket repräsentiert. Diese Klasse dient uns später dazu, Nachrichten über Sockets zu versenden und zu empfangen, unabhängig davon, ob es sich um einen Server- oder einen Client-Socket handelt. Die Funktionen für Server und Client unterscheiden sich lediglich in der Initialisierungsphase voneinander; das Senden und Empfangen der Daten verläuft identisch.

SOCKET

```
class ZFXSocketObject {
public:
    ZFXSocketObject(FILE *pLog);
    ~ZFXSocketObject(void);

    bool    IsRunning(void) { return m_bRunning; }

    // Socket-MISC-Funktionen
    HRESULT CreateSocket(void);
    HRESULT Bind(int nPort);
    HRESULT Listen(void);
    HRESULT Accept(SOCKET *skToNewClient);
    HRESULT Connect(char *chServer, int nPort);
    void    Disconnect(void);

    // Daten senden und empfangen
    int     Send(const char*,UINT);
    int     Send(const char*,UINT,SOCKET);
    HRESULT Receive(SOCKET sk);
};
```



```

// Rein informativ
SOCKET GetSocket(void) { return m_skSocket; }

// Informationen der Inbox
bool IsPkgWaiting(void)
    { return (m_pInbox->GetCount() > 0); }
UINT GetNextPkgSize(void)
    { return (m_pInbox->GetFrontSize()); }
HRESULT GetNextPkg(ZFXPACKAGE*);

private:
    ZFXQueue *m_pInbox;
    SOCKET m_skSocket;
    char *m_Buffer;
    bool m_bRunning;
}; // class

```

MSIC Diese Klasse stellt eine Menge an Funktionen zur Verfügung, mit denen wir das SOCKET-Objekt komfortabel bedienen können. Diese Funktionen werden wir gleich der Reihe nach durchgehen, aber hier schon mal ein kurzer Überblick: Die unter dem Kommentar MISC gelisteten Funktionen dienen lediglich zur Initialisierung und zum Herunterfahren eines Sockets. Dazu verwenden Server und Clients verschiedene der hier deklarierten Funktionen, wie wir später sehen werden.

Senden und empfangen Als Nächstes hält die Klasse Funktionen bereit, mit denen man Daten aus einem Pointer heraus verschicken kann, sowie eine Funktion, die Daten empfängt. Diese ist parameterlos, da die empfangenen Daten in die Inbox, also die FIFO-Warteschlange, geschoben werden.

Inbox Die Inbox hat schließlich zwei Funktionen, über die man von außen den Status abfragen kann, sowie eine Funktion, über die man ein Päckchen aus der Inbox abholen kann. Aber beginnen wir am besten ganz am Anfang, nämlich mit dem Konstruktor und dem Destruktor:



```

ZFXSocketObject::ZFXSocketObject(FILE *pLog) {
    m_skSocket = INVALID_SOCKET;
    m_bRunning = false;
    m_pInbox = NULL;
    m_Buffer = NULL;
} // constructor

/*-----*/

ZFXSocketObject::~ZFXSocketObject(void) {
    if (IsRunning()) {
        Disconnect();
        m_bRunning = false;
    }
    if (m_pInbox) {

```

```

delete m_pInbox;
m_pInbox = NULL;
}
if (m_Buffer) {
delete [] m_Buffer;
m_Buffer = NULL;
}
m_skSocket = INVALID_SOCKET;
} // destructor

```

Aufgrund der besseren Lesbarkeit der Quelltexte habe ich für den Abdruck im Buch auch an dieser Stelle wieder die Fehlerbehandlung in den Funktionen ein wenig beschnitten. Wie ihr auf der CD-ROM sehen könnt, verfügt jede DLL über ihre eigene Log-Datei, in die entsprechende Fehlermeldungen ausgegeben werden.



Erzeugung eines Sockets

Bevor ein Socket der WinSock-API von uns verwendet werden kann, müssen wir diesen erst einmal erzeugen, unabhängig davon, ob er ein Master-Socket für den Server oder ein Socket für einen Client werden soll.¹ Dazu benötigen wir die WinSock-Funktion `socket()`:

WinSock

```
SOCKET socket(int af, int type, int protocol);
```

Der erste Parameter gibt die Adressen Familie für das Netzwerk an. Hier stehen einige zur Auswahl, beispielsweise `AF_INET` (für TCP/IP), `AF_IPX` (für IPX) oder `AF_APPLETALK` (für Apple Talk). Der zweite Parameter legt den Typ fest. Für den Typ stehen `SOCK_STREAM` und `SOCK_DGRAM` zur Auswahl. `SOCK_STREAM` steht für das TCP-Protokoll und `SOCK_DGRAM` für UDP. Da man durch die Auswahl des zweiten Parameters das Protokoll schon vorgibt, kann man den dritten Parameter einfach ignorieren. Es gibt noch eine dritte Art von Socket-Typen, für die der dritte Parameter von Bedeutung ist, aber diese sind heutzutage unbedeutend geworden.

Der Rückgabewert der Funktion ist dann ein gültiges Socket-Objekt (oder der Fehler `INVALID_SOCKET`), was letzten Endes nicht viel mehr als eine bestimmte ID ist, durch die die WinSock-API den Socket wiedererkennen kann. Und so sieht unsere Kapselung aus:

```

HRESULT ZFXSocketObject::CreateSocket(void) {
    if (m_skSocket != INVALID_SOCKET) Disconnect();

    m_skSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (m_skSocket==INVALID_SOCKET) {

```



¹ Die zusätzlichen Sockets die ein Server für jeden sich anmeldenden Client benötigt werden jedoch anders erzeugt.

```

        return ZFX_FAIL;
    }
    m_pInbox = new ZFXQueue();
    m_Buffer = new char[65536];
    memset(m_Buffer,0,65536);
    return ZFX_OK;
}

```

TCP/IP Für den Fall, dass der Socket bereits geöffnet wurde, schließen wir ihn zunächst über die Funktion `ZFXSocketObject::Disconnect` wieder und geben die entsprechenden Ressourcen frei. Dann erzeugen wir das Socket-Objekt. Man beachte, dass wir das TCP/IP-Protokoll verwenden. Anschließend legen wir die Inbox für das Objekt an, ebenso wie einen 8 Kbyte großen Speicherbereich. Letzterer dient später dazu, über den Socket empfangene Nachrichten zwischenspeichern, ohne zur Laufzeit Speicher allokalieren zu müssen.

Namensgebung

bind() Einen Socket kann man auch benennen, was in der NetzwerkTerminologie *bind* genannt wird. Dabei wird der Socket mit einem bestimmten Port verbunden, also einer Öffnung des Computers in das Netzwerk. Dies ist nur für den Master-Socket des Servers notwendig. Für einen Client ist dies zwar auch möglich, aber vollkommen überflüssig. Die Benennung erfolgt über die WinSock-Funktion `bind()`:

```

int bind(SOCKET s, const struct sockaddr FAR* name,
        int namelen);

```

sockaddr Der erste Parameter der Funktion ist die ID eines unbenannten Socket-Objekts. Der zweite Parameter ist eine Struktur, die die notwendigen Angaben enthält, und der dritte Parameter gibt die Größe der Struktur an. Diese mysteriöse Struktur `sockaddr` wird uns noch öfter begegnen, allerdings auch in Form der Struktur `sockaddr_in`. In der Version 2.0 der WinSock-API wird die Struktur `sockaddr_in` ebenfalls von der Funktion akzeptiert und ist für das TCP-Protokoll zu verwenden.



```

struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct    in_addr sin_addr;
    char     sin_zero[8];
};

```

Das erste Feld der Struktur muss hier zwangsläufig immer `AF_INET` sein, da wir TCP verwenden. Im zweiten Feld speichern wir logischerweise den Port, und das dritte Feld ist eine Struktur für die IP-Adresse des Servers. Hier können wir aber auch `INADDR_ANY` angeben, womit die WinSock-API selbst

die geeignete Adresse herausucht. Das letzte Feld dient nur dem Zweck, die Struktur auf dieselbe Größe zu bringen wie die `sockaddr`-Struktur.

Unsere Kapselung der `bind()`-Funktion sieht wie folgt aus:

```
HRESULT ZFXSocketObject::Bind(int nPort) {
    sockaddr_in saServerAddress;

    memset(&saServerAddress, 0, sizeof(sockaddr_in));
    saServerAddress.sin_family = AF_INET;
    saServerAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    saServerAddress.sin_port = htons(nPort);

    if ( bind(m_skSocket, (sockaddr*)&saServerAddress,
              sizeof(sockaddr)) == SOCKET_ERROR) {
        Disconnect();
        return ZFX_FAIL;
    }
    return ZFX_OK;
} // Bind
```



Was genau es mit den Funktion `htons()` und `htonl()` auf sich hat, erkläre ich in dem Exkurs *Byte-Reihenfolge beim Host und dem Netzwerk*. Ansonsten findet in dieser Funktion eigentlich nicht viel mehr statt als die Weiterleitung des Aufrufs an die entsprechende WinSock-Funktion.

htons() / htonl()

Byte-Reihenfolge beim Host und dem Netzwerk

Für den Entwurf eines Computers bzw. einer CPU gibt es zwei verschiedene Ansätze zur Speicherung der Daten, die man als Big Endian und Little Endian bezeichnet. Bei einem Intel-Prozessor wird beispielsweise mit Little Endian gearbeitet, wogegen Motorola-Chips die Big-Endian-Methode verwenden.

Der Unterschied der beiden Methoden bezieht sich darauf, ob die Daten beginnend mit dem Most Significant Bit (MSB) oder dem Least Significant Bit (LSB) gespeichert werden. Bei der Little-Endian-Methode speichert man einen Wert beginnend mit dem kleinen, also dem rechten Ende. Der Wert 0x1234 würde im Speicher entsprechend als [0x34 0x12] gespeichert. Bei der Big-Endian-Methode läuft es genau anders herum, und der Wert 0x1234 würde im Speicher als [0x12 0x34] erscheinen.

In unserer Netzwerkumgebung unterscheidet man zwischen der Reihenfolge, die der Host (der Computer) verwendet, und der Reihenfolge des Netzwerks. In der WinSock-API verwendet das Netzwerk Big Endian als

Network Byte Order. In unserem Programm (für Intel-kompatible Prozessoren) verwenden wir aber Little Endian als Host Byte Order. Bei der Übergabe von Adressen und Ports an die Netzwerk-API müssen wir die Werte also entsprechend konvertieren. Dazu bietet die WinSock-API die folgenden Funktionen:

```
// host to network (short = 16 Bit)
u_short htons(u_short hostshort);

// network to host (short = 16 Bit)
u_short ntohs(u_short netshort);

// host to network (long = 32 Bit)
u_long htonl(u_long hostlong);

// network to host (long = 32 Bit)
u_long ntohl(u_long netlong);
```

Für uns ist es dabei primär wichtig, die Hostwerte für die `sockaddr_in`-Struktur in Network Byte Order zu konvertieren, damit die API diese richtig interpretieren kann. Entsprechend verwenden wir die Funktionen `htons()` und `htonl()`, je nach der Länge der Werte.

Reden ist Silber, Zuhören ist Gold

Horchen auf neue Clients

Nachdem man für den Server einen Socket erzeugt und diesen benannt hat, ist er betriebsbereit für den Netzwerkverkehr. Der Master-Socket des Servers dient aber nicht dazu, Daten zu versenden oder zu empfangen jedenfalls nicht die applikationsrelevanten Daten. Der Master-Socket hat eigentlich nur eine Aufgabe: Er soll den Server informieren, wenn sich ein neuer Client in das Netzwerk einhängen möchte. Wenn ein Client ins Netzwerk möchte, teilt er dies dem Server über die `connect()`-Funktion mit. Damit der Server diese Aufrufe aber auch mitbekommt, muss er über seinen Master-Socket auf solche Konnektierungsanfragen lauschen. Dazu rufen wir nach der Benennung des Master-Sockets die WinSock-Funktion `listen()` auf:

```
int listen(SOCKET s, int backlog);
```

Der erste Parameter der Funktion gibt natürlich den Socket an, der auf Horchposten gelegt wird. Dies ist in einer Client-Server-Architektur natürlich immer der Master-Socket des Servers. Der zweite Parameter gibt die Anzahl an Clients an, die maximal gleichzeitig einen Konnektierungsversuch starten können. Entsprechend schlank ist unsere Kapselung dieser Funktion:


```
HRESULT ZFXSocketObject::Listen(void) {
    if (listen(m_skSocket, 32) != 0)
        return ZFX_FAIL;

    m_bRunning = true;
    return ZFX_OK;
} // Listen
```



Akzeptieren von Clients

Nachdem ein Server-Master-Socket beim Horchen festgestellt hat, dass ein neuer Client in das Netzwerk möchte, muss er diesen akzeptieren, um das zu ermöglichen. Weiter oben hatte ich gesagt, dass ein Server genau ein separates Socket-Objekt für jeden einzelnen Client im Netzwerk hat. Immer dann, wenn der Server einen neuen Client akzeptiert, wird ein neues Socket-Objekt für den Server erzeugt, über das er exklusiv mit genau diesem neuen Client kommunizieren kann. Dieses Socket-Objekt muss der Server also in seiner Liste mit Socket-Objekten speichern, da es quasi die ID zu einem bestimmten Client ist. Um einen anfragenden Client zu akzeptieren, verwendet man die WinSock-Funktion `accept()`:

```
SOCKET accept (SOCKET s, struct sockaddr FAR* addr,
               int FAR* addrlen);
```

Der erste Parameter dieser Funktion ist der Socket, über den der neue Client akzeptiert werden soll. Dabei muss es sich um den Socket handeln, der auf neu ankommende Clients lauscht, was natürlich wiederum der Master-Socket des Servers ist. Der zweite Parameter ist eine optional anzugebende Struktur, in der die Adressdaten des neuen Clients abgespeichert werden, und der dritte Parameter verlangt nach der Größe in Bytes der Struktur im zweiten Parameter.

Als Rückgabewert der Funktion erhalten wir bei Erfolg genau das neue Socket-Objekt, über das der Server den neuen Client von nun an ansprechen kann. Die Kapselung in unserer Klasse sieht wie folgt aus:

```
HRESULT ZFXSocketObject::Accept(SOCKET *skToNewClient) {
    sockaddr_in saClientAddress;
    int         nClientSize = sizeof(sockaddr_in);

    (*skToNewClient) = accept(m_skSocket,
                              (sockaddr*)
                              &saClientAddress,
                              &nClientSize);
    if ((*skToNewClient) == INVALID_SOCKET)
        return ZFX_FAIL;
    return ZFX_OK;
} // accept
```



Der Server unserer DLL muss also diese Funktion aufrufen und ihr eine Adresse angeben, an der die ID des neuen Sockets hinterlegt werden soll. Dies ist typischerweise ein freier Platz in der Liste aller Clients, die der Server pflegt.

Konnektieren zum Server

So langsam fühlt sich der Client aber ein wenig zurückgesetzt, weil wir dauernd nur vom Server reden. Also befassen wir uns nun mit der einzigen Funktion, die ein Client nach seiner Socket-Initialisierung aufzurufen hat. Dies ist die Konnektierungsanfrage an einen Server mittels der WinSock Funktion `connect()`:

```
int connect (SOCKET s,
             const struct sockaddr FAR* name,
             int namelen);
```

Der erste Parameter ist hier ein noch unkonnectiertes Socket-Objekt, also der Socket des Clients, der ins Netzwerk möchte. Der zweite Parameter ist der Name des Servers, zu dem konnektiert werden soll, und der dritte Parameter gibt die Größe des zweiten Parameters (in Bytes) an.

*IP-Adresse und
Port-Nummer*

Um sich bei einem Server anmelden zu können, muss der Client natürlich die IP-Adresse und die Port-Nummer des Servers kennen. Dabei kann sich ein Client natürlich auch bei einem Server anmelden, der unter derselben IP-Adresse wie der Client selbst (also auf demselben Computer) läuft. Hier ist unsere Kapselung der Konnektierungs-Funktionalität:



```
HRESULT ZFXSocketObject::Connect(char *chServer,
                                  int nPort) {
    sockaddr_in saServerAddress;
    LPHOSTENT  pHost=NULL;

    // Versuche den Server zu finden
    memset(&saServerAddress,0,sizeof(sockaddr_in));
    saServerAddress.sin_port      = htons(nPort);
    saServerAddress.sin_family    = AF_INET;
    saServerAddress.sin_addr.s_addr=inet_addr(chServer);

    if (saServerAddress.sin_addr.s_addr==INADDR_NONE) {
        pHost = gethostbyname(chServer);
        if (pHost != NULL) {
            saServerAddress.sin_addr.s_addr =
                ((LPIN_ADDR)pHost->h_addr)->s_addr;
        }
    }
    else
        return ZFX_FAIL;
}
```

```
// Konnekte zu der Server-Adresse
if (connect(m_skSocket, (sockaddr*)&saServerAddress,
           sizeof(sockaddr)) == SOCKET_ERROR) {
    Disconnect();
    return ZFX_FAIL;
}
m_bRunning = true;
return ZFX_OK;
} // Connect
```

Wie ihr seht, nimmt unsere Funktion neben der Port-Nummer noch einen String auf, der den Server bezeichnen soll. Dies kann in zwei Formen erfolgen. Entweder gibt man im String direkt die IP-Adresse in Form von vier dreistelligen Nummern von 0-255 an, die durch Punkte voneinander getrennt werden, beispielsweise als String »192.168.0.128«. Mit der WinSock-Funktion `inet_addr()` kann man diesen String so umwandeln, dass er korrekt für das Feld `s_addr` der Struktur `sockaddr_in` aufgelöst wird. Alternativ kann man im String aber auch den Servernamen und nicht die IP-Adresse angeben. Dann schlägt `inet_addr()` fehl, und wir rufen die Funktion `gethostbyname()` auf. Diese löst einen Servernamen, aber keine IP-Adresse so auf, dass die als Rückgabewert erzeugte Struktur `HOSTENT` die entsprechenden Daten beinhaltet.

IP oder WWW

Diskonnektieren vom Server

Möchte sich ein Client vom Netzwerk abmelden, oder etwa der Master-Socket des Servers, so muss das Socket-Objekt erst heruntergefahren werden. Dies erledigt die WinSock-Funktion `shutdown()`. Danach rufen wir die Funktion `closesocket()` auf und können nun davon ausgehen, dass der Socket ordnungsgemäß abgemeldet worden ist. In unserem Code sieht das so aus:

```
void ZFXSocketObject::Disconnect(void) {
    if (m_skSocket != INVALID_SOCKET) {
        shutdown(m_skSocket, SD_BOTH);
        closesocket(m_skSocket);
        m_skSocket = INVALID_SOCKET;
    }
} // Disconnect
```



Der zweite Parameter der `shutdown()`-Funktion sorgt dafür, dass in diesem Fall keinerlei Daten mehr an den Socket gesendet oder von ihm empfangen werden können. Alternativ könnte man auch festlegen, dass beispielsweise noch empfangene Daten über den Socket abgerufen werden können, bevor dieser ganz geschlossen wird. Bei uns macht das aber wenig Sinn, da ein Client keine persistenten Daten speichern muss. Wenn er sich also aus dem

SD_BOTH

Netzwerk ausklinken will, dann interessiert es ihn auch nicht mehr, was der Server ihm noch zu senden hat.

Senden von Daten

Sobald ein Socket-Objekt korrekt initialisiert worden ist und eine Verbindung zwischen Server und Client besteht, können über die entsprechenden Sockets Daten versandt werden. Dies geschieht mit der WinSock-Funktion `send()`:

```
int send(SOCKET s, const char FAR * buf, int len,
        int flags);
```

Logischerweise gibt der erste Parameter den Socket an, über den die Daten gesendet werden sollen. Für einen Client ist dies natürlich das einzige Socket-Objekt, das für den Client erstellt wurde. Für den Server ist dies genau das Socket-Objekt, das bei der `accept()`-Funktion für den Client erzeugt wurde, an den nun Daten gesendet werden sollen. Der zweite Parameter ist ein `char`-Pointer auf die zu versendenden Daten, und der dritte Parameter gibt die Größe der Daten (in Bytes) an. Im letzten Parameter kann man noch Flags für die Art der Datenübertragung angeben, aber hier begnügen wir uns in der Regel mit dem Eintrag 0. Der Rückgabewert der Funktion ist entweder die Fehlermeldung `SOCKET_ERROR` oder die Anzahl der tatsächlich gesendeten Bytes.

Garantierte Lieferung

Das TCP/IP-Protokoll stellt sicher, dass unsere gesendeten Pakete auch wirklich beim Empfänger ankommen. Allerdings kann es vorkommen, dass die `send()`-Funktion es nicht schafft, alle Daten auf einmal zu senden, beispielsweise wenn der Empfangspuffer des empfangenden Sockets bereits gut gefüllt ist. Wir müssen also hier den Rückgabewert auf alle Fälle abfragen und mit der zu sendenden Datenmenge vergleichen. Wurden tatsächlich weniger Daten versandt als in Auftrag gegeben, dann müssen wir nochmals die `send()`-Funktion aufrufen und den bisher nicht gesendeten Teil der Daten verschicken. Dies machen wir folglich in einer Schleife so lange, bis wir alle Daten verschickt haben oder bis ein Fehler aufgetreten ist. Das folgende Listing zeigt unsere tatsächliche Implementierung. Wir verwenden zwei `ZFXSocketObject::Send`-Methoden, denn ein Client hat ja nur seinen eigenen Socket, den er als Attribut hat. Ein Server hingegen muss in der Parameterliste der Funktion den entsprechenden Client-Socket angeben, über den er senden will.



```
int ZFXSocketObject::Send(const char *pPkg, UINT nSize)
{
    UINT nSent=0;
    UINT n=0;

    while (nSent < nSize) {
```

```

        n = send(m_skSocket, pPkg+nSent, nSize-nSent, 0);
        if (n==SOCKET_ERROR)
            return n;
        else nSent += n;
    }
    return nSent;
} // Send
/*-----*/

int ZFXSocketObject::Send(const char *pPkg, UINT nSize,
                        SOCKET skReceiver) {
    UINT nSent=0;
    UINT n=0;

    while (nSent < nSize) {
        n = send(skReceiver, pPkg+nSent, nSize-nSent, 0);
        if (n==SOCKET_ERROR)
            return n;
        else nSent += n;
    }
    return nSent;
} // Send

```

Auch wenn sich alle Clients beim Server angemeldet haben, so führt der Server an sich beispielsweise keine automatische Liste auf dem Master-Socket, wer sich bei ihm angemeldet hat. Wenn wir den Server also eine Nachricht an alle Clients im Netzwerk schicken lassen wollen, dann müssen wir für jeden einzelnen Client-Socket, den der Server über die Funktion `accept()` erhalten hat, dieselben Daten einzeln versenden.

*Päckchen vom
Server*

Empfangen von Daten

Pfannkuchen, nichts als Pfannkuchen – jedenfalls bisher. Doch das Empfangen von Daten über einen Socket ist mal wieder ein typischer Fall von Nicht-Pfannkuchen. Bevor ich gleich ausführe, warum das so ist, schauen wir uns die WinSock-Funktion an, mit der wir Daten über einen Socket empfangen können:

*Endlich mal was
Komplexes!*

```

int recv(SOCKET s, char FAR* buf, int len,
        int flags);

```

Der erste Parameter ist das Socket-Objekt, über das Daten empfangen werden. Der zweite Parameter ist ein Zeiger auf einen Puffer, in dem die empfangenen Daten gespeichert werden sollen, und der dritte Parameter gibt an, wie viele Bytes maximal in diesen Datenpuffer passen. Im dritten Parameter können wir wieder ein paar Flags angeben; diese Option benötigen wir jedoch nicht. Der Rückgabewert ist entweder eine Fehlermeldung oder die Anzahl der tatsächlich empfangenen Bytes.

Das Problem!

Das erste und größte Problem ist, dass wir keine Ahnung haben, wie viele Daten am Socket liegen und darauf warten, von uns empfangen zu werden. Sind es ein paar Bytes, ein paar Dutzend Kilobytes oder gar ein Haufen Megabytes? Wir können also den Empfangspuffer gar nicht entsprechend dimensionieren, um alle Möglichkeiten abdecken zu können. Die offensichtlichste Lösung ist, dass wir so lange in einer Schleife die `recv()`-Funktion aufrufen, bis diese als Rückgabewert 0 liefert, was andeutet, dass 0 Bytes empfangen worden sind, also keine Daten mehr auf den Empfang gewartet haben.

In jedem Schleifendurchlauf haben wir dann eine Anzahl Bytes in unserem Puffer, die die gelesenen Daten repräsentieren. Aus diesem Puffer müssen wir also in jedem Schleifendurchlauf alle dort vorhandenen Pakete auslesen und in unsere Inbox schieben. Wir gehen nun davon aus, dass wir ausschließlich Daten vom Typ `ZFXPACKAGE` versenden. Daraus ergibt sich aber das nächste Problem: Wenn wir nicht absolutes Glück haben, dann passen eben nicht alle einzelnen Pakete exakt in die Größe des Puffers. Wir haben also nach einem Schleifendurchlauf eventuell nicht fünf, sondern fünfeinhalb Pakete empfangen. In jedem Durchlauf müssen wir daher nach dem Empfangen unseren Puffer durchlaufen und dort jedes Paket, das noch komplett im Puffer ist, in die Inbox schieben.

Treffen wir jedoch auf ein Paket, das nicht mehr komplett im Puffer ist, so müssen wir es an den Anfang des Puffers umkopieren. Dann starten wir die nächste Schleife und empfangen wieder Daten. Diesmal gehen wir allerdings nicht an den Anfang des Puffers, sondern an das Ende des halben Pakets, das nun ganz vorne im Puffer sitzt. Dies fügt das zerstückelte Paket wieder korrekt zusammen.

Das klingt nach einer spannenden Aufgabe. Die haben wir in unserer Implementierung wie folgt gelöst:



```
int g_PkgSize=sizeof(ZFXPACKAGE);

HRESULT ZFXSocketObject::Receive(SOCKET sk) {
    HRESULT hr      = ZFX_OK;
    UINT      nSize  = 65536; // max Bytes je Read
    UINT      nBytesRead = 0;  // gelesene Bytes
    UINT      nReadHead = 0;   // Position in m_Buffer
    UINT      n       = 0;     // verbleibende Daten
    bool      bDone   = false; // fertig?

    ZFXPACKAGE *pPkg      = NULL;
    UINT      nPkgSizeTotal = 0;

    // lies bis zu 65.536 Bytes in jedem Aufruf und
    // loope, bis keine Daten mehr am Socket warten
```

```

while (!bDone) {
    nBytesRead = recv(sk, &m_Buffer[n], nSize-n, 0);

    if (nBytesRead == SOCKET_ERROR) {
        int WSAError = WSAGetLastError();

        // ignoriere unkritische Meldungen
        if ( (WSAError != WSAEMSGSIZE) &&
            (WSAError != WSAEWOULDBLOCK) ) {
            hr = ZFX_FAIL;
            bDone = true;
            break;
        }
    }

    // wir haben nun nBytesRead Bytes im m_Buffer
    if (nBytesRead <= 0) bDone = true;
    else {
        // beachte alte Daten im Puffer
        nBytesRead += n;

        // loope, bis wir einen kompletten Header
        // finden
        while ( (nBytesRead-nReadHead) > g_PkgSize ) {
            // nächstes Stück Daten
            pPkg = (ZFXPACKAGE*)&m_Buffer[nReadHead];
            pPkg->pData = &m_Buffer[nReadHead] +
                g_PkgSize;

            nPkgSizeTotal = g_PkgSize + pPkg->nLength;

            // haben wir das ganze Paket empfangen?
            if ( (nBytesRead-nReadHead) >=
                (nPkgSizeTotal) ) {
                m_pInbox->Enqueue(pPkg, nPkgSizeTotal);
                nReadHead += nPkgSizeTotal;
            }
            // nein, also zurück zu recv()
        } else {
            // kopiere halbes Paket an den Anfang
            memcpy(m_Buffer, &m_Buffer[nReadHead],
                nBytesRead-nReadHead);
            n = nBytesRead-nReadHead;
            break;
        }
    } // while

    // wir haben bereits alle Daten, die warteten
    if (nBytesRead < nSize) bDone = true;
}

```

```

    } // while
    return hr;
} // Receive

```

Alternativen Eine alternative Lösung wäre es, wenn man zuerst alle Daten, die vorhanden sind, in einer Schleife empfängt. Dazu müsste man nach jedem Schleifendurchlauf den Puffer dynamisch um seine ursprüngliche Größe an Speicher erweitern (z.B. mit `realloc()`) und dann im folgenden Durchlauf dorthin die Daten empfangen. Das tut man so lange, bis man alle Daten empfangen hat. Dann kann man den Puffer ganz bequem sequenziell auf einmal auslesen.

Bei diesem Ansatz gibt es aber zwei Probleme: Zum einen müsste man unter Umständen zur Laufzeit ständig Speicher allokieren und verschieben, was nicht gerade sehr schnell ist, und zum anderen hätte man hier eventuell das Problem, dass der Speicher zur Neige geht, wenn man sehr viele Daten empfängt und Speicherbereiche in Größenordnungen von mehreren hundert Megabyte allokieren und verschieben muss.

Unser Ansatz mag daher recht verschnörkelt erscheinen und eine gewisse Grundeleganz vermissen lassen. Trotzdem handelt es sich dabei um die stabilere Variante.

Abfragen des Postfachs

Pakete ausgeben

Jetzt haben wir es so gut wie geschafft. Unsere Lowlevel-Kapselung des Windows Sockets-Objekts ist nahezu komplett. Da wir jedoch so frei waren, einem jeden Socket-Objekt auch eine eigene Inbox zu spendieren, brauchen wir natürlich auch die Funktionalität, um die Nachrichten aus der Inbox abzurufen. Falls also noch mindestens ein Päckchen in der Inbox ist, ziehen wir uns eine Kopie von dort und entfernen es aus der Inbox. Dies erledigt die Funktion `ZFXQueue::Front` für uns. Gespeichert wird das Paket zunächst im Puffer `m_Buffer` unserer Instanz. Von dort kopieren wir es in eine `ZFXPACKAGE`-Struktur, die beim Aufruf als Parameter angegeben wurde.

Serialisierung

Dies ist notwendig, da die `ZFXQueue::Front`-Funktion einen zusammenhängenden Speicherbereich erwartet, in den sie die Daten kopieren kann. Die `ZFXPACKAGE`-Struktur besteht aber aus zwei Speicherbereichen, da das Feld `pData` an einer ganz anderen Adresse liegen kann als der erste Teil der Struktur.



```
int g_PkgSize=sizeof(ZFXPACKAGE);
```

```

HRESULT ZFXSocketObject::GetNextPkg(ZFXPACKAGE *pPkg) {
    // ist überhaupt eins da?
    if (m_pInbox->GetCount() > 0) {
        // ziehe die Daten in den Puffer
        m_pInbox->Front(m_Buffer, true);
    }
}

```



```

// betanke die Struktur
memcpy(pPkg, m_Buffer, g_PkgSize);
memcpy(pPkg->pData, m_Buffer+g_PkgSize,
       pPkg->nLength);
return ZFX_OK;
}
return ZFX_FAIL;
} // GetNextPkg

```

Hier ist ebenfalls ersichtlich, dass der Aufrufer dieser Funktion auf alle Fälle bereits im Feld pData der ZFXPACKAGE-Struktur genügend Speicher allokiert haben muss, um alle Daten aufnehmen zu können. Dies ist deshalb so gelöst, weil die Daten aus der DLL typischerweise an eine Applikation ausgegeben werden. Und genau diese Applikation ist also konsequenterweise auch dafür verantwortlich, den Speicher bereitzustellen, ebenso wie sie ihn wieder freigeben muss.

Damit haben wir es geschafft. Wir haben nun eine sehr komfortabel zu bedienende Klasse für die Verwendung von Socket-Objekten. Unsere nächste Aufgabe wird es daher sein, die Verwendung dieses Objekts durch Einführung eines weiteren Abstraktionslayers zu vereinfachen. Die Klasse ZFXSocketObject ist nämlich außerhalb unserer DLL gar nicht bekannt. Dort arbeiten wir nur mit dem Interface ZFXNetworkDevice, das dann natürlich in der Implementierung ZFXWS auf unsere Socket-Objekte zurückgreift.



Interface

Interface-Definition für eine Netzwerk-Klasse

Es war schon ein bisschen unglaublich, als ich weiter oben gesagt habe, dass die Implementierung einer Netzwerkkomponente nicht viel mehr ist als das Versenden und Empfangen von Datenpaketen, oder? Insbesondere, nachdem wir doch recht viel an Code für das ZFXSocketObject entwickeln mussten. An dieser Stelle kann ich mich aber glücklicherweise wieder rehabilitieren, denn wie wir gleich an unserem Interface sehen werden, bietet dieses neben den üblichen Initialisierungs- und Freigabefunktionen lediglich Methoden an, die dem Datentransfer dienen. Dadurch, dass wir den ganzen Lowlevel-Kram in einer für den Anwender unsichtbaren Klasse quasi weggeschlossen haben, ist unser Interface ZFXNetworkDevice das reinste Pfannkuchen-Rezept.

Alles ist so einfach, wie man es sich macht

```

#define WM_ZFXSERVER (WM_USER + 1)
#define WM_ZFXCLIENT (WM_USER + 2)

```

```

typedef enum ZFXNETMODE_TYPE {
    NMD_SERVER=0,
    NMD_CLIENT=1,
} ZFXNETMODE;

```



```

class ZFXNetworkDevice {
protected:
    HWND      m_hWndMain;          // Window Handle
    HINSTANCE m_hDLL;              // DLL-Handle
    bool      m_bRunning;
    int       m_nPort;             // Port-Nummer
    char      m_pIP[256];          // IP-Adresse
    UINT      m_nMaxSize;          // Puffer-Größe

public:
    ZFXNetworkDevice(void) {};
    virtual ~ZFXNetworkDevice(void) {};

    // Init, Freigabe und MISC
    virtual HRESULT Init(HWND, ZFXNETMODE, int nPort,
                        char *pIP, UINT nMaxPkgSize,
                        bool)=0;
    virtual void    Release(void)=0;
    virtual bool    IsRunning(void)=0;

    // Nachrichten-Prozedur
    virtual HRESULT MsgProc(WPARAM, LPARAM)=0;

    // Senden und Empfangen
    virtual HRESULT SendToServer(const ZFXPACKAGE*)=0;
    virtual HRESULT SendToClients(const ZFXPACKAGE*)=0;

    // Informationen über die Inbox
    virtual bool    IsPkgWaiting(void)=0;
    virtual UINT    GetNextPkgSize(void)=0;
    virtual HRESULT GetNextPkg(ZFXPACKAGE*)=0;
}; // class

```

Nachrichten

Dieses Interface dient dem Anwender dazu, sich ein Device zu erzeugen, das er wahlweise als Server oder als Client initialisieren kann. Die Funktionen sollten größtenteils selbsterklärend sein, lediglich auf die `ZFXNetworkDevice::MsgProc` möchte ich noch kurz eingehen. Bisher haben wir schon viele Funktionen gesehen, mit denen man über einen Socket Daten senden oder empfangen kann. Die Frage ist nur: Wann ruft man solche Funktionen auf? Woher weiß man beispielsweise, dass Daten an einem Socket auf Empfang warten oder dass sich ein neuer Client anmelden möchte?

All dies erledigen wir unter der Windows Socket-API durch Nachrichten, so wie wir in jedem anderen Windows-Programm alle möglichen anderen Events auch registrieren und durch eine Windows-Prozedur abarbeiten. Dazu kann man bei der Initialisierung der Windows Sockets-API auch eigene Nachrichten angeben, die immer dann an die Applikation gesendet werden, wenn ein Event im Netzwerk auftritt. Wir haben hier die Nachrichten `WM_ZFXSERVER` und `WM_ZFXCLIENT` definiert. In Applikationen, die unsere

Netzwerk-DLL verwenden, muss man dann in der Windows-Prozedur lediglich auf diese beiden Nachrichten prüfen. Die Windows Sockets-API hat dann die `WPARAM`- und `LPARAM`-Komponenten der Nachricht mit entsprechenden Werten betankt, die wir interpretieren können. Und genau das machen wir in der Funktion `ZFXNetworkDevice::MsgProc`, die eine Applikation immer dann aufrufen muss, wenn `WM_ZFXSERVER` oder `WM_ZFXCLIENT` als Nachricht aufgetreten ist.

Windows Sockets-Kapselung

Machen wir uns also an die Arbeit, und implementieren wir dieses Interface. Diese Implementierung ist nach außen hin die einzige Klasse, die der Anwender unserer DLL sehen kann. Nach innen hin verwendet sie jedoch größtenteils die Funktionen der `ZFXSocketObject`-Klasse, um den gesamten WinSock-Kram elegant zu kapseln, so dass beispielsweise die Abläufe der Initialisierung wesentlich transparenter werden und nicht durch die Initialisierung von Adress-Strukturen oder Ähnlichem aufgeplustert werden. Das Auge programmiert schließlich mit.

ZFXWS-Klasse

Wie schon so oft beginnen wir hier mit einem Blick auf die Definition der Klasse, die das Interface implementiert. Erfreulicherweise ist es auch hier wieder so, dass die Klasse fast ausschließlich mit den Methoden des Interfaces auskommt und wenig eigene Member ergänzen muss.

*Umsetzung des
Interfaces*

```
class ZFXWS : public ZFXNetworkDevice {
public:
    ZFXWS(HINSTANCE hDLL);
    ~ZFXWS(void);

    // Interface-Funktionen
    HRESULT Init(HWND, ZFXNETMODE, int, char*, UINT,
                bool);
    void    Release(void);
    bool    IsRunning(void) { return m_bRunning; }
    HRESULT MsgProc(WPARAM wp, LPARAM lp);
    HRESULT SendToServer(const ZFXPACKAGE*);
    HRESULT SendToClients(const ZFXPACKAGE*);
    HRESULT ServerUpdate(void);
    bool    IsPkgWaiting(void)
        { return m_pSockObj->IsPkgWaiting(); }
    UINT    GetNextPkgSize(void)
        { return m_pSockObj->GetNextPkgSize(); }
    HRESULT GetNextPkg(ZFXPACKAGE *pPkg)
        { return m_pSockObj->GetNextPkg(pPkg); }
```



```

private:
    ZFXSocketObject *m_pSockObj;
    ZFXNETMODE      m_Mode;
    ZFXCLIENT       m_Clients[256];
    char            m_C1Count;
    UINT            m_C1ID;
    char            *m_Buffer;

    // ZFXSocketObject initialisieren
    HRESULT CreateServer(ZFXSocketObject
                        **ppSkObject);
    HRESULT CreateClient(ZFXSocketObject
                        **ppSkObject);

    // zur Nachrichtenverarbeitung
    HRESULT OnAccept(void);
    HRESULT OnReceive(SOCKET skReceiving);
    HRESULT OnDisconnect(SOCKET skDisconnecting);
}; // class

```

Attribute Die meisten der Attribute sind nur für einen Server notwendig. Der Client verwendet lediglich auch das Socket-Objekt, merkt sich den Modus und bietet einen Puffer. Die anderen Attribute nutzt der Server dazu, um sich eine Liste von angemeldeten und akzeptierten Clients aufzubauen. `m_C1ID` ist eine ID, die jedem neuen Client zugeteilt wird. Dabei handelt es sich um einen simplen Zähler, der für jeden neuen Client inkrementiert wird. `m_C1Count` ist ein Zähler, der die aktiv konnektierten Clients mitzählt. Dieser wird beispielsweise auch dekrementiert, wenn sich ein Client abmeldet. Da es sich hier um eine `char`-Variable handelt, können nicht mehr als 256 Clients gleichzeitig im Netzwerk sein. Für gewöhnlich sind session-basierte Spiele aber ohnehin auf 32 oder 64 Spieler beschränkt, weil sonst keine ausreichende Geschwindigkeit garantiert werden kann.

Das Array `m_Clients` beinhaltet schließlich alle aktiv konnektierten Clients, die sich im Netzwerk befinden. Dazu haben wir die Struktur `ZFXCLIENT` wie folgt definiert:



```

typedef struct ZFXCLIENT_TYPE {
    SOCKET skToClient;
    UINT   nID;
} ZFXCLIENT;

```

Das `SOCKET`-Objekt ist eben jener Socket, über den der Server mit genau dem Client kommunizieren kann, der durch die `accept()`-Funktion erzeugt wurde. Das Feld `nID` ist die ID des Clients und entspricht dem Wert von `m_C1ID`, den der Client erhalten hat.

Neben den paar zusätzlichen Attributen fallen die beiden Methoden `ZFXWS::Create ...` auf. Je nachdem, ob der `ZFXNETMODE`-Parameter in der Initialisierung auf Server oder Client steht, wird die entsprechende Funktion aufgerufen. Wie in Tabelle 11.2 gezeigt, muss man ja für Server und Client verschiedene Schritte bei der Initialisierung ausführen. Die restlichen drei Methoden der Klasse werden von der Methode `ZFXWS::MsgProc` für die entsprechenden Fälle aufgerufen. Das sehen wir später noch im Detail, aber die Namensgebung sollte hier bereits darauf hindeuten, welche Fälle dort bearbeitet werden.

Methoden

Bevor wir uns in die Arbeit stürzen, schreiben wir noch schnell den Konstruktor und den Destruktor. Dabei haben wir mal wieder nicht so viel zu tun. Man beachte jedoch bei der Initialisierung von `m_ClID`, dass wir dieses Attribut mit 1 und nicht mit 0 initialisieren. Damit erhält der erste sich anmeldende Client die ID Nummer 1, denn eine ID von 0 identifiziert ja per Definition den Server.

Konstruktor und Destruktor

```
ZFXWS::ZFXWS(HINSTANCE hDLL) {
    m_hDLL      = hDLL;
    m_pSockObj  = NULL;
    m_Buffer    = NULL;
    m_nPort     = 0;
    m_ClID      = 1;          // 0 reserviert für Server
    m_ClCount   = 0;
    m_bRunning  = false;
} // constructor

/*-----*/

ZFXWS::~ZFXWS() { Release(); } // destructor
/*-----*/

void ZFXWS::Release() {
    if (m_Mode == NMD_SERVER) {
        for (int i=0; i<m_ClCount; i++) {
            shutdown(m_Clients[i].skToClient, 0x02);
            closesocket(m_Clients[i].skToClient);
            m_Clients[i].skToClient = INVALID_SOCKET;
        }
    }
    if (m_pSockObj) {
        delete m_pSockObj;
        m_pSockObj = NULL;
    }
    if (m_Buffer) {
        delete [] m_Buffer;
        m_Buffer = NULL;
    }
    WSACleanup();
    m_bRunning = false;
}
```



**Sauber
aufräumen**

Beim Aufräumen müssen wir zunächst einmal beachten, dass wir auch alle Client-Sockets freigeben müssen, die ein Server im Laufe der Laufzeit gebunkert hat. Dazu rufen wir hier zu Fuß die Funktionen `shutdown()` und `closesocket()` der WinSock-API auf, da wir die Client-Socket-Objekte des Servers ja nicht in `ZFXSocketObject`-Instanzen gespeichert haben, sondern nur in der Struktur `ZFXCLIENT`.

**WinSock-API
herunterfahren**

Für Client und Server ist dann wieder gleich, dass wir das eigene Socket-Attribut freigeben müssen, ebenso wie den Puffer für das Empfangen der Daten. Abschließend müssen wir die WinSock-API selbst noch herunterfahren, was die Funktion `WSACleanup()` erledigt. Natürlich muss man die WinSock-API zuerst auch initialisieren, doch das erledigen wir natürlich in der Initialisierungsfunktion der Klasse.

Initialisieren des Netzwerks**Initialisierung der
WinSock-API**

Der erste Schritt bei der Initialisierung ist das Starten der WinSock-API. Dazu gibt es die zweiparametrische Funktion `WSAStartup()`. Im ersten Parameter geben wir die Nummer der Version an, die wir verwenden wollen. Wir arbeiten hier mit der Version 2.0 von WinSock. Der zweite Parameter ist die Struktur `WSADATA`, die von der Funktion mit Informationen über die tatsächliche Implementierung der gefragten Version gefüllt wird. Diese Informationen sind für uns aber nicht weiter relevant.

**Initialisierung des
Servers oder
Clients**

Der zweite und letzte Schritt in der Initialisierung einer Instanz der Klasse `ZFXWS` ist die Initialisierung des `ZFXSocketObject`-Attributs der Instanz. Je nach Wunsch des Aufrufers tun wir dies in Form eines Servers oder eines Clients, wozu wir die Funktion `ZFXWS::CreateServer` oder `ZFXWS::CreateClient` verwenden. Im Quellcode sieht das Ganze dann genauso einfach aus, wie es eben klang:



```
HRESULT ZFXWS::Init(HWND hWnd, ZFXNETMODE nmd,
                    int nPort, char *pIP,
                    UINT nMaxPkgSize, bool bSaveLog)
{
    WSADATA wsaData;
    UINT    nEvents=0;
    WORD    wVersion;
    int     nRes;

    m_nMaxSize = nMaxPkgSize;
    m_Buffer   = new char[m_nMaxSize];
    m_hWndMain = hWnd;
    m_nPort    = nPort;
    m_Mode     = nmd;
    g_bLF      = bSaveLog;

    if (pIP) sprintf(m_pIP, "%s", pIP);
```

```

wVersion = MAKEWORD(2,0);

if ( (nRes = WSASStartup(wVersion, &wsaData)) != 0)
    return ZFX_FAIL;

// Erzeuge SocketObject als Server horchend am Port
if (m_Mode==NMD_SERVER) {
    if ( FAILED( CreateServer(&m_pSockObj) ))
        return ZFX_FAIL;
    }
// Erzeuge SocketObject als Client
else if (m_Mode==NMD_CLIENT) {
    if (strcmp(m_pIP, "")==0)
        sprintf(m_pIP, "LOCALHOST");

    if ( FAILED( CreateClient(&m_pSockObj) ))
        return ZFX_FAIL;
    }
else return ZFX_INVALIDPARAM;

m_bRunning = true;
return ZFX_OK;
} // Init

```

Logischerweise benötigt man für einen Server lediglich die Nummer des Ports, über den die Kommunikation des Netzwerks abläuft. Ein Client hingegen muss dieselbe Port-Nummer kennen und zudem noch die IP-Adresse des Servers. Startet man den Client jedoch auf demselben Computer wie den Server, so kann man anstelle der IP-Adresse des Servers oder des Servernamens auch den String `LOCALHOST` als Servernamen verwenden.

Lokaler Host

Erstellen eines Servers

Nun sollten wir noch mal einen Blick auf Tabelle 11.2 werfen, denn nun werden wir uns ansehen, wie wir das `ZFXSocketObject`-Attribut der Klasse `ZFXWS` als Server initialisieren können. Besagte Tabelle listet die dazu notwendigen Schritte auf. Aber weil ich von mir auf andere schließe und von der Faulheit des Lesers ausgehe, liste ich diese Schritte hier nochmals auf:

*Schritte zur
Erzeugung eines
Servers*

- ➡ `CreateSocket()`
- ➡ `Bind()`
- ➡ `Listen()`

Es kommt gleich noch ein vierter Schritt hinzu, aber sehen wir uns erst einmal die Umsetzung der ersten drei Schritte an:



```

HRESULT ZFXWS::CreateServer(ZFXSocketObject
                           **ppSkObject)
{
    UINT nEvents=0;

    (*ppSkObject) = new ZFXSocketObject(m_pLog);

    if (!(*ppSkObject))
        return ZFX_FAIL;

    // 1. Schritt: Erzeuge Socket-Objekt
    if (FAILED( (*ppSkObject)->CreateSocket() ))
        return ZFX_FAIL;

    // 2. Schritt: Gib dem Kind einen Namen
    if (FAILED( (*ppSkObject)->Bind(m_nPort) ))
        return ZFX_FAIL;

    // 3. Schritt: Besetze den Horchposten
    if (FAILED( (*ppSkObject)->Listen() ))
        return ZFX_FAIL;

    nEvents |= FD_READ | FD_WRITE | FD_CONNECT |
              FD_ACCEPT | FD_CLOSE;

    // 4. Schritt: Windows-Benachrichtigung aktivieren
    if ( WSAAsyncSelect((*ppSkObject)->GetSocket(),
                      m_hWndMain, WM_ZFXSERVER,
                      nEvents) == SOCKET_ERROR)
    {
        m_pSockObj->Disconnect();
        return ZFX_FAIL;
    }

    // setze alle Clients als invalide
    for (int i=0; i<256; i++) {
        m_Clients[i].skToClient = INVALID_SOCKET;
        m_Clients[i].nID = 0;
    }
    return ZFX_OK;
} // CreateServer

```

**Asynchroner
Nachrichten-
verkehr**

Wir unterscheiden bei der WinSock-API grundsätzlich zwischen dem synchronen und dem asynchronen Modus. Im synchronen Modus würden gewisse Funktionsaufrufe blockierend wirken. Das bedeutet, wenn wir eine solche Funktion aufrufen, dann würde das gesamte Programm angehalten werden, bis die Funktion erfolgreich war. Beispielsweise ist die `accept()`-Funktion so ein Kandidat. Das bedeutet, dass der Server so lange still stehen würde, bis sich ein Client angemeldet hat. Danach würden keine weiteren

Anmeldungen mehr akzeptiert werden können, es sei denn, man ruft wieder `accept()` auf und blockiert damit so lange die Anwendung, bis wirklich ein neuer Client konnektieren möchte. Aus dieser Misere gibt es einen unschönen und einen schönen Ausweg.

Der unschöne Ausweg ist die Verwendung von mehreren Threads bei einem Multitasking-Betriebssystem. Hier erzeugt man für diese Blocking-Calls jeweils eigene Threads, in denen man auf ankommende Clients wartet oder über die man Daten versendet und empfängt. Ein Beispiel dazu findet ihr im Buch von Todd Barron.² Diese Methode erzeugt jedoch einen Rattenschwanz von Problemen mit Threads und der Datenkonsistenz zwischen den verschiedenen Threads.

*Threads gegen
Blocking-
Funktionen*

Wenn man jedoch für die Windows-Betriebssysteme entwickelt, dann hat man eine schönere, zweite Möglichkeit zur Auswahl. Diese nennt man *asynchronen Modus*, und sie beruht auf dem typischen Callback-Verfahren, das auch die Grundlage von Windows selbst ist. Hier wird nicht mehr synchron gearbeitet, wo man auf einen Event sofort reagieren muss, wenn dieser auftritt. Vielmehr gibt man der WinSock-API einfach eine Nachricht an, mit der sie uns informieren soll, wenn ein Event auftritt, der für uns interessant sein könnte. Diese Nachricht wird wie eine ganz normale Windows-Nachricht (z.B. `WM_PAINT`) an unsere Applikation gesendet, und es bleibt uns überlassen, wann wir darauf reagieren möchten. Damit umgehen wir die *Blocking-Calls*, die spätestens dann problematisch werden, wenn wir mehr als einen Client im Netzwerk haben. Ebenso umgehen wir dadurch die Probleme, die sich aus einer Anwendung mit mehreren Threads ergeben. Wenn man diese Option zur Verfügung hat, dann sollte man sie auf alle Fälle nutzen. Siehe hierzu auch die weiterführenden Beschreibungen im exzellenten Buch von Bob Quinn und Dave Shute.³

*Windows gegen
Blocking-
Funktionen*

Wenn wir uns für den asynchronen Modus entscheiden, rufen wir die Funktion `WSAAsyncSelect()` der WinSock-API auf. Dieser müssen wir im ersten Parameter den Socket angeben, der im asynchronen Modus betrieben werden soll. Der zweite Parameter ist das Handle auf das Hauptfenster der Applikation, die unsere DLL verwendet. Im dritten Parameter geben wir dann die Nachricht an, die die WinSock-API beim Auftreten eines Events an unsere Applikation senden soll. Hierfür haben wir weiter oben schon `WM_ZFXSERVER` und `WM_ZFXCLIENT` definiert. Interessant ist dann noch der vierte Parameter, denn hier müssen wir angeben, bei welchen Events überhaupt eine Nachricht gesendet werden soll. Die folgenden stehen dabei unter anderem zur Auswahl:

*WSAAsync-
Select() und
Events der
WinSock-API*

- ➔ `FD_READ`: Es sind zu empfangende Daten am Socket.
- ➔ `FD_WRITE`: Daten wurden über den Socket versendet.

2 Multiplayer Game Programming; Prima Publishing, 2001

3 Windows Sockets Network Programming; Addison-Wesley, 1996

- ➔ `FD_CONNECT`: Ein Client wurde akzeptiert.
- ➔ `FD_ACCEPT`: Ein neuer Client möchte akzeptiert werden.
- ➔ `FD_CLOSE`: Ein Client meldet sich ab.

Wie man im Code oben erkennen kann, muss ein Server relativ viele Events bearbeiten. Wie wir gleich bei dem Client sehen werden, sind es dort viel weniger. Zu `FD_CONNECT` und `FD_WRITE` möchte ich noch erwähnen, dass unser Server zwar über diese Fälle informiert wird, wir diese Nachrichten aber nicht weiter bearbeiten. Die Bestätigung einer erfolgreichen Konnektierung könnte man dazu nutzen, um Initialisierungsinformationen an den Client zu senden. Von Seiten des Servers gibt es da aber nichts zu unternehmen, also überlassen wir das der Applikation, die unsere DLL später verwendet.

Sehen wir uns nun die Initialisierung für einen Client an.

Erstellen eines Clients

*Schritte zur
Erzeugung eines
Clients*

Da der Server ja die Hauptarbeit leistet und der Client quasi nur Nutznießer ist, sieht auch die Initialisierung des Clients wesentlich schlanker aus als die des Servers. Wir benötigen hier lediglich zwei Schritte:

- ➔ `CreateSocket()`
- ➔ `Connect()`

Auch hier kommt noch der weitere Schritt hinzu, dass wir die Windows-Benachrichtig aktivieren. Und so sieht das Ganze im Code aus:



```
HRESULT ZFXWS::CreateClient(ZFXSocketObject
                           **ppSkObject) {
    UINT nEvents=0;

    (*ppSkObject) = new ZFXSocketObject(m_pLog);

    if (!(*ppSkObject))
        return ZFX_FAIL;

    // 1. Schritt: Erzeuge ein Socket-Objekt
    if (FAILED( (*ppSkObject)->CreateSocket() ))
        return ZFX_FAIL;

    if (m_pIP == NULL)
        gethostname(m_pIP, 10);

    // 2. Schritt: Versuche einen Konnekt zum Server
    if (FAILED( (*ppSkObject)->Connect(m_pIP, m_nPort) ))
        return ZFX_FAIL;
```

```

nEvents |= FD_READ | FD_CLOSE;

// 3. Schritt: Windows-Benachrichtigung aktivieren
if (WSAAsyncSelect( (*ppSkObject)->GetSocket(),
                    m_hWndMain, WM_ZFXCLIENT,
                    nEvents) == SOCKET_ERROR) {
    m_pSockObj->Disconnect();
    return ZFX_FAIL;
}
return ZFX_OK;
} // Create Client

```

Im Gegensatz zu dem Server hat der Client selbst nicht so viele Events, die überhaupt bei ihm auftreten können. Wichtig sind hierbei nur die Events, die uns mitteilen, dass sich der offene Socket geschlossen hat. Dies bedeutet für den Client, dass sich der Server abgemeldet hat. Die Nachricht `FD_READ` ist auch existenziell für den Client, denn dies ist für ihn die Benachrichtigung, dass an seinem Socket Daten angekommen sind, die empfangen werden müssen.

Benachrichtigungs-Events

Damit ist die Initialisierung unseres gekapselten Socket-Objekts entweder für den Server oder den Client abgeschlossen, und wir können uns dem laufenden Betrieb des Netzwerks zuwenden. Von nun ab läuft fast alles im Netzwerk über Nachrichten. Eine Ausnahme von dieser Regel bildet lediglich das Senden von Nachrichten. Dies generiert bei dem Empfänger zwar das Event `FD_READ`, kann aber für den Sender direkt von der Applikation aufgerufen werden.

Initialisierung komplett

Daten an den Server senden

Kommen wir nun zu dem Versenden von Daten über das Netzwerk. Im Prototyp der Methode in der Klassendefinition konnte man ja schon sehen, dass wir hier ein Objekt von Typ `ZFXPACKAGE` versenden können. Doch hier muss man ein wenig Vorsicht walten lassen. Schließlich enthält diese Struktur auch einen `void-Pointer`, so dass die Größe der Struktur nicht mit `sizeof()` bestimmt werden kann. Vielmehr muss man zu diesem Wert noch den Wert des Feldes `nLength` addieren, um die Gesamtgröße der Struktur zu haben. Doch damit haben wir noch nicht alle Fallstricke beseitigt: Wenn wir der `ZFXSocketObject::Send-Methode` einfach einen Pointer auf ein `ZFXPACKAGE-Objekt pPkg` übergeben, so würde es ab dieser Speicheradresse auf dort liegende Daten der Länge `sizeof(ZFXPACKAGE)+pPkg->nLength` zugreifen. Das Problem dabei ist aber, dass der Speicherbereich an der Adresse `pPkg->pData` ganz woanders im Speicher liegen kann und für gewöhnlich auch ganz woanders liegen wird. Um also das gesamte Paket inklusive des Datenpakets in `pData` versenden zu können, müssen wir alle diese Daten hintereinander weg linear im Speicher liegen haben. Um dies sicherzustellen, führen wir vor dem Versenden der Daten eine Serialisierung der Daten durch, indem wir zuerst die Daten aus der `ZFXPACKAGE-Struktur` und dann die Daten aus dem

Serialisierung der Daten vor dem Versenden

pData-Zeiger der Struktur hintereinander in den Puffer der Klasse ZFXWS kopieren und anschließend von dort aus versenden.

Abgesehen von diesem kleinen Stolperstein, ist die Methode dann wirklich so einfach, wie man es sich wünschen würde:



```
int g_PkgSize=sizeof(ZFXPACKAGE);

HRESULT ZFXWS::SendToServer(const ZFXPACKAGE *pPkg) {
    int nBytes=0;
    int nSize=g_PkgSize+pPkg->nLength;

    if (m_Mode != NMD_CLIENT) return ZFX_FAIL;
    if (nSize > m_nMaxSize) return ZFX_OUTOFMEMORY;

    // Daten-Serialisierung
    memcpy(m_Buffer, pPkg, g_PkgSize);
    memcpy(m_Buffer+g_PkgSize, pPkg->pData,
           pPkg->nLength);

    nBytes = m_pSockObj->Send(m_Buffer, nSize);
    if ( (nBytes==SOCKET_ERROR) || (nBytes<nSize) )
        return ZFX_FAIL;
    return ZFX_OK;
} // SendToServer
```

Konstante Werte vorberechnen

Wie ihr auch hier wieder sehen könnt, bestimmen wir die Größe der ZFX-PACKAGE-Struktur nicht ständig neu, sondern berechnen sie einmal in unserem Projekt und speichern sie dann als globale Konstante ab. Da wir diesen Wert in unserer DLL sehr häufig brauchen, spart uns dies ein paar Zyklen der CPU je Frame ein. Man sollte in seinen Methoden grundsätzlich nach solchen oft berechneten, aber immer konstant bleibenden Werten Ausschau halten und sie entsprechend vorberechnen.

Daten an die Clients senden

An wen sendet der Server?

Ganz analog wie das Senden von Daten an den Server im Falle eines Clients funktioniert auch das Senden von Daten des Servers an alle Clients. Natürlich muss der Server seine Daten nicht immer an alle Clients verschicken; bei bestimmten Applikationen kann ein Server durchaus die Aufgabe haben, gezielt Nachrichten an einzelne Clients zu schicken. Bei Multiplayer-Computerspielen wird es aber der Regelfall sein, dass der Server seine Nachrichten grundsätzlich an alle aktiven Clients verschickt. Daher implementieren wir dies als entsprechende Methode in unserer Klasse.

```

HRESULT ZFXWS::SendToClients(const ZFXPACKAGE *pPkg) {
    HRESULT hr=ZFX_OK;
    int     nBytes=0;
    int     nSize=g_PkgSize+pPkg->nLength;

    if (m_Mode != NMD_SERVER) return ZFX_FAIL;
    if (nSize > m_nMaxSize) return ZFX_OUTOFMEMORY;

    // Daten-Serialisierung
    memcpy(m_Buffer, pPkg, g_PkgSize);
    memcpy(m_Buffer+g_PkgSize, pPkg->pData,
           pPkg->nLength);

    for (UINT i=0; i<m_ClCount; i++) {
        if (m_Clients[i].skToClient != INVALID_SOCKET) {
            nBytes = m_pSockObj->Send(m_Buffer, nSize,
                                     m_Clients[i].skToClient);

            if ( (nBytes==SOCKET_ERROR) || (nBytes<nSize) )
                hr = ZFX_FAIL;
        }
    }
    return hr;
} // SendToClients

```

Der Server läuft hier durch die Liste aller angemeldeten Clients und sendet ihnen dasselbe Datenpaket zu. Man beachte, dass die Funktion zwar eine Fehlermeldung liefert, wenn das Senden an einen oder mehrere Clients fehlerhaft war, aber dennoch versucht, alle verbleibenden Clients sauber mit dem Paket zu beliefern.

Nachrichten-Prozedur

Da wir mit der asynchronen Methode für unsere Sockets arbeiten, erhält die Applikation, die unsere DLL verwendet, von Windows automatisch die Nachrichten WM_ZFXSERVER und WM_ZFXCLIENT, wenn es Aktivität in Form der registrierten Events im Netzwerk gibt. In diesem Fall enthält der untere Teil des LPARAM-Teils der Nachricht das Event, das aufgetreten ist. Der WPARAM-Teil der Nachricht hingegen entspricht dem Socket-Objekt, an dem das Event registriert wurde. Unsere DLL stellt nun eine Prozedur zur Bearbeitung dieser beiden Nachrichten zur Verfügung, der die Applikation einfach die beiden Parameter der Nachricht übergeben muss.

```

HRESULT ZFXWS::MsgProc(WPARAM wp, LPARAM lp) {
    WORD  wEvent, wError;

    // Daten holen
    wError = HIWORD(lp);
    wEvent = LOWORD(lp);

```



*Bearbeitung der
Netzwerk-Events*



```

// Aktivität auswählen
switch (wEvent) {
    // ein neuer Client ist akzeptiert worden
    case FD_CONNECT: break;

    // ein neuer Client meldet sich an
    case FD_ACCEPT: { return OnAccept(); } break;

    // Daten am Socket zum Empfangen
    case FD_READ: { return OnReceive(wp); } break;

    // ein Socket schließt sich
    case FD_CLOSE: { return OnDisconnect(wp); }
                    break;

    // nach dem Senden
    case FD_WRITE: break;

}
return ZFX_OK;
} // MsgProc

```

Die beiden Events `FD_CONNECT` und `FD_WRITE` lassen wir unbearbeitet, denn auf sie wollen wir nicht reagieren. Für die anderen drei Events haben wir entsprechende Subroutinen implementiert, damit die Nachrichten-Prozedur schön übersichtlich bleibt.

Anmeldung von Clients

*Ein neuer Client
klopft an*

Wenn wir einen Server betreiben, wird es nicht selten vorkommen, dass das Event `FD_ACCEPT` auftritt. Es zeigt dem Server an, dass ein Client die Funktion `connect()` ausgeführt hat und in unser Netzwerk möchte. Der Server reagiert gewohnt freundlich darauf und akzeptiert den neuen Client mit einem neuen Socket-Objekt, das er speziell für diesen Client anlegt.

*Die restlichen
Clients informieren*

Ist dies geschafft, dann hat der Server noch zwei weitere Dinge zu erledigen. Zuerst muss er dem Client mitteilen, welche ID er hat. Dazu schnürt der Server ein Paket mit der ID-Nummer 0, die wir per Definition als Meldung des Servers an den Client mit dem Inhalt der zugewiesenen ID bestimmt haben. Dann sendet der Server dieses Päckchen an den neu angemeldeten Client. Die Applikation hat dann sicherzustellen, dass der Client dieses Päckchen bearbeitet und seine ID speichert.

Die zweite Aufgabe des Servers besteht darin, alle anderen Clients über den Neuzugang zu informieren. Dazu ändern wir die ID des Päckchens einfach auf die Nummer 1, die bei uns per Definition genau diese Meldung des Servers an alle Clients definiert. Dann schicken wir das Päckchen an alle

Clients, so dass eine Applikation beispielsweise auf dem Bildschirm melden kann, dass ein neuer Spieler am laufenden Spiel teilnimmt.

```
HRESULT ZFXWS::OnAccept(void) {
    int nSize=0, nBytes=0, i=m_ClCount;

    if (m_ClCount >= 255) return ZFX_OUTOFMEMORY;

    if ( FAILED( m_pSockObj->Accept(
        &(m_Clients[i].skToClient) )))
        return ZFX_FAIL;

    // SENDE DEM CLIENT SEINE ZUGEWIESENE ID:
    ZFXPACKAGE *pPkg = (ZFXPACKAGE*)m_Buffer;
    pPkg->pData = &m_Buffer[g_PkgSize];
    pPkg->nLength = sizeof(UINT);
    pPkg->nType = 0;    // ID Msg
    pPkg->nSender = 0;   // Server
    memcpy(pPkg->pData, &m_ClID, sizeof(UINT));

    // erhöhe die Zähler
    m_Clients[i].nID = m_ClID;
    m_ClCount++;
    m_ClID++;

    nSize = g_PkgSize + pPkg->nLength;
    nBytes = m_pSockObj->Send(m_Buffer, nSize,
        m_Clients[i].skToClient);
    if ( (nBytes==SOCKET_ERROR) || (nBytes<nSize) )
        return ZFX_FAIL;

    // INFORMIERE ALLE ANDEREN CLIENTS ÜBER DEN NEUZUGANG
    pPkg->nType = 1;
    SendToClients(pPkg);
    return ZFX_OK;
} // OnAccept
```



Abmeldung von Clients

Sollte ein Spieler genug von unseren Spielchen haben und das Netzwerk verlassen wollen, so erhält der Server eine entsprechende Meldung. Dies passiert beispielsweise dann, wenn eine Applikation ihr `ZFXNetworkDevice`-Objekt, das als Client betrieben wurde, wieder freigibt. Sobald der Server von diesem Disconnect etwas mitbekommt, muss er entsprechende Schritte ausführen. Zuerst muss er den Client aus seiner Liste herausuchen. Hat er diesen gefunden, schließt er den zugehörigen Socket, da dieser nun nicht mehr benötigt wird. Das andere Ende der Leitung ist ja sowieso schon tot. Nun ist die Arbeit schon so gut wie erledigt. Der Server schnürt nur noch schnell ein Päckchen mit der entsprechenden Meldung über den Abgang

eines Clients und verschickt dieses an alle verbleibenden Clients. So kann eine Applikation die übrigen Spieler darüber informieren, dass ein Spieler das Spiel verlassen hat.



```
HRESULT ZFXWS::OnDisconnect(SOCKET skDisconnecting) {
    ZFXPACKAGE Pkg;
    UCHAR      i=0;

    if (skDisconnecting==INVALID_SOCKET) return ZFX_FAIL;

    if (m_Mode==NMD_SERVER) {
        // lösche ihn aus der Liste
        for (i=0; i<m_ClCount; i++) {
            if (m_Clients[i].skToClient == skDisconnecting)
                break;
        }
        if (i>=m_ClCount)
            return ZFX_FAIL;

        // schließe den Socket
        shutdown(m_Clients[i].skToClient,0x02);
        closesocket(m_Clients[i].skToClient);
        m_Clients[i].skToClient = INVALID_SOCKET;

        // INFORMIERE DIE ANDEREN ÜBER DEN ABGANG
        Pkg.pData = &m_Buffer[g_PkgSize];
        Pkg.nLength = sizeof(UINT);
        Pkg.nType = 2; // ID disconnecting Message
        Pkg.nSender = 0; // Server
        memcpy(Pkg.pData, &m_Clients[i].nID,
            sizeof(UINT));

        SendToClients(&Pkg);

        // kopiere den letzten Eintrag um
        memcpy(&m_Clients[i], &m_Clients[m_ClCount-1],
            sizeof(ZFXCLIENT));
        m_ClCount--;
    }
    else {
        shutdown(m_pSockObj->GetSocket(),0x02);
        closesocket(m_pSockObj->GetSocket());
    }
    return ZFX_OK;
} // OnDisconnect
```

Wie ihr seht, verfügt auch ein Client über die Möglichkeit, eine Disconnect-Nachricht zu erhalten. In diesem Fall muss er einfach sein eigenes Socket-Objekt dicht machen.

Pakete über das Netzwerk empfangen

Wir nähern uns mit großen Schritten der Fertigstellung unserer DLL. Man mag es glauben oder nicht, aber die paar Seiten Quelltext reichen vollkommen aus, um eine lauffähige Netzwerk-Anwendung jeglicher Art zu implementieren. Unsere DLL bietet die grundlegendsten Techniken an, mit denen eine Applikation einen Client- und einen Server-Part für einen Netzwerk-Chat, ein Netzwerk-Tool oder natürlich ein Netzwerk-Spiel implementieren kann.

Uns fehlt lediglich noch die Methode, mit der wir den Empfang von Daten einleiten, falls unser Device das Event `FD_READ` erhält und damit weiß, dass an seinem Socket Daten zum Empfangen bereitliegen.

```
HRESULT ZFXWS::OnReceive(SOCKET skReceiving) {  
    if (m_bRunning)  
        return m_pSockObj->Receive(skReceiving);  
    else  
        return ZFX_FAIL;  
} // OnReceive
```



*ID des Senders
ermitteln*

Als Parameter der Nachricht wird hier praktischerweise der Socket übergeben, an dem die Daten aufgeschlagen sind. Im Falle eines Servers muss man daher nicht einmal prüfen, um welchen Client es sich handelt, da wir direkt den Socket zur Verfügung haben. Aus diesem Grunde ist es aber auch hilfreich, wenn bei jeder Nachricht die ID des Absenders beiliegt, damit der Server den Client zur Not auch identifizieren kann, falls dies nötig sein sollte. Alternativ könnten wir nun die Liste der Clients durchlaufen und die dortigen Sockets mit diesem hier vergleichen. Einer aus der Liste muss der gesuchte sein, und dann kennen wir auch seine ID, da dieser Socket dann ja zu dem Sender der Daten führt. Im Falle eines Clients ist der Sender natürlich *immer* der Server.

Resümee

Jetzt ist es an der Zeit für das Popcorn. Also Tüten auf, und los. Wir haben nun für unsere *ZFXEngine* eine vollwertige Netzwerk-DLL im Repertoire. Sie mag zwar noch einige Feinheiten missen lassen und keine Lobbies oder Ähnliches bieten. Aber das sind alles Dinge, die man nachträglich einpflegen kann, so man sie benötigt. Für ein normales Multiplayer-Spiel reicht die Funktionalität dieser DLL aber allemal aus, wie wir im Verlauf dieses Buches noch sehen werden.

*Die Mutter aller
Netzwerk-
Applikationen*

11.5 Demo-Applikation zur Anwendung der DLL

Bevor wir nachher unsere Netzwerk-DLL im Life-Fire-Test direkt in einem Computerspiel implementieren, macht es an dieser Stelle Sinn, eine kleine Demo-Applikation zu entwickeln, die ganz isoliert nur die Anwendung der Netzwerk-DLL zeigt. So könnt ihr am besten die Eleganz des Entwurfs

erkennen, insbesondere wie wenige Funktionsaufrufe notwendig sind, um das Netzwerk vollwertig zu betreiben. Hier greift wieder das KISS-Prinzip: Keep it simple, stupid. Wer hier gut aufpasst und auch versteht, wie man die Netzwerk-DLL verwendet, der wird auch später keine Probleme damit haben, die Netzwerk-DLL in weitaus komplexeren Projekten unterzubringen. Denn wie sagte ich schon fast tausendmal in diesem Kapitel: *Es ist doch nur das Senden und Empfangen von Daten.*

Chatten über das Netzwerk

Der erste Teil der Demo-Applikation ist natürlich ein Chat. Wer hätte das gedacht. Durch einen Chat kann man sehr gut alle Prinzipien einer Netzwerk-Applikation erklären und zeigen, denn ein Chat erfüllt die Mindest- und Maximalanforderung an eine session-basierte Applikation: Es werden Daten versandt und empfangen. Bei diesen Daten handelt es sich um Text-strings, also um `char`-Arrays, die von einem Chatter an alle anderen Chatter geschoben werden.

*Unser guter alter
Demo-
Arbeitsbereich*

Als Ausgangsbasis verwenden wir denselben Code wie auch schon für die Demo-Applikationen der drei anderen DLL-Komponenten, daher werde ich hier nicht alles zeigen. Die Initialisierung unserer DLL-Komponente läuft wie gewohnt ab. Das bedeutet, wir erzeugen erst ein Objekt der statischen Bibliothek `ZFXNetwork` und ziehen aus diesem ein Objekt von Typ `ZFXNetwork-Device` heraus. Wollten wir eine Applikation mit mehreren Clients oder einem Server und einem oder mehreren Clients erstellen, müssten wir dies mit einem separaten `ZFXNetwork`-Objekt für jeden Client und den Server tun.



```
HRESULT ProgramStartup(ZFXNETMODE nmd) {
    g_pNetwork = new ZFXNetwork(g_hInst);

    if (FAILED( g_pNetwork->CreateDevice() ))
        return E_FAIL;

    g_pDevice = g_pNetwork->GetDevice();

    if (FAILED( g_pDevice->Init(g_hWnd, nmd, g_nPort,
                               g_chIP, 5000, true) ))
        return E_FAIL;
    return ZFX_OK;
} // ProgramStartup
```

*Parameter der
Init-Funktion*

In diesem Funktionsaufruf ist `g_hWnd` natürlich das Handle auf das Hauptfenster der Applikation. Es liegt in Form einer globalen Variablen vor. Die beiden anderen globalen Variablen stehen für die Port-Nummer und den String mit der IP-Adresse. Beide Angaben sind für die Erstellung eines Clients notwendig. Um einen Server zu erzeugen, muss man nur die Port-Nummer angeben.

Das DOS-Programm `ipconfig.exe` liefert einem die IP-Adresse des eigenen Computers, um einen Client zu diesem Computer zu konnekten. Dazu muss entsprechend vorher ein Server auf diesem Computer gestartet worden sein. Der Client kann aber auch auf demselben Computer als zusätzliche Instanz der Applikation laufen. Als Portnummer kann man beispielsweise 6000 verwenden.

;-)
TIPP

Den Wert für die Port-Nummer und die IP-Adresse erhält das Programm aus einem Dialog, der nach dem Erzeugen des Fensters und vor dem Aufrufen der Funktion `ProgrammStartup()` eingeblendet wird. Abbildung 11.4 zeigt einen Screenshot dieses Dialogs. Sobald man eine Port-Nummer eingegeben hat, wird der Button zur Erstellung eines Servers freigegeben. Ist auch eine valide IP-Adresse vollständig angegeben, kann man auch einen Client starten.

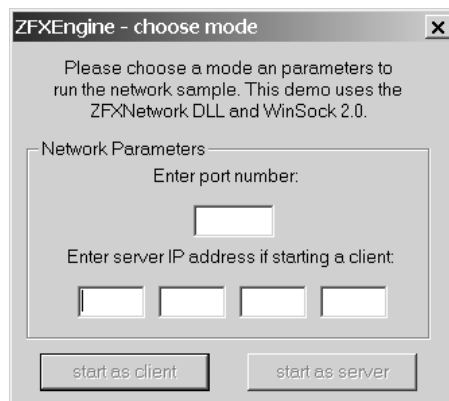


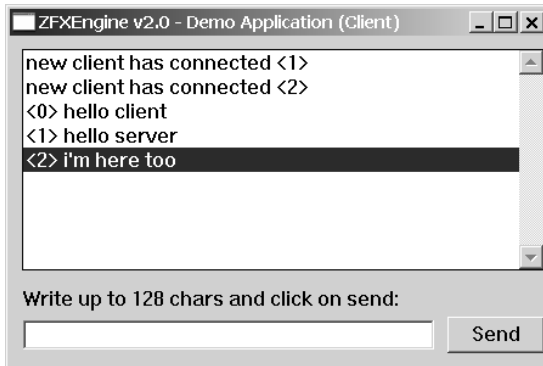
Abbildung 11.4:
Eingabe-Dialog des
Chat-Programms

Diesen Dialog werde ich nicht weiter beschreiben, da es sich um einen ganz normalen Dialog unter Verwendung der WinAPI handelt, der noch nicht mal sehr komplex ist. Der Quellcode auf der CD-ROM zu diesem Buch sollte zum Verständnis ausreichen. Wir konzentrieren uns lieber auf den Code, der für das Ausführen des Chats von Bedeutung ist. Abbildung 11.5 zeigt einen Screenshot des Chats, den man sieht, wenn man sein Programm gestartet hat.

Um diesen Chat zu realisieren, benötigen wir nach der Initialisierung des Netzwerk-Devices eigentlich nur zwei Dinge im Programm: Wir brauchen eine geänderte Windows-Prozedur, und wir müssen die Hauptschleife des Programms ein wenig frisieren. Beginnen wir mit der Windows-Prozedur: Wir reagieren hier zusätzlich auf die Nachrichten `WM_ZFXSERVER` und `WM_ZFXCLIENT`. In beiden Fällen leiten wir die Nachricht einfach an die Nachrichten-Prozedur des `ZFXNetworkDevice`-Objekts weiter. Dort werden die entsprechenden Events wie oben gezeigt bearbeitet.

**Funktionsweise
des Chats, 1:**
Senden von Chat-
Nachrichten

Abbildung 11.5:
Der Chat ist im
vollen Gange



Die Applikation muss sich in ihrer eigenen Prozedur nur um das Versenden der Nachrichten kümmern. Dies findet in der Prozedur statt, weil der Chat durch einen Klick auf den Button SEND einen String verschicken soll. Dieser Button hat das Handle `IDC_hBtn`, und dieses fragen wir über die Nachricht `WM_COMMAND` ab. Wurde der Button gedrückt, dann holen wir zuerst den Text aus dem Eingabefeld mit dem Handle `hText`. Diesen Text verpacken wir in ein `ZFXPACKAGE`-Objekt.

Unterschiede zwischen Server und Client

Nun kommen wir zu den ersten Unterschieden in der Implementierung eines Servers und eines Clients: Der Server ist eigentlich nicht dazu gedacht, selbst aktiv am Chat beteiligt zu werden, jedenfalls nicht in der grundsätzlichen Idee einer Client-Server-Architektur. Hier arbeitet der Server ja nur die Eingaben der Clients ab und schickt die Auswertung der Eingaben zurück. Da das bei einem Chat aber ziemlich blöd wäre, schreiben wir hier direkt die Nachricht, die der Benutzer in das Textfenster des Chats beim Server eingegeben hat, in die Listbox mit den ganzen Chat-Nachrichten (diese hat das Handle `hLbx`). Dann sendet der Server diesen String, der ja bereits verpackt ist, an alle Clients. Ist die Applikation hingegen ein Client, dann wird das Paket direkt an den Server gesendet, aber wir schreiben den String explizit *nicht* in die eigene Listbox mit den Chat-Nachrichten.



```
case WM_ZFXSERVER: case WM_ZFXCLIENT: {
    g_pDevice->MsgProc(wParam, lParam);
} break;

case WM_COMMAND: {
    switch (LOWORD(wParam)) {
        case IDC_hBtn: {
            ZFXPACKAGE    msgPkg;
            char            buffer[128], buf[200];
            msgPkg.pData = buf;

            GetWindowText(hText, buffer, 128);
            SetWindowText(hText, "");
```

```

if (strcmp(buffer, "exit")==0) {
    g_bDone = true;
    PostMessage(hWnd, WM_CLOSE, 0, 0);
}
else {
    sprintf((char*)msgPkg.pData, "<%d> %s",
        g_ID, buffer);
    // +1 für terminierenden 0-Character
    msgPkg.nType = 3;
    msgPkg.nSender = g_ID;
    msgPkg.nLength = strlen((char*)msgPkg.pData)+1;

    // SERVER-CODE
    if (g_nmd==NMD_SERVER) {
        SendMessage(hLbx, LB_ADDSTRING, 0,
            (LPARAM)msgPkg.pData);
        int n = SendMessage(hLbx, LB_GETCOUNT, 0, 0);
        SendMessage(hLbx, LB_SETCURSEL, (WPARAM)n-1, 0);
        g_pDevice->SendToClients(&msgPkg);
    }
    // CLIENT-CODE
    else g_pDevice->SendToServer(&msgPkg);
}
return 0;
} break;
} // switch [CMD]
return 0;
} break; // WM_COMMAND

```

Damit haben wir den Teil erledigt, in dem das Senden der Nachrichten behandelt wird. Der Server bekommt hier wie gezeigt eine Extrawurst, denn normalerweise ist ein Server gar nicht aktiv in das Geschehen eingebunden, das wir am Bildschirm sehen. Den if-Fall des Servers hätten wir also getrost weglassen können. Die Clients hätten dann immer noch vollwertig miteinander chatten können. Der Server hätte nur die entsprechenden Nachrichten nicht anzeigen und auch keine eigenen Nachrichten senden können. Doch nun müssen die Chat-Nachrichten ja auch empfangen werden. Die Nachrichten-Prozedur der ZFXNetworkDevice-Klasse sorgt dafür, dass die eigene Inbox mit den empfangenen Nachrichten betankt wird. Unsere Applikation muss also lediglich in jedem Frame prüfen, ob in der Inbox Nachrichten angekommen sind. Das machen wir natürlich in der Hauptschleife:

```

ZFXPACKAGE    msgPkg;
char          buffer[5000], temp[300];
msgPkg.pData = buffer;

// HAUPTSCHLEIFE
while (!g_bDone) {
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

```

*Funktionsweise
des Chats, 2:
Empfangen von
Chat-Nachrichten*



```

    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

while ( g_pDevice->IsPkgWaiting() ) {

    g_pDevice->GetNextPkg(&msgPkg);

    // Client erhält seine ID vom Server
    if (msgPkg.nType == 0)
        g_ID = *((UINT*)msgPkg.pData);

    // Ein neuer Client ist im Netzwerk
    else if (msgPkg.nType == 1) {
        sprintf(temp, "new client has connected
        <%d>", *((UINT*)msgPkg.pData));
        SendMessage(hLbx, LB_ADDSTRING, 0, (LPARAM)temp);
    }

    // ein Client hat das Netzwerk verlassen
    else if (msgPkg.nType == 2) {
        sprintf(temp, "client <%d> has
        disconnected", *((UINT*)msgPkg.pData));
        SendMessage(hLbx, LB_ADDSTRING, 0, (LPARAM)temp);
    }

    // sonst ist es eine Chat-Nachricht
    else {
        SendMessage(hLbx, LB_ADDSTRING, 0,
            (LPARAM)msgPkg.pData);

        if (g_nmd==NMD_SERVER)
            g_pDevice->SendToClients(&msgPkg);
    }

    int n = SendMessage(hLbx, LB_GETCOUNT, 0, 0);
    SendMessage(hLbx, LB_SETCURSEL, (LPARAM)n-1, 0);
} // while msg incoming
} // main loop

```

*Pakete
empfangen und
angemessen
bearbeiten*

Der Ablauf sollte klar sein: In jedem Durchlauf der Hauptschleife starten wir eine weitere Schleife, die so lange läuft, wie noch Nachrichten in der Inbox des ZFXNetworkDevice-Objekts sind. Nachdem wir eine solche Nachricht aus der Inbox gezogen haben, folgt eine Fallunterscheidung. Bisher kennen wir nur vier Arten von verschiedenen Nachrichten, daher handeln wir sie alle gleich in der Hauptschleife ab. Handelt es sich bei der Nachricht um die Zusendung einer ID vom Server an einen neu angemeldeten Client, so speichert dieser seine ID ab. Hat sich ein neuer Client am Netzwerk angemeldet oder hat sich ein Client aus dem Netzwerk abgemeldet, dann geben wir entsprechend formatierte Meldungen in die Listbox der Applikation

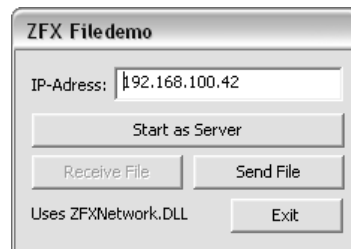
zusammen mit der ID des betroffenen Clients aus. Treffen diese drei Fälle nicht zu, dann handelt es sich um eine ganz normale Chat-Nachricht. Diese wird in der Listbox aufgeführt.

Handelt es sich bei unserer Instanz der Applikation um einen Server, so müssen wir die Nachricht nicht nur in der eigenen Listbox ausgeben, sondern auch an alle Clients schicken. Wir erinnern uns dunkel, dass der Client beim Senden einer Chat-Nachricht diese Nachricht explizit nicht selbst in seine Listbox schiebt. Wir halten uns hier strikt an das Client-Server Prinzip, dass nur mit Daten gearbeitet, wird die der Server an die Clients gesendet hat.

Was soll ich sagen? Das war der Chat. Mehr ist nicht nötig, um mit bis zu 256 Clients über TCP/IP zu chatten.

Dateien versenden

Der zweite Teil der Demo-Applikation zu unserer Netzwerk-DLL regelt das Versenden von Dateien zwischen einem Client und einem Server. Unsere kleine Applikation kann nach dem Start als Server aktiviert werden, oder sie kann durch die Angabe einer gültigen IP und das Auswählen einer Datei als Client gestartet werden. Steht die Verbindung zwischen Client und Server, können diese Daten untereinander austauschen. Unser Dialog, der in Abbildung 11.6 dargestellt wird, ist sehr pragmatisch entworfen.



*Der Server
bringt's!*

Abbildung 11.6:
Dialog der Demo-
Applikation zum
Versenden von
Dateien

Wir nutzen hier ein ähnliches Framework wie das aus den vorangegangenen Demos für unsere Applikation. Der Start unserer Applikation erfolgt in der `Init()`-Prozedur, wo wir grundlegend unsere gesamte Applikation initialisieren und wie gewohnt die DLL-Komponenten instanziiieren.

```
HRESULT Init( void ) {
    g_ofn.lStructSize    = sizeof(OPENFILENAME);
    g_ofn.hwndOwner      = g_hWnd;
    g_ofn.lpstrTitle     = g_szAppName;
    g_ofn.hInstance      = g_hInst;
    g_ofn.lpstrFilter     = TEXT(
        "All Files(*.*)\0*.*\0\0");
```



```

g_ofn.lpstrCustomFilter = NULL;
g_ofn.nMaxCustFilter    = 0;
g_ofn.nFilterIndex      = 0;
g_ofn.nMaxFile          = MAX_PATH;
g_ofn.nMaxFileTitle     = MAX_PATH;
g_ofn.lpstrInitialDir   = NULL;
g_ofn.nFileOffset       = 0;
g_ofn.nFileExtension    = 0;
g_ofn.lpstrDefExt       = TEXT( "*.*" );
g_ofn.lCustData          = 0L;
g_ofn.lpfnHook          = NULL;
g_ofn.lpTemplateName   = NULL;

g_pNetwork = new ZFXNetwork( g_hInst );
if( FAILED( g_pNetwork->CreateDevice() ) )
    return E_FAIL;

g_pDevice = g_pNetwork->GetDevice();
if( g_pDevice == NULL ) return E_FAIL;

g_sData.nSender = 0;
g_sData.nType   = 0;
g_sData.pData   = new BYTE[ MAX_PACKAGE_SIZE ];
ZeroMemory( g_sData.pData, MAX_PACKAGE_SIZE );
return S_OK;
} // Init

```

WinAPI-OpenFile- Name-Dialog

Um uns das Auswählen der zu übertragenden Dateien sowie das Speichern der übertragenen Dateien zu erleichtern, nutzen wir den so genannten Open-FileName-Dialog. Das ist ein Standarddialog, den uns Windows über die WinAPI zur Verfügung stellt. Dieser will aber noch reichlich von uns eingestellt werden, bevor wir ihn mit ein paar simplen Aufrufen nutzen. Wie gewohnt verwenden wir hier wieder globale Variablen, in denen unser Window-Handle (*g_hWnd*), unsere Window-Instanz (*g_hInst*) und auch unser Applikationsname (*g_szAppName*) gespeichert ist. Außerdem müssen wir noch kenntlich machen, welche Art von Dateien angezeigt werden sollen. Dazu übergeben wir mit dem Makro `TEXT` den String »All Files (*.*)\0*.*\0\0«. Wenn wir hier z.B. nur ausführbare Dateien anzeigen lassen wollen, also z.B. nur *.exe, dann müsste der String beispielsweise wie folgt aussehen: »Executable (*.EXE)\0*.EXE\0\0«. Ein solcher Filter wird von dem Dialog dann automatisch genutzt.

Beliebige Grenze setzen!

Als Nächstes holen wir uns wie gewohnt die Netzwerkkomponenten aus der Netzwerk-DLL. Danach definieren wir noch schnell unser Standardpaket und initialisieren den dazugehörigen Speicher. Das `#define MAX_PACKAGE_SIZE` ist in der Headerdatei definiert und steht auf 65536. Daher dürfen unsere Dateien momentan nicht größer als 64 Kbyte sein.

In der Dialog-Prozedur `DLGProc(...)` werten wir all unsere Nachrichten aus, die für unseren Dialog bestimmt sind. Die wichtigen Buttons zum Senden oder Empfangen von Daten werden hier überprüft und weiter bearbeitet. Auch wird hier wie in der Chat-Demo-Applikation auf unsere beiden speziellen Nachrichten `WM_ZFXCLIENT` und `WM_ZFXSERVER` gewartet und ggf. darauf reagiert.

```
// Befehle für uns?
case WM_COMMAND:
    switch( LOWORD( wParam ))
    {
        // Session beenden?
        case IDCANCEL:
            Destroy();
            PostQuitMessage( 0 );
            return 0;
        // File versenden?
        case IDC_SENDFILE:  FileSend();    break;
        // File empfangen?
        case IDC_RECEIVEFILE: FileReceive(); break;
        // Als Server starten?
        case IDC_BESERVER:
            g_nmd = NMD_SERVER;
            g_iID = 1;
            NetworkInit();
            break;
    } // switch
} // WM_COMMAND
```

Am Anfang wird uns die Wahl gelassen, ob wir als Server oder als Client starten wollen. Daher initialisieren wir nach dem Anklicken des Buttons `START AS SERVER` mit der ID `IDC_BESERVER` unser Netzwerk als Server. Dazu müssen wir nur die globale Variable (`g_nmd`) auf `NMD_SERVER` und unser `g_iID` auf 1 setzen und mit der Prozedur `NetworkInit()` unser Netzwerk entgültig initialisieren. Ein Client wird später initialisiert, nämlich dann, wenn wir ein File senden wollen. Dazu später mehr. Durch die beiden Buttons `SENDFILE` und `RECEIVEFILE` werden die dazugehörigen Prozeduren aufgerufen. Wobei `RECEIVEFILE` immer nur dann aktiv ist, wenn auch wirklich ein File vorhanden ist.

```
// Netzwerk-Aktivität
case WM_ZFXCLIENT:
case WM_ZFXSERVER:
    // DLL updaten
    g_pDevice->MsgProc( wParam, lParam );
    // Demo updaten
    NetworkReceive();
    // break
    break;
} // Messages
```



Die beiden Nachrichten, die von Windows an unseren Dialog gesendet werden, wenn etwas für uns auf der Netzwerkleitung liegt, werden hier abgefangen und in unserem Netzwerk-Device bearbeitet. Danach geht es direkt in die Prozedur unseres Netzwerk-Devices, in der wir prüfen, ob Daten tatsächlich für uns angekommen sind. Dazu später mehr.

Schauen wir uns an, wie das Netzwerk für uns initialisiert wird. Wir wissen ja, dass wir dazu vorher ein paar Parameter gesetzt haben.



```
HRESULT NetworkInit( void ) {
    // Netzwerk online?
    if( !g_bNetOK ) {
        // Als Client starten?
        if( g_nmd == NMD_CLIENT ) {
            // IP_Adresse holen
            GetWindowText(GetDlgItem(g_hWnd, IDC_IPADDRESS),
                           g_cIPADDRESS, 16);
        }
        else {
            // Zeige den Zustand an
            SetDlgItemText(g_hWnd, IDC_IPADDRESS, " ...waiting
                for files");
        }

        // Starte das Device
        if( FAILED( g_pDevice->Init(g_hWnd, g_nmd,
                                    g_iPort, g_cIPADDRESS,
                                    MAX_PACKAGE_SIZE,
                                    true ) ) ) {
            // Netzwerk-Init fehlgeschlagen
            MessageBox(g_hWnd, "Network init failed",
                "Systemrequest", MB_ICONERROR );
            return E_FAIL;
        }
        g_bNetOK = true;

        // Knopf deaktivieren
        EnableWindow(GetDlgItem(g_hWnd, IDC_BESERVER),
            false );

        // Feld deaktivieren
        EnableWindow(GetDlgItem(g_hWnd, IDC_IPADDRESS),
            false);
    }
    return S_OK;
}
```

Es fängt alles damit an, dass wir überprüfen, was wir eigentlich sind. Sollten wir als Client gestartet sein, dann holen wir die Ziel-IP-Adresse aus dem Eingabefeld und speichern diese in der globalen Variable `g_cIPADDRESS`. Dabei holen wir uns auch nur wirklich 16 Zeichen. Also muss die IP-Adresse das Format 192.168.100.042 haben. Namen werden also nicht aufgelöst. Als Server schreiben wir einfach stumpf eine kleine Meldung in das Feld.

IP-Adresse

Da wir jetzt alles für die endgültige Initialisierung zusammen haben, starten wir das Netzwerk-Device. Der Port in `g_iPort` ist übrigens von mir in der Headerdatei auf 10042 gesetzt worden. Wer also eine Firewall hat, sollte diesen Port »schleifen«, wie man im Fachjargon dazu sagt. Bei Problemen mit der Initialisierung schicken wir eine freundliche Messagebox auf den Screen. In der Regel sollte das aber nicht fehlschlagen, daher setzen wir danach die globale Variable `g_bNetOK` auf `true`. Anschließend schalten wir den Button `START AS SERVER` und das Eingabefeld für die IP-Adresse aus. Somit erkennt der User, dass wir aktiv sind.

Konstanter Port

Als Server befinden wir uns jetzt schon voll aktiv in freudiger Erwartung eines Clients. Als Client hätten wir vorher eine Datei ausgewählt und würden versuchen, den Server mit der eingegebenen IP zu erreichen und die Datei zu übertragen. Schauen wir uns also als Erstes an, was auf der Client-seite passiert.

*Server versus
Client*

Wir haben also wie oben gezeigt eine IP-Adresse eingetragen und auf den Button `SENDFILE` geklickt. Jetzt wird die Prozedur `FileSend()` aufgerufen.

```
HRESULT FileSend( void ) {
    // Variablen Init
    FILE* pFile = NULL;

    // Einstellungen für den File-Dialog
    g_ofn.Flags      = OFN_HIDEREADONLY |
                     OFN_CREATEPROMPT;
    g_ofn.lpstrTitle = TEXT("Send a File");
    g_ofn.lpstrFile  = (PTSTR)&g_szFile;

    // Öffne den File-Dialog
    if( !GetOpenFileName( &g_ofn ) ) return S_OK;

    if( !g_bNetOK ) {
        // Starte das Netzwerk
        if( FAILED( NetworkInit() ) ) return E_FAIL;
    }

    /* wird fortgesetzt */
}
```



Als Erstes setzen wir in der Prozedur die notwendigen Parameter für unseren File-Dialog. Das sind die spezifischen Parameter für den OPEN-Dialog, also den Dialog zum Öffnen von Dateien. Der Dialog wird dann einfach mit

*Auswählen der zu
versendenden
Datei*

der komplett betankten globalen Variable `g_ofn` gestartet. Erst wenn der Anwender in diesem Dialog eine Datei ausgewählt hat, geht es in unserer Prozedur weiter. Im Falle des Abbruchs springen wir mit `S_OK` aus der Funktion raus, und fertig. Hat der Anwender eine Datei ausgewählt, dann starten wir das Netzwerk, wenn es nicht schon gestartet ist. Auch hier wird geprüft, ob alles glatt lief. Sollte wider Erwarten ein Fehler aufgetreten sein, gehen wir mit `E_FAIL` aus der Funktion heraus.

Jetzt haben wir eine Datei und eine Verbindung zum Ziel, was liegt also näher, als diese zu laden und dann zu übertragen? Nichts!



```
/* fortgesetzt */
// jetzt haben wir die gewünschte Datei
pFile = fopen( g_szFile, "rb" );
if( pFile == NULL ) return E_FAIL;

// Dateigröße
g_sData.nLength = _filelength( _fileno( pFile ) );

// korrekte Dateigröße
if( g_sData.nLength > MAX_PACKAGE_SIZE )
{
    // nein, ist zu groß für uns
    fclose( pFile );
    MessageBox( g_hWnd, "Unexpected filesize",
                "Systemrequest", MB_ICONERROR );
    return E_FAIL;
}
// lösche Buffer
ZeroMemory( g_sData.pData, MAX_PACKAGE_SIZE );

// lies die Datei in den Buffer
fread( g_sData.pData, sizeof(BYTE), g_sData.nLength,
        pFile );

// schließe die Datei
fclose( pFile );
/* wird fortgesetzt */
```

Arbeiten mit File- Funktionen

Den Dateinamen inklusive des kompletten Pfads finden wir in der globalen Variable `g_szFile`. Dieses File öffnen wir auch gleich im »rb-« also im read-binary-Modus. Auch hier führen wir wieder eine Fehlerbehandlung durch die Abfrage durch, ob wir einen Filepointer zurückbekommen haben. Als Nächstes wollen wir wissen, wie viel Speicher im Datenpaket belegt wird. Dazu greifen wir ein wenig in die Kiste und finden dort die Methode `_filelength()`, die aber eine File-Nummer und nicht ein Filehandle braucht. Aber dazu finden wir noch den Befehl `_fileno()`, der aus dem Handle eine File-Nummer macht. Jetzt wird uns netterweise die korrekte Dateigröße

übergeben, die wir gleich als Länge für unser Datenpaket in `g_sData.nLength` speichern. Damit wir nicht in unsere definierten Schranken laufen, prüfen wir, ob die Dateigröße zulässig ist. Wenn das der Fall ist, löschen wir den gesamten Paketspeicher und laden dann die Datei dort hinein.

Danach brauchen wir das Filehandle nicht mehr und schließen die Datei wieder. Die Daten sind jetzt sauber im Paket verstaut, also versenden wir es an unseren Empfänger.

```
/* fortgesetzt */
g_sData.nSender    = g_iID;
g_sData.nType      = 0;
// Client oder Sever?
if( g_nmd == NMD_CLIENT ) {
    // Paket senden
    if (FAILED(g_pDevice->SendToServer(&g_sData))) {
        MessageBox( g_hWnd, "Sending data failed",
                    "Systemrequest", MB_ICONERROR );
        return E_FAIL;
    }
}
else {
    if (FAILED(g_pDevice->SendToClients(&g_sData))) {
        MessageBox( g_hWnd, "Sending data failed",
                    "Systemrequest", MB_ICONERROR );
        return E_FAIL;
    }
}
return S_OK;
} // FileSend
```



Das Datenpaket bekommt noch unsere ID und den entsprechenden Nachrichtentyp verpasst. Dann prüfen wir, ob wir ein Client oder ein Server sind. Für die Sendemethode in der Netzwerk-DLL ist das relevant. Dort werden die Daten unterschiedlich bearbeitet. Ja, wie gesagt: prüfen, senden und sich freuen. Sollte etwas fehlschlagen, melden wir das höflich.

An dieser Stelle verlassen wir die gesamte Prozedur mit `S_OK`. Was soll ich sagen – das war schon die komplexeste Prozedur in unserem Programm, der Rest wird ziemlich einfach werden. Also, let's go...

Unser Dialog reagiert ja freundlicherweise aufgrund der abgefangenen `WM_ZFXCLIENT`- und `WM_ZFXSERVER`-Nachricht. Das Netzwerk-Device wurde von uns schon in der Dialog-Prozedur `DLGProc()` angestoßen, und danach wollten wir ja prüfen, ob ein Paket für uns dabei ist. Das tun wir jetzt mit der Prozedur `NetworkReceive()`:

*Empfangen von
Dateien*



```

HRESULT NetworkReceive( void ) {
    // Check
    if( !g_bNetOK ) return E_FAIL;
    // wartet ein Package auf uns?
    while( g_pDevice->IsPkgWaiting() ) {
        if ( FAILED(g_pDevice->GetNextPkg(&g_sData)) ) {
            MessageBox( g_hWnd, "GetNextPkg() failed",
                "Systemrequest", MB_ICONERROR );
            return E_FAIL;
        }
        else {
            // wir haben das Package empfangen
            if( (g_sData.nType == 0) && ( g_iID !=
                g_sData.nSender ) ) {
                // Button zum Speichern aktivieren
                EnableWindow( GetDlgItem( g_hWnd,
                    IDC_RECEIVEFILE ), true );
            }
        }
    } // while
    return S_OK;
} // NetworkReceive

```

*Abfragen des
Postfachs in der
Netzwerk-DLL*

Sollte wider Erwarten das Netzwerk nicht aktiv sein, springen wir mit einem Fehler raus, ansonsten warten wir mit der Methode `IsPkgWaiting()` auf unser nächstes Paket in unserem Netzwerk-Device. Ist eins angekommen, holen wir uns das Paket mit der Methode `GetNextPkg()` ab, wobei wir unser Datenpaket als Referenz übergeben. In dieses als Referenz angegebene Paket wird das eigentliche Paket geschrieben, und wir können es später auspacken – fast wie Weihnachten J. Schlägt das fehl, so melden wir das unserem ehrbaren Anwender. Lief alles glatt, so prüfen wir, ob das auch wirklich ein echtes Paket ist; der Pakettyp muss hier 0 sein. Ist dem so, dann schalten wir den Button `RECEIVEFILE` ein, und der Anwender erkennt, dass eine Datei angekommen ist, die er jetzt empfangen und abspeichern kann.

`ReceiveFile()` ist unsere Prozedur, in der wir das zweite Mal unseren `Open-FileName-Dialog` nutzen. Sie öffnet für den Anwender einen freundlichen Dialog, in dem er den Speicherort und den Namen der Datei angeben kann. Aber schauen wir uns die Prozedur erst mal an.



```

HRESULT FileReceive( void ) {
    // Variablen initialisieren
    FILE* pFile = NULL;
    // Einstellungen für den File-Dialog
    g_ofn.Flags      = OFN_OVERWRITEPROMPT;
    g_ofn.lpstrTitle = TEXT("Receive a File");
    g_ofn.lpstrFile  = (PTSTR)&g_szFile;

```

```
// auf Daten prüfen
if( g_sData.nLength == 0 ) return S_OK;

// öffne den File-Dialog
if( !GetOpenFileName( &g_ofn ) ) return S_OK;

// jetzt haben wir den gewünschten Pfad
pFile = fopen( g_szFile, "wb" );
if( pFile == NULL )
    return E_FAIL;

// schreibe die Datei in den Datenstream
fwrite( g_sData.pData, sizeof(BYTE), g_sData.nLength,
        pFile );

fclose( pFile );

// zeige, dass wir die Datei gespeichert haben
EnableWindow( GetDlgItem( g_hWnd, IDC_RECEIVEFILE ),
              false );
return S_OK;
} // FileReceive
```

Wie schon in der `FileSend()`-Prozedur werden als Erstes die speziellen Parameter für den `OpenFileName`-Dialog gesetzt. Er wird geöffnet, nachdem wir geprüft haben, ob überhaupt ein Paket für uns da ist. Der Anwender darf dann den Speicherort auswählen und den Namen setzen. Dieser steht wie gewohnt, wenn der Dialog nicht abgebrochen wurde, in der globalen Variable `g_szFile`. Die Daten öffnen wir dann mit dem Flag »wb«, also *write binary*. Auch hier prüfen wir, ob ein Filehandle erstellt wurde. Danach schreiben wir einfach den gesamten Paketinhalt, der sich in dem Datenpaket `g_sData.pData` mit der Länge `g_sData.nLength` befindet. Als Nächstes wird die Datei geschlossen und der Button `RECEIVEFILE` deaktiviert.

*OpenFileName-
Objekt einstellen*

Ob ihr es glaubt oder nicht, das war schon alles. Warum ist das bloß alles so einfach?

Ich will euch noch ein paar Gedanken mit auf den Weg geben, was ihr noch ändern und implementieren könnt. Die Paketgröße ist nicht umsonst auf 64 Kbyte gesetzt. Das ist zufällig die Größe, die unsere Netzwerk-DLL als ein Paket versenden kann. Um größere Dateien zu versenden, muss vorher der Datenstream in handliche 64-Kbyte-Stücke zerlegt und beim Empfangen wieder zusammengesetzt werden.

*Erweiterungs-
potenzial*

Außerdem könntet ihr einen eigenen Pakettyp erstellen, der den Namen der Datei beinhaltet, so dass der Anwender den Originalnamen schon in dem File-Dialog beim Speichern vorgeschlagen bekommt.

Dann könntet ihr auch mehrere Dateien zulassen, die übertragen werden können. Momentan überschreibt jedes neue Datenpaket das alte. Daher müsstet ihr hier eine kleine verkettete Liste implementieren, mit der ihr die Daten aus den Paketen verwaltet. Viel Spaß.

11.6 Ein Blick zurück, zwei Schritt nach vorn

Im Schweinsgalopp

Ist es nicht immer wieder erstaunlich, wie einfach viele Dinge doch umzusetzen sind? Wer vor dem Lesen dieses Kapitels einen großen Respekt vor der Programmierung für Netzwerke gehabt hat, der möge nun die Hand heben. Aha, ich sehe. Gut. Und nun heben bitte nur noch diejenigen die Hand, die jetzt immer noch einen solchen Respekt davor haben. Nanu? Keiner mehr? :-)

In diesem Kapitel haben wir tatsächlich im Schweinsgalopp die Funktionalität zur Kommunikation über Netzwerke in unsere Engine eingebaut. Und so schwer war es wirklich nicht. Ein Netzwerkspiel macht doch nichts anderes, als einfach ein paar Daten über ein Netzwerk zu senden. Und genau das können wir nun tun. Wir haben in diesem Kapitel zwar nur zwei kleinere Beispiele unter Verwendung der WinAPI gesehen, aber das zeigt auch, dass man die Netzwerk-Komponente unserer Engine auch verwenden kann, ohne deren Grafik-, Input- oder Audio-Komponente mit in seinem Programm haben zu müssen.

Weiter geht's

Wenn wir noch die folgenden Kapitel durchhalten, werden wir schon bald sehen, wie wir ein richtiges 3D-Spiel mit Netzwerkfunktionalität programmieren können. In diesem Kapitel haben wir uns mit dem Netzwerk-Part auch gleichzeitig der letzten API-unabhängigen Komponente unserer Engine gestellt. Im nächsten Kapitel geht es noch um ein paar kleine Helferlein, und dann tauchen wir direkt ab in die schwarze Magie der Spieleprogrammierung. Auf geht's.

12 Timing und Bewegung in der ZFXEngine

*»Der Topf von Eisen will mit einer silbernen Zange
gern aus der Glut gehoben sein, um selbst einen Topf
von Silber sich zu dünken..« (Schiller, Nathan der Weise)*

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Timing und bewegungsabhängige Bewegungen
- ➔ Bewegung des Spielers in der virtuellen Welt
- ➔ Transformation von Objekten mit sechs Freiheitsgraden
- ➔ Implementierung einer Basisklasse für verschiedene Kamera-Modi
- ➔ Programmierung einer freien Kamera und einer FPS-Kamera
- ➔ Entwicklung einer statischen Zusatz-Bibliothek für die ZFXEngine

12.1 Hilfsbibliothek ZFXGeneral

In diesem Kapitel werde ich mich extra kurz fassen, denn wir haben noch so einiges vor, was mir wichtiger ist. An dieser Stelle geht es darum, eine kleiner Helper-Bibliothek für die *ZFXEngine* zu programmieren. Diese soll zwei Aufgaben erledigen, die zwar von jeder Grafik-Applikation benötigt werden, die aber dennoch kein Teil des Renderers sind. Ich spreche hier natürlich, der Titel des Kapitels lässt das bereits vermuten, von der Kameraführung und dem Timing. Unsere eiserne Engine können wir daher mit Silberglanz versehen, indem wir ihr diese Bibliothek mit auf den Weg geben.

*Von Eisen zu
Silber?*

Diese Bibliothek nennen wir `ZFXGeneral.lib`. Sie kann vollkommen unabhängig von anderen Teilen der Engine verwendet werden. Sie benötigt einzig und allein noch ihre Schwester aus der ZFXEngine, nämlich die andere statische Bibliothek, die wir schon am Anfang des Buches implementiert haben. Ich spreche natürlich von der Mathe-Bibliothek `ZFX3D.lib`. Mit Hilfe dieser Bibliothek wird es dem Anwender dann möglich sein, einen Frame-Rate-Counter und einen Timer für die vergangene Zeit eines Frames (engl. *elapsed time*) zu integrieren und eine bewegliche Kamera in der virtuellen

*Statische
Bibliothek*

Welt zu platzieren. Zum Timing gibt es dabei nicht viel zu sagen, und eine solche Klasse zu implementieren ist auch nicht sonderlich schwer. Daher sprechen wir zunächst über verschiedene Kamera-Modi.

12.2 Verschiedene Kamera-Modi

*Ton ... Kamera ...
Action!*

Dem einen oder anderen kommt der Ausdruck »Kamera« im Kontext der Grafik-Programmierung vielleicht ein wenig komisch vor. Daher bringen wir es hier auf den Punkt: Als »Kamera« bezeichnet man eine Position mit einer zugehörigen Orientierung in der virtuellen Welt. Bedeutsam wird dieses Konzept insbesondere dann, wenn man die Szene aus der Perspektive einer Kamera betrachtet. Dann nämlich sieht man die Szene am Bildschirm quasi wie durch das Objektiv einer Kamera. Wir haben also implizit schon eine Kamera in unseren bisherigen Programmen, nur sind diese rein statisch. Im Folgenden sind einige Kamera-Modi aufgelistet, die neben einer solchen statischen Kamera existieren und die für den Spieler verwendet werden können.

Freie Kamera

*Jenseits der
physikalischen
Gesetze*

Als vollkommen freie Kamera bezeichnet man eine Kamera mit sechs Freiheitsgraden. Eine solche Kamera kann im 3D-Raum auf allen drei Achsen rotiert und auf allen drei Achsen verschoben werden – daher auch die sechs Freiheitsgrade. Eine solche Kamera kennt keine physikalischen Gesetze, wie beispielsweise eine beschränkte Fähigkeit, den Blick in eine Richtung zu drehen oder nur beschränkt um die Z-Achse rollen zu können. Eine solche freie Kamera ist also allenfalls in der Schwerelosigkeit recht interessant, wie etwa in Weltraum-Simulatoren.

Allerdings darf man auch den Debugger nicht vergessen, denn wenn man seine Level als Programmierer oder Level-Designer einmal austesten möchte, ist es oftmals hilfreich, eine vollkommen freie Kamera zur Verfügung zu haben. Mit dieser kann man dann durch seine Levels schweben und bis in die kleinste Ecke vorstoßen, die man sonst nicht erreichen würde. Man beachte allerdings, dass man bei einer solchen Kamera den Gimbal-Lock-Effekt berücksichtigen muss, der im vierten Kapitel bereits beschrieben wurde.

1st-Person-Kamera

Egozentriker

Eine der wohl berühmtesten Kamera-Perspektiven ist natürlich die 1st-Person-Kamera, wie sie in einem First-Person-Shooter (oder kurz FPS) verwendet wird. Daher stammt wohl auch der Name. Bei einer solchen Kamera kann der Spieler sich bzw. seinen virtuellen Character zwar frei auf der Y-

Achse, also der Welt-Hochachse drehen. Die Rotationen auf den anderen beiden Achsen sind jedoch i. d. R. eingeschränkt. So kann man den Kopf nur zu einem gewissen Grad heben und senken, und auch das seitliche Neigen des Spielers, sofern es überhaupt implementiert ist, um z. B. um die Ecken zu schießen, ist auf ein paar Grad Bewegungsfreiheit beschränkt.

3rd-Person Kamera

Spätestens seit sich *Lara Croft* mit Hilfe ihrer großen Ober- ... äh, -arme durch die *Tomb Raider*-Serie in die Herzen der Computerspieler und nun auch der Kinogänger gekämpft hat, sollte jeder wissen, was eine 3rd-Person-Kamera ist. Dabei sieht der Spieler das Spielgeschehen nicht durch die Augen der Figur, die er steuert, sondern betrachtet diese Figur aus einer Verfolgeransicht. Dabei muss die Kamera intelligent genug programmiert worden sein, um auch Positionen zu erkennen, wo diese direkte Verfolgeransicht nicht möglich ist, beispielsweise wenn die Spielfigur mit dem Rücken direkt an einer Wand steht. In so einem Fall darf die Kamera natürlich nicht in die Wand hineingehen, sondern muss eine seitliche Ausweichposition einnehmen und trotzdem auf die Spielfigur fixiert bleiben.

Lara lässt grüßen

Der klare Nachteil einer solchen Steuerung ist, dass der Spieler nur sehr indirekt Einfluss darauf hat, was er am Bildschirm sieht. Gerade in hektischen Kampfszenen, in denen er seine Spielfigur mit dem Rücken an eine Wand stellt oder in denen die Figur nur mit dem halben Kopf um eine Ecke schaut, wird man durch die automatisch positionierte Kamera selten genau das im Blick haben, was man eigentlich sehen möchte: nämlich das, was auch die Spielfigur sehen würde. Nicht zuletzt ist die plötzlich umschwingende oder zoomende Kamera mitten im Kampfgeschehen ein sehr verwirrendes Element. Der Reiz einer solchen 3rd-Person-Kamera ist aber natürlich, dass man seine eigene Spielfigur sehen kann. Auch ist es natürlich in gewissen Situationen möglich, sich so zu positionieren, dass man Dinge um eine Ecke sehen kann, die die Spielfigur durch ihre Augen noch nicht sehen kann. Zudem ist es gerüchteweise in Japan so, dass eine 1st-Person-Kamera die so genannte Motion-Sickness auslöst. Durch die Betrachtung eines Spiels aus der 1st-Person-Perspektive wird bei den japanischen Spielern also Übelkeit ausgelöst. Eine 3rd-Person-Kamera hat diesen negativen Effekt den Gerüchten zufolge nicht. Sony Japan testet beispielsweise ausländische Spiele zunächst auf dieses Phänomen und fordert dann entsprechende Nachbesserungen, wenn Probleme auftreten. Der Grund, warum 1st-Person-Shooter in Japan aber keinen so hohen Verbreitungsgrad haben wie im Rest der Welt, ist aber trotzdem eher darin zu sehen, dass dort Spielkonsolen wesentlich verbreiteter sind als der PC.

*Vorteile und
Nachteile der 3rd-
Person-View*

Fixe Kamera

Nichts rührt sich

Der letzte Typ Kamera, auf den wir hier kurz eingehen wollen, ist die so genannte fixe Kamera. Wie der Name schon sagt, ist die Position der Kamera dabei fixiert. So eine Kamera wird man in Spielen recht selten finden, allenfalls bei simplen 2D-Spielen wie *Tetris* oder bei Cut Scenes im Spielverlauf. Das »fix« muss sich dabei jedoch nicht notwendigerweise auf die Position der Kamera beziehen. Eine andere Alternative wäre, dass die Ausrichtung der Kamera fixiert ist, aber die Position trotzdem verändert werden kann. Klassische Beispiele für diese Art von Kamera sind natürlich die isometrischen Spiele wie beispielsweise *Diablo*.

12.3 Bewegung durch ZFXMovementController

*Mehr als nur eine
Kamera*

Wenn man über das Konzept der Kamera genauer nachdenkt, wird man schnell erkennen, dass sich dahinter mehr verbirgt als nur die Position und Ausrichtung, von der aus ein Betrachter die Szene sieht. Es geht dabei natürlich auch um die Bewegungen, die der Spieler als Betrachter ausführen kann. Wir werden die Kamerafunktionalität also in Form des so genannten ZFX-MovementControllers umsetzen.

Wozu einen Movement-Controller?

*Jedes bewegte
Objekt führt eine
Kamera bei sich*

Der Sinn des Movement-Controllers ist, dass jedes bewegte Objekt einen solchen Movement-Controller als Attribut verwendet. Jeder Character im Spiel hat also quasi seine eigene Kamera bei sich, die der Spieler theoretisch anspringen und durch deren Linse er die 3D-Welt betrachten kann – natürlich immer vorausgesetzt, die Applikation gestattet dies. Aber auch jedes Objekt im Spiel benötigt die Funktionalität, sich drehen und bewegen zu können und danach immer noch seine eigenen lokalen Achsen zu kennen. Das entspricht einer Kamera, die auch ihre lokalen Achsen kennen und für die Viewmatrix des Renderers angeben muss. Von daher ist es also nur sinnvoll, ein lokales Achsensystem in einer Klasse einzuführen und diese Klasse mit Methoden auszustatten, die eine Rotation des lokalen Achsensystems zulassen.

Alles nur abstrakt

Genau eine solche Klasse ist der ZFXMovementController, den wir im Folgenden implementieren werden. Dabei ist der Movement-Controller als solches keine Klasse, die man direkt verwenden kann. Sie ist als abstrakte Basis-klassse angelegt, weil sie eine rein virtuelle Methode beinhaltet und daher nicht instanziiert werden kann. Der Sinn und Zweck dieser Klasse ist es, eine Basis für Ableitungen zu bieten. Im folgenden Abschnitt erfahrt ihr mehr dazu.

Implementierung der Basisklasse

Die Basisklasse `ZFXMovementController` definiert einen Satz an Attributen und Methoden, die man für die oben erläuterte Aufgabe i.d.R. immer benötigen wird. In diesem Abschnitt zeige ich kurz die sehr einfache Implementierung der Basisklasse, und danach gehen wir auf zwei Ableitungen ein, die wir ebenfalls für unsere Bibliothek `ZFXGeneral` implementieren werden. Nach einem analogen Schema ist es dann sehr einfach möglich, jede beliebige Art von Bewegung bzw. Kameraführung als Ableitung eines Movement-Controllers zu programmieren – von einem Controller für einen Kampfpanzer bis hin zu einem Controller für ein Kampfflugzeug.

*Das ist mir zu
abstrakt!*

ZFXMovementController-Klasse

Hier gibt es nicht viel zu sagen. Jeder, der schon einmal etwas mit 3D-Grafik zu tun hatte, weiß genau, was man alles für das Erstellen einer View-Matrix benötigt. Das sind natürlich die drei lokalen Achsen und die Position der Kamera. Diese Dinge brauchen wir also in unserer Klasse auf alle Fälle und zusätzlich noch ein paar Attribute, in denen wir Drehwinkel und Geschwindigkeiten speichern. Die Klassen-Deklaration sieht daher wie folgt aus:

*Klassen-
Deklaration*

```
class ZFXMovementController {
public:
    ZFXMovementController();
    virtual ~ZFXMovementController();

    virtual void Update(float fElapsedTime)=0;

    // Accessor-Methoden
    ZFXVector GetPos(void)      { return m_vcPos; }
    ZFXVector GetRight(void)   { return m_vcRight; }
    ZFXVector GetUp(void)      { return m_vcUp; }
    ZFXVector GetDir(void)     { return m_vcDir; }
    ZFXVector GetVelocity(void) { return m_vcV; }

protected:
    ZFXVector m_vcPos;        // Position
    ZFXVector m_vcRight;     // Rechts-Vektor
    ZFXVector m_vcUp;        // Hoch-Vektor
    ZFXVector m_vcDir;       // Richtungs-Vektor
    ZFXVector m_vcV;         // Geschw.-Vektor
    ZFXQuat  m_Quat;         // Quaternion für Rotation

    // Rotations-Speed auf lokalen Vektoren
    float    m_fRollSpd;
    float    m_fPitchSpd;
    float    m_fYawSpd;
```



```

float    m_fRollSpdMax;
float    m_fPitchSpdMax;
float    m_fYawSpdMax;

// Rotationswert auf lokalen Vektoren
float    m_fRotX;
float    m_fRotY;
float    m_fRotZ;

float m_fThrust;

// Methoden
virtual void RecalcAxes(void);
virtual void Init(void);
};

```

*Rein virtuelle
Methode*

Die Methode `ZFXMovementController::Update` ist hier rein virtuell gehalten, also nicht definiert. Das hat den Sinn, dass jede Ableitung für eine bestimmte Art von Kamera ihre eigene Update-Methode implementieren muss, die ja letzten Endes das Verhalten der Kamera überhaupt erst ausmacht. In dieser Methode muss darüber entschieden werden, wie die vom Benutzer gesetzten Werte für die Rotationswinkel und die verschiedenen Geschwindigkeiten nach vorne und für die Rotationen in eine Bewegung der Kamera umgerechnet werden.

Setzen der Attribute

*Konstruktor,
Destruktor und
Init-Methode*

Der Konstruktor und der Destruktor dieser Klasse sind mehr oder weniger leer. Im Konstruktor wird nur die `Init()`-Methode aufgerufen, die die Attribute der Klasse auf sinnvolle Initialwerte einstellt. Man beachte, dass der Destruktor sowieso als `virtual` definiert ist, so wie er das eigentlich immer sein sollte, und daher im normalen Programmablauf i.d.R. sowieso nie aufgerufen werden wird.



```

ZFXMovementController::ZFXMovementController(void) {
    Init();
} // constructor
/*-----*/

ZFXMovementController::~ZFXMovementController(void) {
    } // destructor
/*-----*/

void ZFXMovementController::Init(void) {
    m_vcPos.Set(0.0f, 0.0f, 0.0f);
    m_vcRight.Set(1.0f, 0.0f, 0.0f);
    m_vcUp.Set(0.0f, 1.0f, 0.0f);
    m_vcDir.Set(0.0f, 0.0f, 1.0f);
}

```

```

m_vcV.Set(0.0f, 0.0f, 0.0f);
m_fRotX = m_fRotY = m_fRotZ = m_fThrust = 0.0f;
m_fRollSpd = m_fPitchSpd = m_fYawSpd = 0.0f;
m_Quat.x = m_Quat.y = m_Quat.z = 0.0f;
m_Quat.w = 1.0f;
} // constructor

```

Neuberechnung der Achsen

Neben den Accessor-Methoden und der `Init()`-Methode bietet diese Klasse nur noch eine einzige Methode an, und zwar die Neuberechnung der lokalen Achsen. In dieser Methode achten wir nur darauf, dass wir die Rotationswinkel in den Attributen im Bereich von 2π , also 360 Grad, halten. Danach erstellen wir ein neues Quaternion für diese Rotationswinkel und fügen dieses dem bisherigen Quaternion hinzu, das die bisher aktuelle Rotation gespeichert hat. Danach ziehen wir noch die lokalen Achsen des Controllers aus der entsprechenden Rotationsmatrix, die wir aus dem Quaternion extrahiert haben.

Standard

```

void ZFXMovementController::RecalcAxes(void) {
    ZFXQuat    qFrame;
    ZFXMatrix  mat;

    static float f2PI = 6.283185f;

    // im Bereich von 360 Grad halten
    if (m_fRotX > f2PI) m_fRotX -= f2PI;
    else if (m_fRotX < -f2PI) m_fRotX += f2PI;

    if (m_fRotY > f2PI) m_fRotY -= f2PI;
    else if (m_fRotY < -f2PI) m_fRotY += f2PI;

    if (m_fRotZ > f2PI) m_fRotZ -= f2PI;
    else if (m_fRotZ < -f2PI) m_fRotZ += f2PI;

    // neues Quaternion für diese Rotation
    qFrame.MakeFromEuler(m_fRotX, m_fRotY, m_fRotZ);

    // zur bisherigen Rotation addieren
    m_Quat *= qFrame;

    // lokale Achsen extrahieren
    m_Quat.GetMatrix(&mat);

    m_vcRight.x = mat._11;
    m_vcRight.y = mat._21;
    m_vcRight.z = mat._31;
}

```



```

m_vcUp.x    = mat._12;
m_vcUp.y    = mat._22;
m_vcUp.z    = mat._32;

m_vcDir.x   = mat._13;
m_vcDir.y   = mat._23;
m_vcDir.z   = mat._33;
} // RecalcAxes

```

Gimbal Lock?
Nie gehört!

Dadurch, dass wir an dieser Stelle Quaternions für die Rotation verwenden, umgehen wir das Problem des Gimbal Lock. Wir kommen in keine Situation, in der sich unsere Achsen ineinander verheddern und die Rotationen falsch ausgeführt werden. Damit kann die Basisklasse bereits eine Kamera mit sechs Freiheitsgraden bieten. Ob eine Ableitung das auch wirklich wünscht oder ob sie die Neuberechnung der Achsen nicht lieber durch eine überladene Methode selbst implementiert, sein mal dahingestellt.

Kopplung mit der View-Matrix

**View-Matrix aus
lokalen Achsen
und der Position**

Die View-Matrix, um die Szene am Bildschirm über unseren Renderer anzuzeigen, können wir aus den drei Vektoren der lokalen Achsen und dem Positionsvektor des Movement-Controllers erstellen. Dazu verwenden wir in einer Applikation die entsprechenden Accessor-Methoden. Es wäre natürlich auch denkbar, dass man eine entsprechende View-Matrix gleich im Movement-Controller erstellt und diese dann per Methodenaufruf zurückgibt. Dann müsste unser Render-Interface aber auch eine entsprechende Methode mit einer Matrix als Parameter anbieten. Wir haben jedoch dort bisher nur die Methode mit den vier einzelnen Vektoren als Parameter implementiert. Aber wer es gern etwas bequemer mag, der kann das noch in die Engine einbauen.

Ableitung einer freien Kamera

**Klassen-
Deklaration**

Die Ableitung einer freien Kamera ist nicht weiter kompliziert. Betrachten wir zunächst die Klassen-Deklaration ZFXMCFree.



```

class ZFXMCFree : public ZFXMovementController {
public:
    ZFXMCFree(void) {ZFXMovementController::Init();}
    virtual ~ZFXMCFree(void) { /*nichts zu tun*/ }

    virtual void Update(float fElapsedTime);

    // Methoden zur Beeinflussung der Bewegung
    void AddRotationSpeed(float x, float y, float z);
    void SetRotationSpeed(float x, float y, float z);
    void SetRotationSpeedX(float f) {m_fPitchSpd=f;}
    void SetRotationSpeedY(float f) {m_fYawSpd =f;}

```



```

void SetRotationSpeedZ(float f) {m_fRollSpd =f;}
void AddThrust(float f)         {m_fThrust += f;}
void SetThrust(float f)         {m_fThrust  = f;}

// Attribute direkt setzen (nur für Init)
void SetRotation(float rx, float ry, float rz);
void SetPos(ZFXVector &vc)
    { memcpy(&m_vcPos, &vc, sizeof(ZFXVector)); }
void SetRight(ZFXVector &vc)
    { memcpy(&m_vcRight, &vc, sizeof(ZFXVector)); }
void SetUp(ZFXVector &vc)
    { memcpy(&m_vcUp, &vc, sizeof(ZFXVector)); }
void SetDir(ZFXVector &vc)
    { memcpy(&m_vcDir, &vc, sizeof(ZFXVector)); }
}; // class

```

Einstellen der Rotationen

Die Kontrolle der Bewegung der freien Kamera soll explizit nicht über das Einstellen von Rotationswinkeln erfolgen, auch wenn dies natürlich möglich ist. Normalerweise sollte eine Applikation einem Movement-Controller lediglich eine Geschwindigkeit für die geradlinige Bewegung sowie die Rollgeschwindigkeiten auf den drei lokalen Achsen angeben. Daraus berechnet die Klasse dann selbst die entsprechenden Rotationswinkel. Hier sind die drei Methoden, mit denen beide Möglichkeiten umgesetzt werden können. Man sieht auch gleich, dass die erstgenannte Möglichkeit deshalb ineffizient ist, weil sie eine Neuberechnung der Achsen an dieser Stelle erforderlich macht.

Totale Kontrolle

```

void ZFXMCFree::SetRotation(float x, float y, float z){
    m_fRotX = x;
    m_fRotY = y;
    m_fRotZ = z;
    RecalcAxes();
} // SetRotation

/*-----*/

void ZFXMCFree::AddRotationSpeed(float sx, float sy,
                                float sz) {

    m_fPitchSpd += sx;
    m_fYawSpd   += sy;
    m_fRollSpd  += sz;
} // AddRotationSpeed

/*-----*/

void ZFXMCFree::SetRotationSpeed(float sx, float sy,
                                float sz) {

    m_fPitchSpd = sx;
    m_fYawSpd   = sy;
    m_fRollSpd  = sz;
} // SetRotationSpeed

```



Update der Kamera

Straight-forward- Implementierung

Das Update der freien Kamera in jedem Frame ist nun vollkommen trivial. Wir müssen die eingestellte Rotationsgeschwindigkeit in RAD pro Minute einfach zu den Rotationswinkel-Attributen der Klasse addieren und zuvor mit der Zeit verrechnen, die seit dem letzten Frame vergangen ist. Danach berechnen wir den Geschwindigkeitsvektor und versetzen die Position entsprechend. Man erkennt hier bereits, dass sich ein Movement-Controller gemäß dem Trägheitsgesetz im reibungsfreien Raum so lange geradlinig fortbewegt, bis seine Geschwindigkeit von außen verändert wird. Zu guter Letzt rufen wir die Methode zur Neuberechnung der Achsen, basierend auf den veränderten Rotationswinkeln, auf. Da diese Methode hier nicht überladen ist, wird dabei natürlich die Implementierung aus der Basisklasse angesprungen, die Quaternions verwendet.



```
void ZFXMCFree::Update(float fET) {
    // Rotationsgeschwindigkeit hinzufügen
    m_fRotX = (m_fPitchSpd * fET);
    m_fRotY = (m_fYawSpd * fET);
    m_fRotZ = (m_fRollSpd * fET);

    // Geschwindigkeitsvektor berechnen
    m_vcV = m_vcDir * m_fThrust * fET;

    // Position verschieben
    m_vcPos += m_vcV;

    // Achsen neu berechnen
    RecalcAxes();
} // Update
```

Fertig!

Das war es tatsächlich schon für diese Klasse. Mit Hilfe dieser paar Zeilen haben wir eine Kamera-Klasse entwickelt, die dem Anwender eine vollkommen freie Bewegung im Raum ohne lästigen Gimbal Lock ermöglicht.

Ableitung einer 1st-Person-Kamera

Zweites Beispiel

Das zweite Beispiel für eine hilfreiche Ableitung unserer Basisklasse für Movement-Controller ist die folgende Klasse für eine First-Person-Kamera. Die meisten Methodenaufrufe sind bereits in der Klassen-Deklaration erläutert. Hier fällt aber sofort auf, dass diese Klasse eine eigene Überladung der Methode `RecalcAxes()` definiert. Aber schauen wir uns zuerst die Klasse an.

```

class ZFXMCEgo : public ZFXMovementController {
public:
    ZFXMCEgo(void);
    virtual ~ZFXMCEgo(void);

    virtual void Update(float fElapsedTime);

    // Methoden zur Beeinflussung der Bewegung
    void      GetRotation(float*X,float*Y,float*Z);
    ZFXVector GetRotation(void);

    void SetRotationSpeedX(float f) {m_fPitchSpd=f;}
    void SetRotationSpeedY(float f) {m_fYawSpd  =f;}
    void SetSpeed(float a)          {m_fSpeed   =a;}
    void SetSlideSpeed(float a)     {m_fSlide   =a;}

    // Attribute direkt setzen (nur für Init)
    void SetRotation(float rx, float ry, float rz);
    void SetPos(ZFXVector &vc)
        { memcpy(&m_vcPos, &vc, sizeof(ZFXVector)); }
    void SetRight(ZFXVector &vc)
        { memcpy(&m_vcRight, &vc, sizeof(ZFXVector)); }
    void SetUp(ZFXVector &vc)
        { memcpy(&m_vcUp,   &vc, sizeof(ZFXVector)); }
    void SetDir(ZFXVector &vc)
        { memcpy(&m_vcDir,  &vc, sizeof(ZFXVector)); }

private:
    float m_fSpeed;
    float m_fSlide;
    void RecalcAxes(void);
};

```



Einstellen der Rotationen

Die Implementierungen bezüglich der Einstellung von Rotationen bzw. deren Abfrage sind nun wiederum vollkommen simpel und bedürfen sicherlich keiner weiteren Erläuterung mehr.

Nichts Neues

```

void ZFXMCEgo::SetRotation(float rx, float ry, float rz)
{
    m_fRotX = rx;
    m_fRotY = ry;
    m_fRotZ = rz;
    RecalcAxes();
} // SetRotation

/*-----*/

void ZFXMCEgo::GetRotation(float *pfX, float *pfY,

```



```

float *pfZ) {
    if (pfX) *pfX = m_fRotX;
    if (pfY) *pfY = m_fRotY;
    if (pfZ) *pfZ = m_fRotZ;
} // GetRotation

/*-----*/

ZFXVector ZFXMCEgo::GetRotation(void) {
    return ZFXVector( m_fRotX,
                     m_fRotY,
                     m_fRotZ );
} // GetRotation

```

Neuberechnung der Achsen

*Warum eine
Überladung?*

Die freie Kamera, die die Neuberechnung der Achsen aus der Basisklasse übernimmt, hat einen kleinen Schönheitsfehler: Die Rotationen werden tatsächlich vollkommen frei umgesetzt, wobei es dem Spieler sehr wohl möglich ist, die Kamera hilflos zu verdrehen. Man kann beispielsweise die Kamera nach links schwenken und dann nach oben. Wenn man nun wieder nach rechts schwenkt, hat man natürlich eine andere Rotationsachse als bei der ursprünglichen Drehung nach links. Das ist das Schicksal einer vollkommen freien Kamera mit sechs Freiheitsgraden. Man wird sie nur schwerlich nach einer Drehung auf mindestens zwei Achsen wieder in die Ausgangsposition zurückführen können.

Das ist bei einem First-Person-Shooter bzw. einer entsprechenden Kamera natürlich unerwünscht, denn der Spieler soll hier immer aufrecht auf dem Boden stehen und nicht irgendwie schief im Raum hängen. Zudem müssen die möglichen Rotationswinkel auf der Z- und der X-Achse beschränkt werden, damit der Spieler seiner virtuellen Spielfigur nicht im wahrsten Sinne des Wortes das Genick bricht oder sie Kopf stehen lässt. Die folgende Methode setzt also die Rotation der lokalen Achsen nach der guten alten Euler-Winkel-Methode um.



```

void ZFXMCEgo::RecalcAxes(void) {
    ZFXMatrix mat;

    static float f2PI = 6.283185f;

    // im Bereich von 2 PI = 360 Grad halten
    if (m_fRotY > f2PI) m_fRotY -= f2PI;
    else if (m_fRotY < -f2PI) m_fRotY += f2PI;

    // Hoch/Runter nur maximal 80 Grad
    if (m_fRotX > 1.4f) m_fRotX = 1.4f;
    else if (m_fRotX < -1.4f) m_fRotX = -1.4f;

```

```

// initialisiere Achsen
m_vcRight = ZFXVector(1.0f, 0.0f, 0.0f);
m_vcUp    = ZFXVector(0.0f, 1.0f, 0.0f);
m_vcDir   = ZFXVector(0.0f, 0.0f, 1.0f);

// rotiere um Y-Achse
mat.RotaArbi(m_vcUp, m_fRotY);
m_vcRight = m_vcRight * mat;
m_vcDir   = m_vcDir * mat;

// rotiere um die Z-Achse
mat.RotaArbi(m_vcRight, m_fRotX);
m_vcUp    = m_vcUp * mat;
m_vcDir   = m_vcDir * mat;

// Rundungsfehler korrigieren
m_vcDir.Normalize();
m_vcRight.Cross(m_vcUp, m_vcDir);
m_vcRight.Normalize();
m_vcUp.Cross(m_vcDir, m_vcRight);
m_vcUp.Normalize();
} // RecalcAxes

```

Wie ihr seht, erlauben wir dem Controller gar keine Bewegung auf der Z-Achse. Wer möchte, kann dies ja noch nachträglich einbauen, um auch um Ecken schielen zu können. Die Bewegung des Kopfes auf der X-Achse ist auf jeweils 80 Grad nach oben und nach unten beschränkt.

Kein Z und nur ein bisschen X

Wichtig zu beachten ist hierbei auch, dass wir zuerst die Rotation auf der Y-Achse durchführen und erst danach die Rotation auf der X-Achse. Würden wir zuerst auf der X-Achse rotieren, dann würde die Y-Achse auch gleich mit rotiert und wäre nicht mehr vertikal. Somit würde die Drehung des Controllers nach rechts und links in einer schrägen Drehung auch zum Teil nach oben bzw. unten enden, was nicht korrekt wäre. Drehen wir zuerst die Y-Achse, so wird die X-Achse zwar auch mit verdreht, aber nur nach rechts bzw. links. Sie bleibt aber horizontal, und daher können wir anschließend die Drehung auf der Y-Achse korrekt durchführen.

Zuerst Y, dann X

Update der Kamera

Das Update der First-Person-Kamera verläuft hier ganz analog zu dem Update der freien Kamera. Wir haben es allerdings noch mit einem Attribut `m_fSlide` zu tun, das die Geschwindigkeit angibt, mit der sich der Controller seitlich bewegen möchte. Diese Bewegung kommt dann gegebenenfalls noch zu der Bewegung entlang des Richtungsvektors hinzu.

Ausweichen



```
void ZFXMCEgo::Update(float fET) {
    ZFXVector vcS;
    // Rotationsgeschwindigkeit addieren
    m_fRotX += (m_fPitchSpd * fET);
    m_fRotY += (m_fYawSpd * fET);
    m_fRotZ += (m_fRollSpd * fET);

    // Achsen neu berechnen
    RecalcAxes();

    // Geschwindigkeitsvektor berechnen
    m_vcV = m_vcDir * m_fSpeed * fET;
    vcS = m_vcRight * m_fSlide * fET;

    // Position bewegen
    m_vcPos += m_vcV + vcS;
} // Update
```

*Fertig, zum
Zweiten*

Damit ist auch unsere zweite Beispiel-Ableitung zur Implementierung einer Kameraklasse fertig. Nun erstellen wir eine kleine Demo-Applikation, in der wir unsere Kameras testen können.

12.4 Demo-Applikation zur Anwendung der Bibliothek

*Ich sprach
»Bewege dich«,
und ich bewegte
mich*

Natürlich gibt es jetzt auch wieder eine kleine Applikation, in der wir die Funktionalität unserer hier geleisteten Arbeit testen können. Schließlich glauben wir nichts, was wir es nicht wirklich in Aktion gesehen haben. Die Demo-Applikation bietet dem Betrachter eine freie Kamera, mit der er durch den virtuellen Raum manövrieren kann. Die Applikation ist ansonsten mehr oder weniger identisch mit der aus dem sechsten Kapitel. Die Applikationen findet ihr natürlich auf der CD-ROM zu diesem Buch.

12.5 Ein Blick zurück, zwei Schritt nach vorn

Kurzer Rückblick

Der Rückblick kann an dieser Stelle sehr kurz ausfallen. Die eigentliche Umsetzung einer Kamera ist natürlich trivial, und darauf brauchen wir nicht länger einzugehen. Interessant ist hier vielleicht noch die Variante der freien Kamera mit dem Quaternion, da sicherlich auch jeder schon einmal den Gimbal Lock am eigenen virtuellen Leib gespürt hat. Neu sollte den meisten aber die Kapselung der Kamera-Funktionalität in einem Movement-Controller sein, ebenso wie das Verfahren über die Ableitungen einer Basis-klassse um verschiedenartige Kameratypen zu implementieren.

Nach diesem eher lässigen und daher vielleicht etwas langweiligen Kapitel ziehen wir jetzt unseren Helm auf und schließen die Splitterschutzweste. Im nun folgenden Kapitel stürzen wir uns wagemutig auf das Scene-Management und werden detailliert in Erfahrung bringen, was ein BSP-Baum ist, wozu man eine Portal-Engine braucht und wer eigentlich John Carmack ist. Alles klar? Dann los!

Hauptsache, Helm auf!

13 Scene-Management

*»Den klugen Mann erkennt man an den Antworten, die er gibt.
Den weisen Mann erkennt man an den Fragen, die er stellt.«
(Chinesisches Sprichwort)*

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Sinn und Zweck des Scene-Managements
- ➔ Einführung in Quadrees und Terrain-Rendering
- ➔ Betrachtung von BSP-Bäumen und Octrees für Kollisionsabfragen
- ➔ Portal-Engine-Ansatz und PVS-Theorie
- ➔ Implementierung eigener Klassen für den BSP-Baum und den Octree
- ➔ Demo-Applikation mit BSP-Baum und Octree

13.1 Sinn des Scene-Managements

Ganz im Sinne der alten Chinesen wollten wir in diesem Kapitel versuchen, die richtigen Fragen zu stellen. Die erste Quizfrage dieses Kapitels lautet natürlich: Warum werden Computerspiele mit der immer neueren und immer besseren Hardware im Bereich der Grafikkarten nicht automatisch so rasend schnell, dass man sich über Performance keine Gedanken mehr zu machen braucht? Die Hardware hat bald einen Status erreicht, bei dem eine Beschleunigung des Vertex-Processing (3D-Pipeline) kaum noch durch herkömmliche Technologie erreicht werden kann. Bevor wir aber Computer entwickeln, die durch die Tunnelung im Quantenbereich wieder einen nennenswerten Geschwindigkeitszuwachs bringen werden, wird es noch eine ganze Weile dauern.

*Die richtigen
Fragen*

Trotzdem kann man auf modernen Grafikkarten einen ganzen Level von einem Spiel wie beispielsweise *Quake 1* von *id Software* brute-force in Echtzeit ohne Scene-Management rendern – eine Aufgabe, an der eine Grafikkarte Ende 1996, als das Spiel erschien, unter Garantie erstickt wäre. Seinerzeit musste man aufwendige Scene-Management-Verfahren bemühen, um ausschließlich den Teil eines Levels zu berechnen, der in einem Frame

Damals und heute

auch potenziell sichtbar war und beispielsweise nicht etwa hinter der Kamera lag. Doch wenn man das heutzutage so einfach machen kann, warum brauchen wir ein Kapitel über Scene-Management? Nun, wir brauchen es, weil die modernen Computerspiele auf aktuellen Grafikkarten eher langsamer werden als schneller. Das kommt natürlich daher, dass die Entwickler immer mehr Effekte ausnutzen und jeder einzelne Pixel mit Effekten wie Spot-Light, Bump-Mapping, vorberechneten Shadow-Maps, Specular-Reflection und Shadow-Volumes bombardiert wird, bis er im wahrsten Sinne des Wortes grün und blau ist. Früher begnügte man sich dort mit einer Textur und maximal noch einer vorberechneten Lightmap.

*Man nimmt, was
man kriegen kann*

Je schneller eine Grafikkarte also wird und je mehr Effekte sie anbietet, desto mehr von diesen Features werden die Entwickler auch nutzen. Auch wenn eine Grafikkarte heutzutage weitaus mehr Geometrie weitaus schneller rendern kann, als man es vor fünf Jahren zu träumen wagte, so muss nun bei einem anspruchsvollen Grafikspiel pro gerendertem Pixel unverhältnismäßig viel Mehrarbeit geleistet werden. Daher gilt heute ebenso wie damals, dass nur ein geculltes Triangle ein gutes Triangle ist – oder wie mein guter Freund, die DirectX SDK-Dokumentation, im Abschnitt *Performance Optimization* sagen würde: »*Remember, the fastest polygons are the ones you don't draw.*«

Scene-Management

Das gute alte Scene-Management ist also nach wie vor ein wichtiges Thema, wenn man mehr als nur ein paar tausend Demo-Polygone auf dem Schirm anzeigen lassen will. Wenn man mit Leveln von mehreren Hundertausenden Polygonen arbeitet und diese gleichzeitig mit Dutzenden von Effekten rendern will, verdampft auch eine GeForce4-GPU schneller, als der Kühler für eine Umdrehung braucht. Mit dem Begriff Scene-Management beschreibt man alle Verfahren, die dazu dienen, aus der Gesamtmenge der vorliegenden Level-Daten diejenigen herauszufinden, die für den aktuellen Frame benötigt werden. Objekte, die beispielsweise hinter dem Betrachter sind, um die nächste Ecke liegen oder fünf Kilometer entfernt sind, braucht man i.d.R. weder zu rendern noch muss man sie bei Kollisionsabfragen berücksichtigen.

Fachjargon zum Scene-Management

*Die Entfernung von Polygonen, die von einer gegebenen Position und Ausrichtung eines Betrachters aus nicht gesehen werden können, weil sie durch andere Polygone vollkommen verdeckt werden, nennt man **Hidden Surface Removal** (Entfernung verdeckter Oberflächen) oder für die Profis nur kurz HSR. Ein wohl etwas bekannterer Begriff für dieselbe Sache lautet **Occlusion Culling** (etwa: Aussortierung des Verdeckten). Dabei ist der letztgenannte Begriff häufig eher als ein Synonym für gröbere schrotflintenartige Ansätze zu sehen, während ein HRS-Verfahren*

mehr darauf ausgerichtet ist, eine exakte Lösung zu finden. Das heißt, beim Occlusion Culling testet man i.d.R. größere und komplexere Objekte mit verschiedenen so genannten »Occludern«, um mit möglichst wenig Aufwand eine möglichst große Menge an verdeckten Polygonen auf der Ebene von Objekten (wie z.B. ganzen 3D-Modellen) vor der Berechnung eines Frames zu entfernen. Bei einem genauen HSR-Verfahren wird man versuchen, jedes einzelne Polygon auf Verdeckung zu testen und wirklich nur diejenigen Polygone zu betrachten, die nicht verdeckt sind.

Occlusion Culling und HSR sind nicht mit dem Entfernen der Polygone zu verwechseln, von denen man nur die Rückseite sehen würde, weil sie vom Betrachter wegschauen. Das nennt man **Backface Culling** oder manchmal auch **Backface Removal** (Rückseiten-Entfernung).

Die dritte Kategorie beim Entfernen von Teilen der Level-Daten aus der Betrachtung eines bestimmten Frames ist das allgemeine **Object Culling** (etwa: Objekt-Aussortierung), das auch **Frustum Culling** genannt wird. Damit meint man das Aussortieren von Polygonen oder Polygon-Objekten, die außerhalb des View-Frustums liegen und aus diesem Grunde nicht sichtbar sind und aus der Berechnung für einen gegebenen Frame ausgeschlossen werden können.

Nachdem wir oben die Notwendigkeit für ein Scene-Management geklärt haben, schauen wir uns im folgenden Abschnitt ein paar der gängigsten Verfahren erst einmal in der Theorie an, bevor wir im letzten Teil des Kapitels zu der praktischen Arbeit kommen. An dieser Stelle kann ich jedoch schon einmal verkünden, dass das Wort »Management« in just dieser Sekunde zum meistgebrauchten kontextgebundenen Substantiv dieses Buches avanciert ist und dafür von mir mit dem *Goldenen Orthographen* ausgezeichnet wird.

*Wo wären wir
bloß ohne
Management?*

13.2 Scene-Management-Techniken

An dieser Stelle verwende ich gern die Begriffe Indoor, Outdoor und Nodoor, um 3D-Grafik-Computerspiele zu kategorisieren. Als »Indoor-Spiele« bezeichnet man normalerweise Spiele, die größtenteils in geschlossenen Umgebungen stattfinden. Das sind insbesondere die klassischen Ego-Shooter wie beispielsweise *Doom*, die in geschlossenen Gebäuden, Anlagen, Höhlen oder Dungeons spielen. Solche Spiele bieten heutzutage auch ein gewisses Maß an Aktivitäten unter freiem Himmel. Doch bei genauerer Betrachtung ist dieser Freiraum dann **immer** durch Mauern oder enge, steile Berghänge extrem begrenzt. Das hat den Grund, dass man für diese Art von Spielen immer Scene-Management-Ansätze verwendet, die sich für große, offene Flächen unter freiem Himmel nicht so gut eignen würden.

*Indoor, Outdoor,
Nodoor*

*Unter freiem
Himmel*

Spiele, die unter freiem Himmel in der Natur stattfinden und die nicht durch künstliche Hindernisse zur Vereinfachung der Arbeit für die Programmierer begrenzt sind, nennt man »Outdoor-Spiele«. Dort hat man es weniger mit engen, verwinkelten Geometrien zu tun, sondern mit sehr weitläufigen und sehr flachen Strukturen. Selbst die Hügel und kleinen Berge, die man in Outdoor-Spielen findet, sind im Vergleich zu dem Inneren eines mehrstöckigen Gebäudes eher flach. Damit ist in diesem Kontext gemeint, dass über und unter dem Terrain des Berges nicht mehr viel an Geometrie kommt, höchstens noch ein paar Bäume. Outdoor-Settings haben daher die angenehme Eigenschaft, dass man sie sehr gut auf die zweidimensionale Betrachtung vereinfachen kann. Selbst wenn es eine Höhe im Terrain gibt, so ist diese für das Scene-Management nicht von Bedeutung. Es zählt dort viel mehr die Position eines Stücks Terrain in der Ebene. Dafür haben Outdoor-Spiele das Problem, dass das Terrain eigentlich unendlich groß sein müsste, wenn man beispielsweise mit einem schnellen Fahrzeug in diesem unterwegs ist.

Im freien Raum

Die letzte Kategorie sind die »Nodoor-Spiele«. Dabei handelt es sich um alles, was nicht in die erste oder zweite Kategorie fällt. Der Hauptvertreter hier dürfte die klassische Weltraum-Simulation im Stil von *Wing Commander* oder *Freelancer* sein. Das Scene-Management wird hier dadurch extrem vereinfacht, dass man in einem Sonnensystem vergleichsweise wenig Polygone hat, weil man keine Gebäude oder Landschaften modellieren muss. Die paar Planeten und Raumschiffe, die dort unterwegs sein werden, bekommt man relativ zügig gut gemanagt. Interessant wird es erst dann, wenn man in den Grenzbereich von Nodoor und Outdoor kommt, beispielsweise wenn man die Oberflächen der Planeten so modellieren will, dass man sie vom Weltraum aus erkennen oder gar auf ihnen landen kann, ohne den billigen Ausweg über eine vorgerenderte Animation zu gehen.

Im Folgenden möchte ich ein paar Ansätze zum Scene-Management kurz vorstellen, und dabei jeweils auch darauf hinweisen, für welchen Einsatz sich diese jeweils eignen.

Keine Lösung ist auch eine Lösung

*Warum schwer,
wenn es auch
einfach geht?*

Als eine Variante des Scene-Managements kann man natürlich auch den **Brute-Force-Ansatz** (brutale Gewalt) bezeichnen. Damit ist gemeint, dass man sich gar keine Gedanken über das Management einer Szene macht, sondern einfach alles in die Betrachtung eines Frames mit einbezieht, alle Objekte und deren Polygone rendert und Kollisionsabfragen mit Ihnen macht usw. Das mag ziemlich unsinnig erscheinen, kann in einigen Fällen aber durchaus ein gangbarer Weg sein – insbesondere dann, wenn die darzustellende Szene nicht so komplex oder mit Effekten belegt ist, dass sich der Aufwand zur Implementierung eines besseren Ansatzes rechtfertigen würde.

Im Falle von Computerspielen kann man aber davon ausgehen, dass der Brute-Force-Ansatz relativ selten verwendet wird. Einige Beispiele sind jedoch die kleinen Outdoor-Szenen in Ego-Shootern, bei denen das Terrain auf aktuellen Grafikkarten einfach per Brute Force gerendert wird, da das für die paar hundert Polygone vollkommen ausreichend ist.

Kontinuierliche und diskrete Detail-Level

Wieder sind wir bei meinem Lieblingsthema angelangt, nämlich bei der Kunst, die Motivation hinter einer Aktion zu erkennen. Fragen wir uns also nochmals, was wir mit dem Scene-Management eigentlich erreichen wollen? Wer nun vorschnell antwortet, dass wir möglichst alle verdeckten Polygone aus einer Szene entfernen wollen, der denkt zwar ungefähr in die richtige Richtung. Aber er offenbart damit eine falsche Denkart, die sich zu sehr auf Bekanntes stützt und zu wenig abstrahiert. Der Sinn des Scene-Managements liegt doch darin, eine Szene möglichst schnell berechnen und darstellen zu können. Die Entfernung verdeckter Objekte und Polygone in all ihren Facetten verschiedener Algorithmen ist dabei eine der wichtigsten Methoden. Aber es gibt noch andere. Die Komplexität einer Szene können wir auch dadurch vereinfachen, dass wir die Komplexität sichtbarer Objekte reduzieren. Und genau darum geht es in dem Level-of-Detail-Ansatz, oder auch kurz LoD.

*»Die Augen
können dich
täuschen, traue
ihnen nicht.«*

Man stelle sich vor, man wäre der Kommandant eines Kampfpanzers. Plötzlich meldet der Richtschütze die Sichtung eines anderen Panzers in dreitausend Metern Entfernung. Man schaut durch seine Optik und erhält einen fünfzehnfach vergrößerten Blick auf das Schlachtfeld. Doch bei dieser Entfernung kann man selbst bei Tag und guter Sicht nicht genügend Details ausmachen, um das gesichtete Fahrzeug sicher zu identifizieren. Die Anzahl und die Position der Antennen, die Einheitsmarkierungen und selbst die Form des Turmes verschmelzen zu einem vergleichsweise kleinen Punkt in der Szenerie. Freund oder Feind?

*Kampfpanzer.
Sechs Uhr.
Entfernung Drei-
tausend. KE
geladen.*

Dieses Problem unseres Panzerkommandanten können wir insbesondere dann ganz gut nachvollziehen, wenn wir uns die Statistiken über *Friendly Fire* aus den Waffengängen der jüngsten Geschichte unserer Rasse betrachten. Selbst modernste Waffensysteme helfen eben nicht zu verhindern, dass Kampfhubschrauber die eigenen Truppentransporter zerstören und panzerknackende Flugzeuge ihre eigenen Panzer zielsicher vernichten. Zurück zur Computergrafik: Wenn wir einen solchen Panzer detailliert als 3D-Objekt modelliert haben, mit viertausend Polygonen und wunderschönen Texturen, so haben wir doch dasselbe Problem: Die Bordkanone (BK) eines modernen Panzers schießt auf bis zu drei Kilometer zielgenau. Optimalerweise lässt man also auf dem virtuellen Schlachtfeld unserer Panzersimulation keinen Feind näher als tausendfünfhundert Meter an sich herankommen. Doch selbst auf diese, aus Sicht der BK recht kleine Distanz kann man bei weitem

*Von Panzern zu
Pixeln*

nicht alle Details unserer viertausend Polygone erkennen. Die einzelnen Pixel der rasterisierten Dreiecke belegen vielleicht 2 % des gesamten Bildschirms – wenn überhaupt so viel. Auf diese, aus Sicht der Pixel, große Distanz macht es gar keinen Sinn, ein so detailliertes Modell zu verwenden. Ein Modell mit maximal tausend Polygonen ist dafür vollkommen ausreichend. Einen optischen Unterschied für den Betrachter am Bildschirm wird es nicht geben.

Level of Detail Gerade in den offenen, weiträumigen Outdoor-Spielen ist ein Großteil der Objekte während der längsten Spielzeit eher vergleichsweise weit vom Betrachter am Bildschirm entfernt als so nahe dran, dass man viele Details des Modells ausmachen könnte. Daher zielt der Level-of-Detail-Ansatz genau darauf ab, dem Rechnung zu tragen. Im oben genannten Beispiel würde man daher nicht nur das hochdetaillierte Modell des Panzers mit viertausend Polygonen haben, sondern auch dasselbe Modell mit weniger Details, etwa noch einmal mit zweitausend und einmal mit nur tausend Polygonen. Je weiter weg das Modell dann in einem bestimmten Frame vom Betrachter ist, desto weniger Details muss man verwenden und wählt eine entsprechende Instanz des Modells, beispielsweise den Panzer mit eintausend Polygonen.

Kontinuierlich versus diskret Diese Methode mit verschiedenen Instanzen des Modells nennt man **Discreet Level of Detail** oder auch kurz **DLOD**. Dieser Ansatz heißt so, weil man nur bestimmte, also diskrete Detailstufen des Modells hat. Diese müssen vorher durch den Modell-Designer erstellt werden. Dem steht die Variante des **Continuous Level of Detail**, oder kurz **CLOD**, gegenüber. Dabei geht man i.d.R. von dem hoch detaillierten Modell aus und berechnet zur Laufzeit eine Version mit weniger Details kontinuierlich, wobei der Detailgrad eine Funktion über die Entfernung des Modells zum Betrachter sein wird. Dieser Ansatz eignet sich eher für umfassende Objekte wie beispielsweise ein Terrain. Ein 3D-Modell kann man durch einen solchen Ansatz (wie beispielsweise **Progressive Meshs** von Hugues Hoppe als Form von CLOD) zwar auch behandeln. Allerdings wäre hier die Performance vorberechneter Detailstufen über einen DLOD-Ansatz aufgrund der Vorberechnung effizienter und hätte auch ein optisch besseres Ergebnis als ein automatisiertes Verfahren. Abbildung 13.1 zeigt das Modell einer Cessna, das als Progressive Mesh vereinfacht wurde, zu drei diskreten Zeitpunkten. Natürlich kann man auch ein Tool programmieren, das ein solches CLOD-Verfahren verwendet, um diskrete Stufen eines Modells für ein DLOD-Verfahren zu berechnen.

Wie man DLOD anwendet, dürfte damit klar sein. Für CLOD habe ich eben schon ein Beispiel genannt, zu dem ihr beim Googlen einiges finden werdet. Der folgende Abschnitt beschreibt mit den so genannten **Quadtrees** ein weiteres Verfahren der CLOD-Kategorie, das speziell für Terrains geeignet ist.

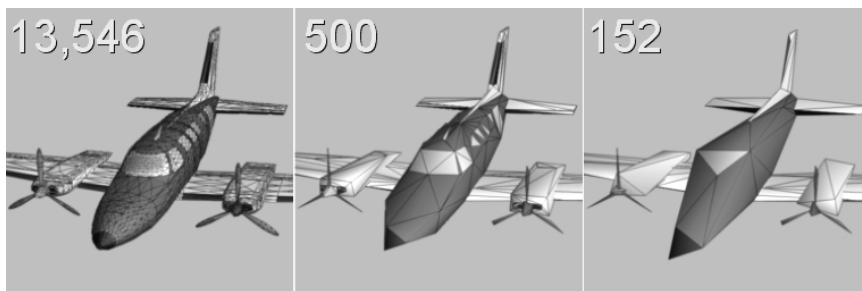


Abbildung 13.1: Berechnung einer Low-Detail-Version einer Cessna über Progressive Meshes von Hugues Hoppe, ausgehend von der High-Detail-Variante links. Die Nummer gibt jeweils die Anzahl der Dreiecke des Modells an. (Quelle: SIGGRAPH 1996 Slides von Hugues Hoppe, <http://research.microsoft.com/~hoppe/#pm>)

Quadtrees

Eben haben wir einen Ansatz des Scene-Managements gesehen, der auf der Reduktion der Details von 3D-Modellen basiert. Nun schauen wir uns ein Verfahren an, mit dem man schnell entscheiden kann, ob ein Objekt überhaupt sichtbar ist oder nicht. Wir kennen zwar schon den View-Frustum-Test. Doch dieser hat zunächst den Nachteil, dass man jedes Objekt in einem Spiel einzeln testen müsste. Viel besser ist ein hierarchischer Ansatz, bei dem man den gesamten Level in eine Hierarchie aufteilt. Innerhalb dieser Hierarchie kann man dann sehr schnell große Teile des Levels mit einem Test verwerfen.

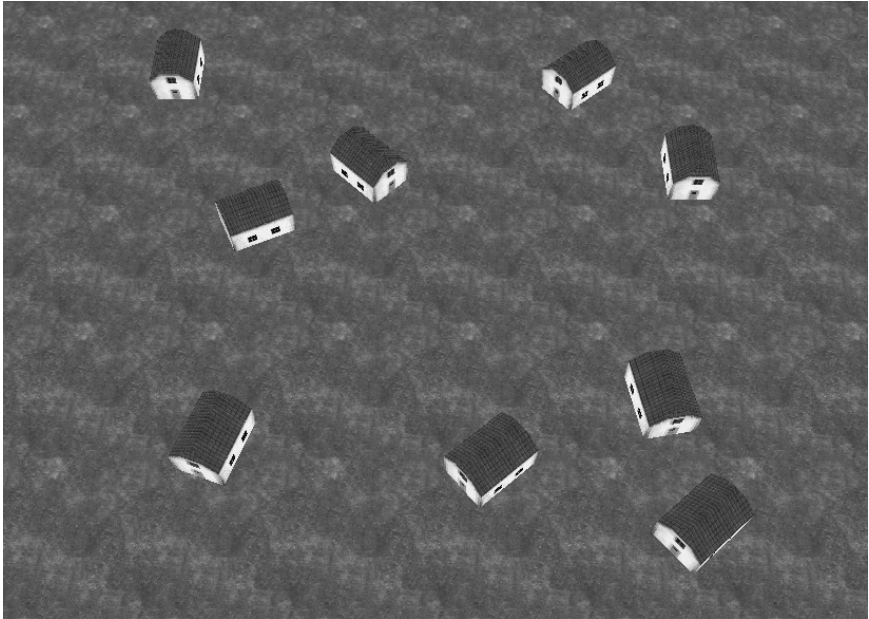
Quattro Treegioni

Allgemeine Quadtrees

Einen sehr einfachen Einstieg in solche hierarchischen Verfahren bietet der Quadtree. Vorweg sei aber gesagt, dass sich ein Quadtree eigentlich nur für zweidimensionale Probleme eignet. Doch bevor wir uns gleich wieder von ihm abwenden, sei ebenfalls gesagt, dass man insbesondere Terrain-Probleme auf eine zweidimensionale Betrachtung reduzieren kann, weil die Höhe des Terrains im Vergleich zu seiner Ausdehnung auf den anderen beiden Achsen zu vernachlässigen ist. Auch für Spiele, die sich auf einer Ebene abspielen, wie beispielsweise in der Iso- oder der Vogel-Perspektive, eignet sich der Quadtree sehr gut. Sinngemäß übersetzt bedeutet Quadtree so viel wie »Vierfach-Baum«. Aber betrachten wir zunächst Abbildung 13.2, in der ein kleines Dörfchen aus der Vogelperspektive zu sehen ist.

Abbildung 13.2:

Eine 3D-Szene mit
einem Dörfchen aus
einer schrägen
Vogel-Perspektive



*Die Welt ist keine
Scheibe, sondern
ein Viereck*

Ein Scene-Management dieses Dörfchens kann nun an zwei Punkten im Programmablauf ansetzen und Performance-Steigerungen bringen: zum einen beim Rendern der Szene und zum anderen bei der Kollisionsabfrage zwischen einem bewegten Objekt wie z.B. der Spielfigur und den Häusern des Dorfes. Obwohl wir das Dorf hier aus der Vogel-Perspektive sehen, nehmen wir einmal an, dass der Spieler das Spiel aus der Ego-Perspektive wie durch die Augen seiner Figur im Spiel sieht. Wenn er nun seine Spielfigur durch das Dorf manövriert, dann müssen wir darauf achten, dass diese Figur nicht durch die Wände der Häuser laufen kann. Wir brauchen also eine Kollisionsabfrage. Der Brute-Force-Ansatz besteht darin, dass man beispielsweise über Bounding-Boxen testen, ob die Spielfigur mit den Häusern kollidiert. Das funktioniert sehr gut, ist aber viel zu langsam.

*Quadtree-
Algorithmus*

Nun kommt im wahrsten Sinne des Wortes der Quadtree ins Spiel. Der Quadtree ist eine Baumstruktur, bei der jeder Node (dt. Knoten) des Baumes entweder genau vier Children (dt. Kinder) hat oder gar keine. Ein Node ohne Kinder ist ein Leaf (dt. Blatt) des Baums, und ein Node ohne einen Parent (dt. Vater) ist der Root-Node (dt. Wurzel-Knoten). Die Wurzel des Baums ist ein zweidimensionales Rechteck in der horizontalen Ebene, das den gesamten Level umspannt, also ein zweidimensionales **Bounding-Rectangle**. So lange innerhalb des Bounding-Rectangles noch eine bestimmte Menge an Geometrie enthalten ist, muss der Node weiter unterteilt werden. Dazu zerlegt man dieses Bounding-Rectangle in vier gleich große Bounding-Rectangle als neue Children, die jeweils ein Viertel der Fläche des Nodes belegen. Diese werden dann bei Bedarf jeweils wieder geviertelt, bis die entsprechende Bedingung für einen Node erreicht ist, der damit ein Leaf ist.

Abbildung 13.3 zeigt das Ergebnis, nachdem wir diesen Algorithmus für unser Dörfchen durchgeführt haben. Die Abbruchbedingung lautet dabei, dass ein Node nur genau einen Teil zusammenhängende Geometrie, sprich nur den Teil eines einzigen Hauses, enthalten darf. Sind zwei ganze Häuser oder auch nur Teile von mindestens zwei verschiedenen Häusern in einem Node, so muss er weiter unterteilt werden.

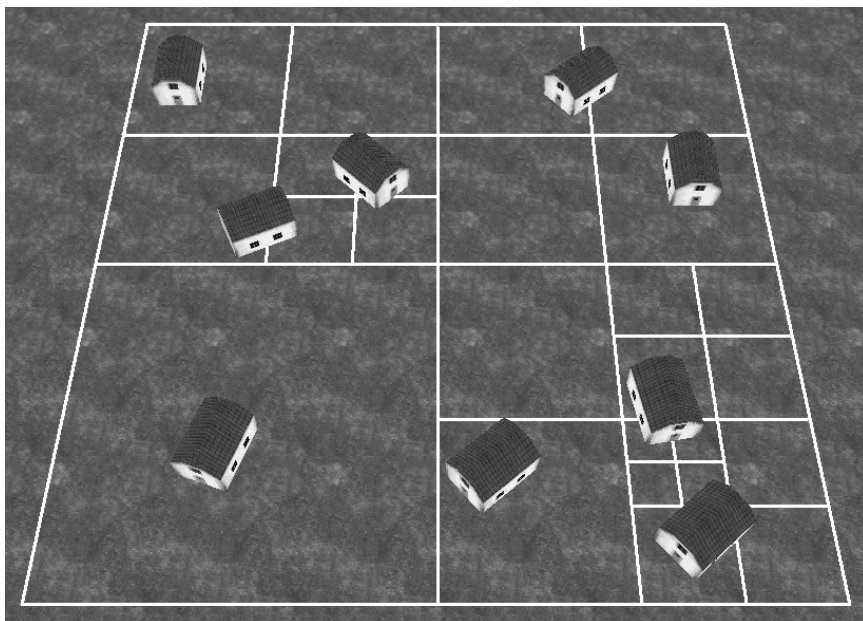


Abbildung 13.3:

Der Quadtree für das Dörfchen. Der Raum ist jeweils so weit unterteilt, dass jedes Leaf des Baums nur Teile eines einzigen Hauses enthält.

Der Quadtree hat damit also den Raum zweidimensional so weit unterteilt, bis jedes Leaf nur noch Teile eines einzelnen Hauses enthält. Der Vorteil ist nun, dass wir ohne viel Rechenaufwand exakt das eine Haus aus dem ganzen Dorf ermitteln können, mit dem die Spielfigur potenziell kollidiert. Wir nehmen dazu die Position des Spielers oder auch die Bounding-Box der Spielfigur und schieben sie in die Wurzel des Quadtrees, die dem gesamten Dorf entspricht. So lange ein Node des Quadrees noch Children hat, prüfen wir, in welchem der vier Children die Position des Spielers bzw. die Bounding-Box liegt. Im ersten Schritt sortieren wir damit effektiv drei Viertel der Fläche des Dorfes für die weiteren Berechnungen aus und brauchen uns nur auf das eine Viertel des Dorfes zu konzentrieren, in dem der Spieler sich tatsächlich befindet. Diesen Test führen wir so lange durch, bis wir in das eine Leaf des Baums gelangen, in dem sich der Spieler wirklich befindet. Erst hier führen wir dann den vergleichsweise aufwändigen Kollisionstest durch. Der große Vorteil hierbei ist, dass wir einen echten Kollisionstest schlimmstenfalls für ein einziges Haus ausführen müssen. Bestenfalls gelangen wir in ein Leaf, das gar keine Geometrie enthält, und damit wissen wir auch gleich, dass gar keine Kollision stattgefunden hat.

***Vorteile dieser
Aufteilung***

*Quadrees für die
Performance-
Steigerung beim
Rendern*

Das Rendern kann man mit Quadrees zwar nicht wirklich beschleunigen, aber man kann effektiv eine Menge an Objekten aussortieren, die gar nicht im View-Frustum des Spielers liegen. Normalerweise würden wir jedes Haus unseres Dorfes einzeln mit einer Bounding-Box gegen das View-Frustum testen, um zu evaluieren, ob wir es rendern müssen. Mit dem Quadtree reicht es aus, wenn wir die Bounding-Box eines Nodes des Baums gegen das View-Frustum testen. Wenn dieser Test ergibt, dass der Node nicht im View-Frustum liegt, dann können wir sofort auch alle Children dieses Nodes verwerfen. Diese können dann ebenfalls nicht mehr im Frustum liegen. Für unser Dorf bedeutet dies im Idealfall, dass der Spieler innerhalb eines Viertels des Dorfes steht und quasi in Richtung Dorfrand schaut. Dann sind bereits die drei Children des Root-Nodes, in denen sich der Spieler nicht befindet, außerhalb des Frustums. Somit können wir mit drei Bounding-Box-Tests drei Viertel der Fläche des Dorfes verwerfen und brauchen diese nicht weiter zu betrachten: genial einfach, und darum einfach genial.

Ausflug in die Grundlagen von Heightmaps

*Höheninforma-
tionen aus
Heightmaps*

Der Quadtree-Ansatz an sich ist nicht sonderlich spektakulär. Dass er dennoch so populär ist, liegt nicht nur an seiner Einfachheit und Eleganz. In der Abteilung Terrain-Rendering im Bereich 3D-Computergrafik ist der Quadtree auch deswegen so berühmt, weil etliche sehr bekannte Algorithmen aus Abhandlungen über das Terrain-Rendering einen Quadtree für das Scene-Management auf die eine oder andere Weise verwenden.¹ Bevor wir uns das anschauen, möchte ich kurz darauf eingehen, wie man überhaupt ein Terrain auf den Bildschirm bekommt. Dazu verwendet man in der Regel eine so genannte Heightmap. Dies ist ein zweidimensional zu interpretierendes Array, in dem Höhenwerte gespeichert sind. Im Folgenden ist zum Beispiel ein Array der Dimension 9x9 angeben:

```
UCHAR hMap[9*9] = { 1, 5, 6, 3, 4, 6, 8, 5, 6,
                    3, 4, 6, 5, 4, 5, 6, 4, 5,
                    2, 3, 5, 6, 5, 7, 8, 6, 4,
                    3, 5, 7, 4, 5, 8, 9, 7, 5,
                    4, 4, 5, 3, 4, 7, 8, 7, 6,
                    3, 5, 4, 2, 3, 6, 7, 6, 5,
                    2, 3, 4, 3, 5, 5, 6, 5, 4,
                    4, 4, 6, 4, 3, 4, 5, 4, 3,
                    3, 3, 4, 3, 4, 3, 3, 3, 2 };
```

*Interpretation der
Heightmap*

Eine solche Heightmap stellt die Informationen über die Höhe des Terrains an einem bestimmten Punkt zur Verfügung. An der Stelle $x=3$ und $z=5$ gibt die Heightmap beispielsweise eine Höhe von 2 an, wenn man jeweils bei 0 anfängt zu zählen. Dabei ist die Heightmap natürlich dimensionslos. Die Koordinaten (3,5) können in Fuß, Metern, Kilometern oder halben Gurken

¹ Vgl. <http://www.vterrain.org/LOD/Papers/index.html>

interpretiert werden. Selbiges gilt für den Höhenwert, den wir dort finden. Wenn wir nun ein Mesh für das Terrain erzeugen sollten, dann würden wir für die Ecke oben links im Array für das erste Rechteck die Vektoren $(0, 1, 0)$, $(1, 5, 0)$, $(0, 3, 1)$ und $(1, 4, 1)$ extrahieren. Für die gesamte Heightmap hätten wir also 8×8 Rechtecke und damit insgesamt 128 Dreiecke. Je größer die Abmessung des Terrains ist, desto mehr Dreiecke wird man im Endeffekt haben und desto mehr Arbeit hat die Grafikkarte beim Rendern. Wenn wir die Dimension der Heightmap als Meter annehmen und ein Terrain von zwei Quadratkilometern erstellen möchten, was für ein Fahrzeug nicht wirklich viel ist, dann hätten wir $(2000-1) \times (2000-1)$ Vierecke für das Mesh, also vier Millionen Vertices und sechs Millionen Indices für Dreiecke. Selbst wenn man ein intelligentes Frustum-Culling für Batches von Dreiecken einsetzt, bleiben immer noch viel zu viele Dreiecke im Bereich des View-Frustum übrig.

Eine solche Menge an Dreiecken in Echtzeit zu rendern ist nicht nur nicht möglich (vom heutigen technischen Stand aus gesehen), sondern es ist auch absolut unsinnig. Im Abschnitt über den Level of Detail hatte ich ja schon dargelegt, dass man diesen hohen Detailgrad über eine gewisse Entfernung sowieso optisch nicht wahrnehmen kann. Bei Terrain verfolgt man daher den Ansatz, dass man umso weniger Details für das Terrain-Mesh verwendet, je weiter weg das entsprechende Stück Terrain vom Betrachter ist. Das zweite Kriterium zur Vereinfachung ist das Ausmaß der Schwankung der Höhenwerte in der Heightmap. Man beachte den folgenden Ausschnitt aus einer Heightmap:

```
3, 4, 3, 4, 3, 4,
4, 3, 2, 2, 2, 3,
5, 3, 2, 2, 2, 4,
4, 3, 2, 2, 2, 3,
5, 3, 3, 3, 3, 3,
```

Hier fällt sofort auf, dass das Terrain in der Mitte des Ausschnitts auf einem Feld von drei Quadrateinheiten immer den Höhenwert 2 hat. Natürlich wäre es unsinnig, dort vier Rechtecke und acht Dreiecke zu erzeugen. Dort reichen ein einziges Viereck und zwei Dreiecke zur Darstellung aus, ohne dass eine Information verloren geht. Schon haben wir sechs Dreiecke gespart. Und es kommt sogar noch besser: Die angrenzenden Höhenwerte links und unterhalb dieses Stücks haben alle den Höhenwert 3. Wenn der Betrachter ausreichend weit von diesem Stück Terrain entfernt ist, kann er den Höhenunterschied von einer Einheit in dem Terrain gar nicht wahrnehmen. Wenn der Betrachter also weit genug entfernt ist, dann können wir auch diese Höhenwerte mit zu der Fläche der Höhe 2 zählen und spannen unsere beiden Dreiecke mit über diesen Bereich. Damit haben wir auch schon den ultrageheimen Kern der CLOD-Verfahren für das Terrain-Rendern aufgedeckt.

*CLOD-Scene-
Management für
Terrain-Meshs*

*Entfernung und
Fehlerwert*



Da das Mesh für das zu rendernde Terrain von der Entfernung des Betrachters abhängig ist, muss es immer dann neu erstellt werden, wenn sich der Betrachter signifikant bewegt hat. Wenn man sinnvollerweise nur die Teile des Meshs erstellt hat, die im View-Frustum liegen, dann muss man das Mesh auch dann neu erstellen, wenn der Betrachter nur rotiert, ohne sich zu bewegen.

Quadtrees für das Terrain

Quadtree für CLOD

Wir wissen jetzt, dass man das Mesh für ein zu renderndes Terrain abhängig von der Distanz des Betrachters und von der Differenz aneinander grenzender Höhenwerte in der Heightmap erstellt. Nun fragen wir uns natürlich, wie man das effizient implementieren kann? Und da wir uns immer noch im Abschnitt über den Quadtree befinden, ahnen wir schon, dass man einen Quadtree dafür verwenden kann. Wir nehmen also einen Root-Node eines Quadtrees und definieren diesen so, dass er sich einmal über das gesamte Terrain spannt. Dann machen wir den Kick Off für den Algorithmus, und dieser beginnt nun, sich immer wieder zu vierteln und damit immer jeweils nur noch ein Viertel der an dem Parent gespeicherten Heightmap mit einem Child zu umschließen. Und das machen wir genau so lange, bis sich ein Node über genau 3×3 Höhenwerte der Heightmap erstreckt. Dies sind dann die Leaves des Quadtrees. Für das obige Beispiel ist der Baum also recht klein. Nach der ersten Iteration haben die tiefsten Nodes je ein 5×5 -Stück der Heightmap erhalten. Nach der zweiten Iteration haben die Leaves der untersten Ebene dann schon nur noch 3×3 Höhenwerte. Ein solches Leaf mit 3×3 Höhenwerten nennt man im Kontext eines Terrain-Octrees normalerweise **Block**.

Der komplette Quadtree

Abbildung 13.4 zeigt uns, wie der fertige Quadtree aussieht. Wir blicken in dieser Darstellung quasi von oben auf den Quadtree und sehen in der Mitte die Wurzel mit dem Index 0. Diese Wurzel hat vier Children mit den Indices 1, 2, 3 und 4. Diese werden wiederum jeweils in vier Children unterteilt, die dann die Leaves des Baums sind, weil sie nur noch ein 3×3 Einheiten großes Stück der Heightmap umspannen. Die Indices in dem Quadtree kann man übrigens auch ganz einfach berechnen, wenn man zu einem gegebenen Node seinen Parent oder seine Children berechnen möchte. So kann man den Quadtree bequem in einem Array speichern und beliebig durch die Nodes und Children navigieren, ohne direkt in den entsprechenden Objekten zu sein und den dortigen Pointern folgen zu müssen:

```
P := Index des Parents
N := Index eines beliebigen Nodes
Upper Left Node =  $4 * P + 1$ 
Upper Right Node =  $4 * P + 2$ 
Lower Left Node =  $4 * P + 3$ 
Lower Right Node =  $4 * P + 4$ 
P von N =  $\text{floor}((N-1)/4)$ 
```

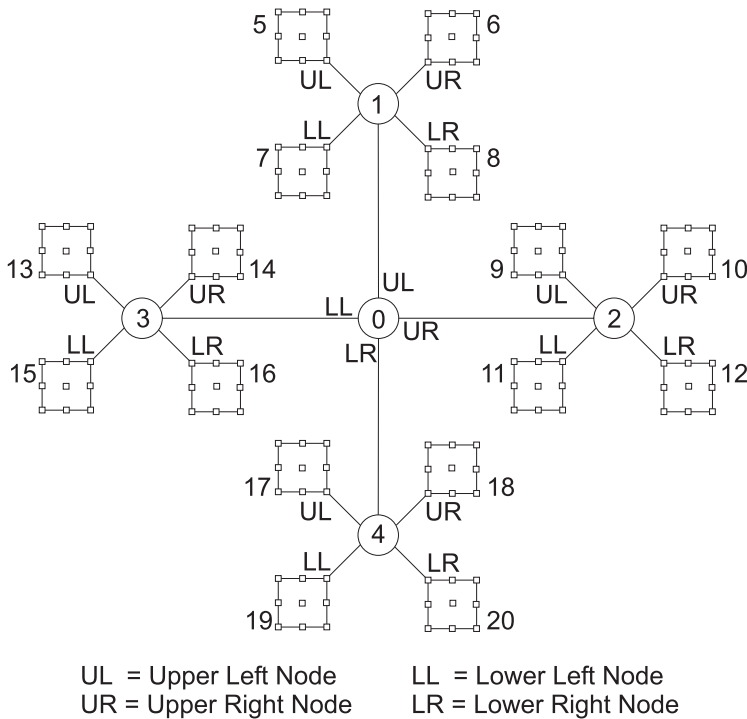


Abbildung 13.4: Schematische Darstellung eines vollständigen Quadtrees für eine 9x9-Heightmap. Die Kreise symbolisieren die Nodes des Quadtrees, die Rechtecke sind Leaves/Blöcke mit je 3x3 Vertices. Die Ziffern geben jeweils den Index der Nodes bzw. Leaves an.

Nun haben wir unsere Heightmap in einen Quadtree zerlegt. Doch was ist der Witz dabei, und wie bringt uns das eine bessere Performance? Das Geheimnis liegt hier in der Tatsache, dass man das Mesh in jedem Frame komplett neu aufbauen muss, wenn sich der Betrachter signifikant bewegt oder rotiert hat. Dabei geht man in die Wurzel des Quadtrees und wandert in die Children des Baums. Dabei nimmt man natürlich wie selbstverständlich ein Culling der Bounding-Boxen vor, die man um den jeweiligen Node berechnet hat. Doch bei den Nodes, die im Frustum liegen, wandert man nicht stur bis zu jedem Leaf durch und sammelt dort Vertices für das Mesh ein. Stattdessen überlegt man sich an jedem Node, ob man wirklich noch tiefer in seine Children abtauchen muss, sofern diese vorhanden sind. Damit sind wir bei der Frage gelandet, wie man entscheidet, ob man tiefer abtauchen muss. Sagen wir mal, wir stecken gerade in einem Node einer großen Heightmap, der ein 9*9 Einheiten großes Stück Heightmap umschließt. Darunter befinden sich noch zwei Ebenen im Baum mit 5*5 Einheiten großen Nodes und 3*3 Einheiten großen Leaves. Nun wissen wir jedoch, dass der Betrachter sehr weit weg von diesem Node ist. Durch eine Vorberechnungsstufe zur Initialisierungszeit des vollständigen Quadtrees wissen wir, dass die Höhenwerte in den zwei tieferen Ebenen an dieser Stelle vergleichsweise differieren. Das heißt, das Stück Terrain ist auf dieser betrachteten Fläche nur ein wenig *rau*. Diese minimalen Höhenunterschiede kann man aber aufgrund der großen Entfernung des Betrachters vernachlässigen, denn

Wo ist die Performance?

durch die Verkleinerung dieser Stelle des Meshs im Zuge der perspektivischen Projektion könnte man diesen Höhenunterschied optisch gar nicht mehr ausmachen.

*Die Such-Tiefe
eines Asts
bestimmt CLOD-
Auflösung*

Nun kommt, was sich schon jeder denken kann: Wir wandern einfach nicht tiefer in die Children dieses Nodes. Wir betrachten also nicht die Leaves mit 3×3 Vertices, sondern nur den 9×9 Einheiten der Heightmap umspannenden Node. Damit ist dieser ehemals innere Node des Baums jetzt für diesen Frame ein Leaf und wird als Block für das Terrain-Rendering betrachtet. Man berechnet also *on-the-fly* gerade so viele Details für das Terrain, wie man für angemessen hält. Die Heuristik, nach der man entscheidet, ob man noch tiefer in den Quadtree geht und dem Mesh damit mehr Details hinzufügt, kann sehr verschieden sein. Der trivialste Ansatz geht lediglich von der Entfernung eines potenziellen Blocks zu dem Betrachter aus. Andere Ansätze können beispielsweise auch die projizierte Größe des Blocks auf dem Bildschirm mit berücksichtigen.

Cracks im Terrain

Beim CLOD-Terrain-Rendering gibt es aber eine sehr unschöne Eigenheit, und zwar die so genannten **Cracks** (dt. Brüche) im Terrain. Diese entstehen immer dann, wenn ein Block aus einer bestimmten Ebene des Quadrees an einen anderen Block angrenzt, der von einer anderen Ebene, also einer anderen Zerlegungstiefe des Quadrees, stammt. Dies demonstriert Abbildung 13.5. Dort sieht man einen Streifen Terrain. Die rechte Hälfte besteht aufgrund geringer Höhenschwankungen in der zugrunde liegenden Heightmap nur aus vier Rechtecken mit je zwei Dreiecken. Dabei handelt es sich also um vier Nodes des Quadrees auf eine Ebene der Tiefe n . Die linke Hälfte des Terrains ist ein wenig hügeliger. Daher hat unsere Heuristik ganz eigenmächtig entschieden, dass der Quadtree bis auf die Ebene $n+1$ durchlaufen wird. Wir haben dort also nicht vier, sondern 4×4 Nodes, da ja alle vier Nodes der Ebene n vier Children haben. Man erkennt eindeutig, dass diese zusätzlichen Detailinformationen an der Schnittkante der beiden Hälften zu Cracks im Terrain führen.

*Aktivierbare Hilfs-
Vertices gegen
Cracks*

Glücklicherweise gibt es im Quadtree-Ansatz für das Terrain-Rendering ein einfaches Hilfsmittel gegen diese Cracks. Ein Block, also der Node des Quadrees, an dem man nicht weiter in die Children wandert, sondern den man rendert, enthält nicht nur vier Eck-Vertices, sondern insgesamt neun Vertices. Die zusätzlichen fünf Hilfs-Vertices – je einer auf den vier Kanten des Rechtecks und einer in der Mitte des Rechtecks – werden je nach Bedarf aktiviert oder deaktiviert. Wenn der Node, der an eine Kante dieses Nodes angrenzt, bis auf eine tiefere Ebene zerlegt ist, dann muss man den Hilfs-Vertex auf dieser Kante aktivieren. Ansonsten benötigt man ihn nicht. Dadurch behebt man den sonst an dieser Stelle entstehenden Crack, weil man einen Teil des eigentlich weniger detaillierten Blocks nachträglich auf dieselbe Ebene detailliert. Abbildung 13.6 zeigt die sieben möglichen Auflösungen, die ein Block einnehmen kann.

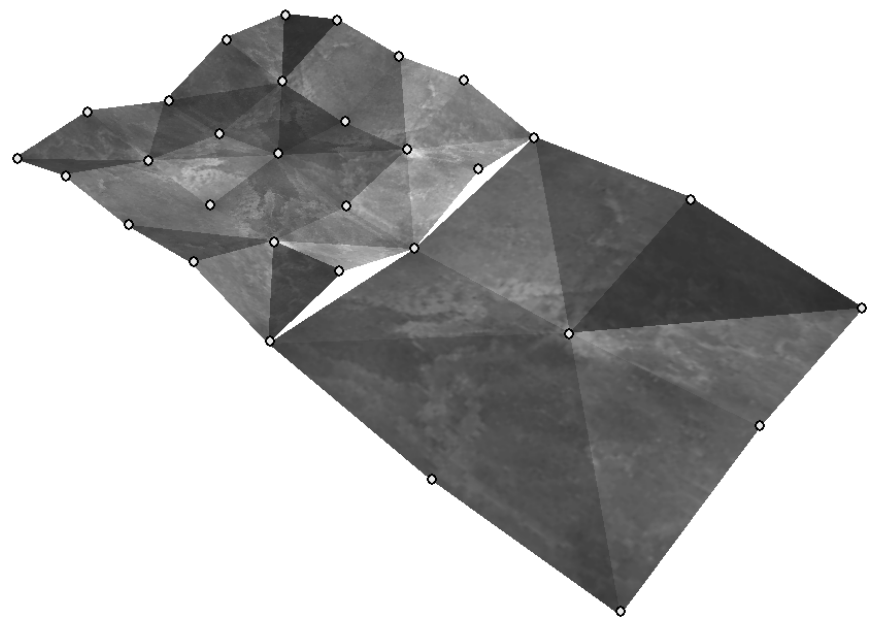


Abbildung 13.5:
Cracks im Terrain durch unterschiedliche Ebenen im Quadtree. In der Mitte erkennt man zwei Löcher, wo jeweils ein Node der Zerlegungstiefe n auf zwei Nodes der Zerlegungstiefe $n+1$ trifft.

Wenn alle Blöcke, die an die vier Kanten eines Blocks angrenzen, auf derselben Ebene zerlegt sind, dann muss keiner der Hilfs-Vertices aktiviert sein. Dies entspräche dem Typ »6« in Abbildung 13.6.

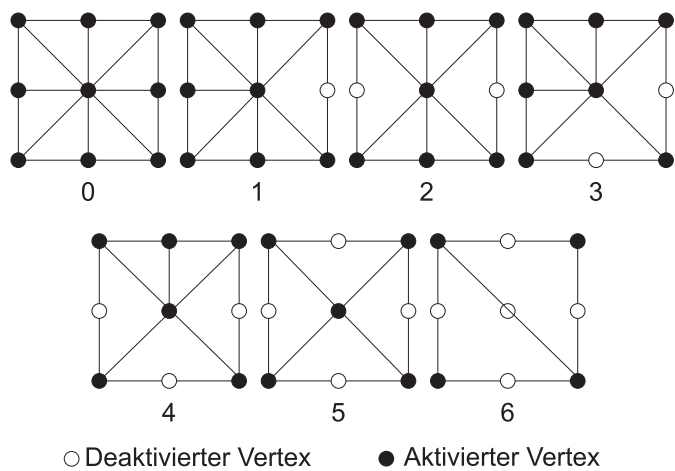
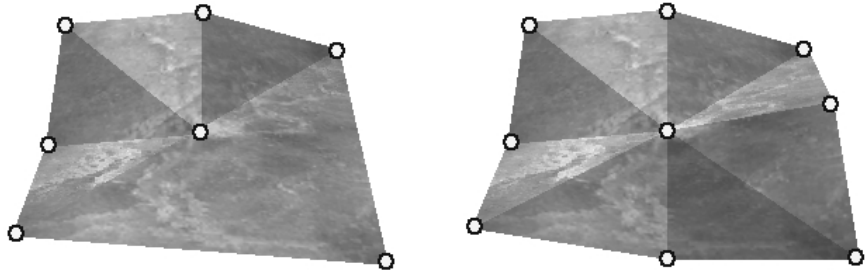


Abbildung 13.6:
Die sieben verschiedenen Typen der Auflösung eines Blocks aus dem Terrain-Quadtree und ihre Triangulationen. Symmetrische analoge Anordnungen sind nicht dargestellt und werden dem jeweiligen Typ zugeordnet.

Abbildung 13.7 zeigt als Beispiel zwei unterschiedliche Auflösungen desselben Blocks. Auf der linken Seite sieht man einen Block mit drei aktiven Hilfs-Vertices, und auf der rechten Seite sind alle fünf Hilfs-Vertices des Blocks aktiv.

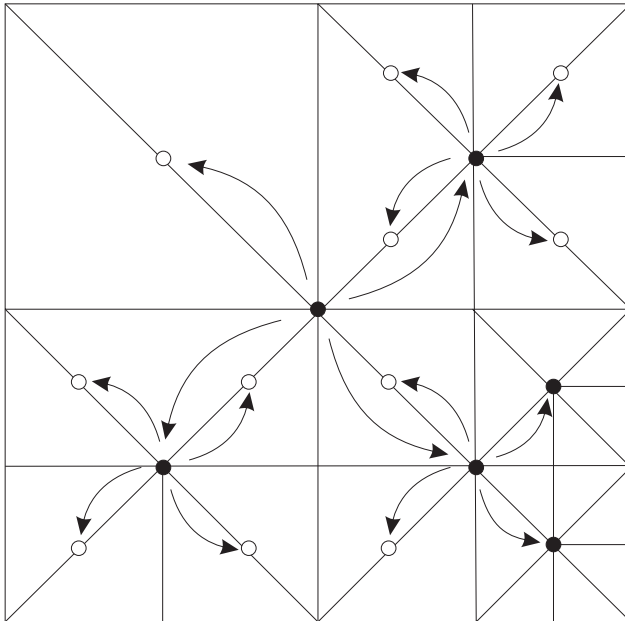
Abbildung 13.7:
Zwei unterschiedliche Auflösungen eines Blocks aus dem Terrain-Quadtree; rechts die höchste Auflösung vom Typ »0« und links eine Variante vom Typ »3«.



**Beispiel
Zerlegung**

Abbildung 13.8 zeigt ein Beispiel für die Zerlegung einer 9*9-Heightmap. Man kann sehr gut erkennen, wie die eingefügten Hilfs-Vertices dazu beitragen, dass nie eine Kante eines Dreiecks auf zwei Kanten von zwei anderen Dreiecken stößt. Dies würde zu Cracks im Terrain führen.

Abbildung 13.8:
Beispiel für die Zerlegung einer 9*9 Einheiten großen Heightmap. Die Pfeile deuten die Parent-Child-Beziehungen an. (Quelle: Röttger et al: »Real-Time Generation of CLOD for Height Fields«, Page 2, Figure 1)



**Weitere
Informationen**

Ich hätte gern noch ein kleines Beispiel für das Terrain-Rendering und für Quadrees in diesem Buch gebracht, doch leider reicht der Platz dazu nicht aus. Zudem liegt der Fokus ja auch auf den Indoor-Techniken. Weitere Informationen über das Terrain-Rendering findet ihr jedoch im Internet auf der Seite www.vterrain.org/LOD. Dort sind sowohl die älteren und neuen Papers als auch Implementierungen diverser Techniken mit Source Code verlinkt. Einen besseren Startpunkt für das Terrain-Rendering wird man nicht finden.

Octrees

Nun ist es für die Quadrees an der Zeit, ihr Schattendasein im zweidimensionalen Raum zu verlassen und in die dritte Dimension vorzustoßen. Dasselbe, was man im 2D-Raum mit Bounding-Rectangles gemacht hat, lässt sich im 3D-Raum mit Bounding-Boxen durchführen. An dieser Stelle brauche ich gar nicht mehr so viel zu erklären, denn der Algorithmus ist ebenso wie der Sinn und Zweck des Verfahrens derselbe wie bei dem allgemeinen Quadtree. Man legt im 3D-Raum einen Würfel als Bounding-Box um die gesamte Geometrie. In der ersten Iteration teilt man den Würfel dann auch in gleich große Unter-Würfel auf. Logischerweise ergeben sich dabei genau acht Würfel, die den Raum des ursprünglichen Würfels genau ausfüllen – daher der Name Octree.

Quadrees in der dritten Dimension

Bei jeder weiteren Iteration nimmt man nun die acht Child-Nodes und unterteilt sie bei Bedarf wieder in acht kleine Würfel. Das macht man so lange, bis eine entsprechende Heuristik anzeigt, dass man einen Node nicht weiter zu unterteilen braucht. Das Abbruchkriterium kann hier eine bestimmte Anzahl an Polygonen in einem Leaf sein oder auch eine bestimmte maximale Tiefe, die ein Leaf in dem Baum haben darf. Dafür gibt es keine fest vorgegebenen Regeln. Abbildung 13.9 zeigt einen kleinen Ausschnitt einer Level-Geometrie und visualisiert den dafür erzeugten Octree. Das Bild ist ein Screenshot aus der Demo-Applikation zu diesem Kapitel.

Algorithmus

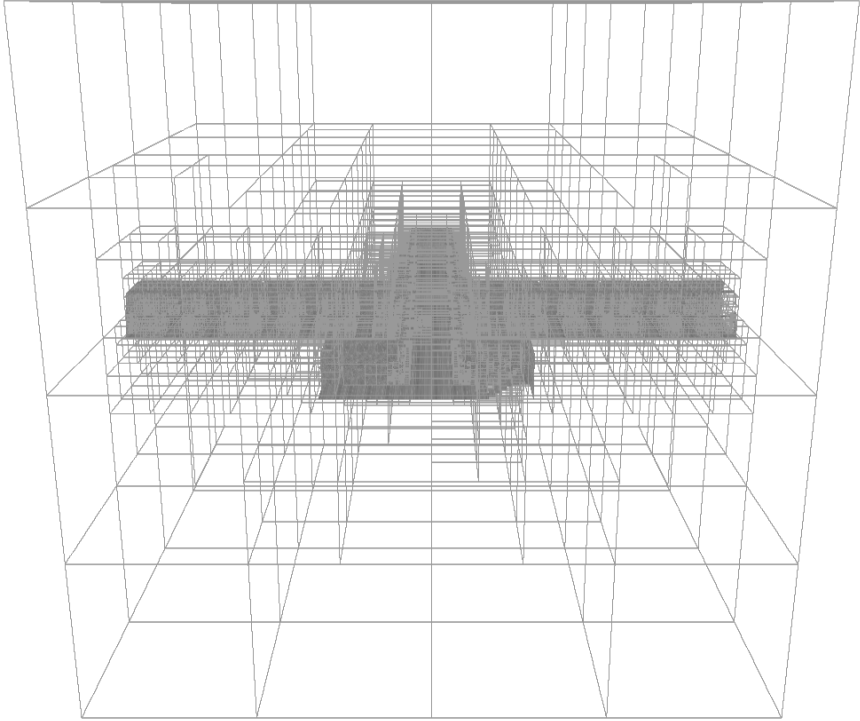
Nun kommen wir doch noch zu einer interessanten Thematik. Wie verfahren wir mit Polygonen der ursprünglichen Geometrie, die nicht vollständig in einen Node hineinpassen? Dazu gibt es zwei Lösungsansätze. Entweder teilt man das ursprüngliche Polygon in zwei kleinere auf, die sich genau auf der Grenze der beiden entsprechenden Nodes berühren, in denen sie liegen. Die andere Methode besteht darin, diese Polygone nur demjenigen Node zuzuordnen, in dem ihr Mittelpunkt liegt. Die erstgenannte Methode erzeugt eine Menge Splits in der ursprünglichen Menge an Polygonen und führt zu einer sehr großen Zahl von Objekten. Die zweite Methode hat den Nachteil, dass man immer darauf achten muss, dass in einen Node auch Polygone aus den Nachbar-Nodes hineinragen können. Zudem muss man ein Polygon auch mit einem Flag belegen, wenn man es für eine bestimmte Operation bereits einmal verwendet hat (Rendern, Kollisionsabfrage usw.), damit es durch einen Nachbar-Node nicht nochmals zu dieser Operation im selben Frame verwendet wird.

Aufteilung schneidender Polygone

Welche der beiden Methoden man wählt, hängt ganz entscheidend von dem Ziel ab, das man mit dem Octree erreichen will. Wir werden am Ende dieses Kapitels einen Octree implementieren und die Methode des Beschneidens von Polygonen verwenden. Das macht uns die Arbeit mit dem Octree nachher um einiges leichter.

Abbildung 13.9:

Ein visualisierter Octree für ein kleines Stück Geometrie. Man erkennt deutlich die hohe Zerlegungstiefe des Baums in Regionen mit viel Geometrie.



***Bewegte Objekte
in Raumaufteilungs-
verfahren***

Ein letzter Punkt, den ich hier noch erwähnen muss, ist die Behandlung von dynamischen Objekten, also solchen Objekten, die sich bewegen können. Implizit konnte man bereits aus dem Quadtree- und Octree-Verfahren herauslesen, dass diese sich nur für statische Geometrie eignen, die sich nicht bewegt. Denn wenn sich ein Teil der Geometrie bewegt, dann müsste man ja die gesamte Raumaufteilung neu berechnen. Bewegte Objekte kann man nun losgelöst von dem Baum betrachten und den Baum dennoch zu Hilfe nehmen. Bei der initialen Aufteilung des Raums berücksichtigt man solche bewegten Objekte (wie beispielsweise Monster) einfach gar nicht. Nun speichert man eine Liste von bewegten Objekten in dem Baum. Dazu gibt es auch wieder zwei Strategien: Entweder speichert man diese Listen nur in den Leaves des Baums, oder man erlaubt eine Liste an jedem Node des Baums. Dann sortiert man die bewegten Objekte zu Beginn des Programms in den Baum ein: nach der ersten Strategie in dem Leaf, in dem ihr Mittelpunkt liegt, und nach der zweiten Strategie in den Node, in den ihre Bounding-Box vollständig hinein passt. Wenn sich ein Objekt nun bewegt, dann prüft man einfach, ob seine Bounding-Box bzw. sein Mittelpunkt die Grenzen des Nodes bzw. des Leafs überschritten hat, und sortiert es entsprechend um.

***Wie viel Octree
darf es denn sein?***

Nun haben wir in dem Octree auch Informationen über die bewegten Objekte gespeichert. So können wir den Octree nicht nur für Kollisionen

von Objekten mit der statischen Level-Geometrie verwenden, sondern auch für Kollisionen mit bewegten Objekten. Es ist also uns überlassen, für welche Zwecke wir den Octree einsetzen möchten. Machbar ist mit diesem Ansatz eigentlich fast alles, was man sich im Bereich des 3D-Scene-Managements wünschen kann. Mehr gibt es zu einem Octree nicht zu sagen, und daher brechen wir das Thema an dieser Stelle ab. Auf all die kleinen Details werden wir am Ende des Kapitels eingehen, wenn wir unsere Demo-Applikation entwickeln und einen Octree implementieren.

Interessanterweise gibt es sogar US-Patente auf Scene-Management-Algorithmen und die zugehörigen Datenstrukturen. Die Anwendung von Quad-trees zur Raumaufteilung zwecks Hidden Surface Removal findet sich unter der US-Patentnummer 3602702. Das US-Patent 4694404 hingegen hat die Beschleunigung der Darstellung solider 3D-Objekte durch eine hierarchische Unterteilung mittels eines Octrees zum Inhalt. Daneben gibt es noch zahlreiche weitere Patente, die sich mit spezialisierten Anwendungsmöglichkeiten dieser und anderer Techniken befassen.

(Quelle: <http://www.uspto.gov/patft/index.html>)

Binary Space Partitioning Trees

Der eben betrachtete Octree dient dazu, den Raum gleichmäßig und achsen-ausgerichtet aufzuteilen. Daher kann man die achsen-ausgerichteten Bounding-Boxen (AABB) sehr gut in einem Octree verwenden, und mit einer umfangreichen Klasse für solche AABB hat man quasi schon den halben Octree fertig. Die Binary Space Partitioning-(BSP-)Bäume sind nun eine Art generalisierter Ansatz zur Raumaufteilung. Während der Octree den Raum immer gleichmäßig entlang den Standardachsen aufteilt, kann man mit einem vollwertigen BSP-Baum den Raum an beliebigen Ebenen aufteilen. Die Unterteilung der Geometrie richtet sich also nicht nach den Standardachsen, die keinen Bezug zu der Geometrie haben, sondern direkt nach der Geometrie. Der offensichtliche Vorteil dabei ist, dass man nicht so viele Polygone zerschneiden muss, wie das bei dem Octree der Fall ist.

Auch die BSP-Bäume können dazu verwendet werden, den Raum anhand der Weltachsen aufzuteilen. Sie sind dann die achsen-ausgerichtete Variante des BSP-Algorithmus. In der Regel wird man aber die polygon-orientierte Variante bevorzugen, die wir im Folgenden auch explizit behandeln werden.

Bevor wir dazu kommen, was ein BSP-Baum überhaupt ist, betrachten wir ein wenig die Geschichte und wie der BSP-Baum zur Wunderwaffe (fast) aller 3D-Grafik-Programmierer wurde. Unbestrittenerweise zählen die Spiele der US-Firma *id Software* wie beispielsweise *Doom*, *Quake* und *Castle Wolfenstein* zu den wohl aufsehenerregendsten interaktiven 3D-Grafikprogrammen, die es zu ihrer jeweiligen Zeit gab. Und auch der Name



**Generalisierte
Raumaufteilung**



**Siegeszug der
BSP-Bäume**

John Carmack, des Technical Directors und Engine-Programmierers der Firma, hat seinen festen Platz in den Annalen der Computerspiele. Das hat den Grund, dass die Spiele dieser Firma immer gewisse Dinge bieten konnten, bevor andere Firmen überhaupt daran glaubten, dass solche Dinge in Echtzeit möglich wären. Dies fing mit den ersten First-Person-Shootern überhaupt an, ging weiter über echte 3D-Grafik mit freiem Sichtwinkel² und hin zu echten Modellen aus Polygonen für alle Charaktere und Objekte³ im Spiel in vergleichsweise riesigen virtuellen Welten.

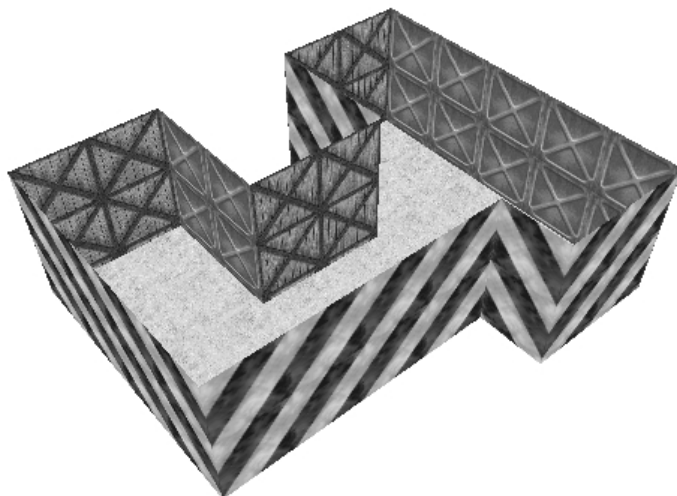
Carmack = BSP-Tree Als bekannt wurde, dass das Geheimnis hinter der guten Performance der Spiele von *id Software* ein BSP-Baum war, ging ein Aufschrei durch die Programmierer-Welt. Es handelte sich um eine in der Informatik durchaus schon lange Zeit bekannte Datenstruktur. Seither sind nahezu alle angehenden Spiele-Programmierer so auf BSP-Bäume fixiert, dass es eine gewisse Zeit dauern wird, die Köpfe des Nachwuchses auf die neue Generation der Hardware auszurichten. Einen BSP-Baum zum Rendern zu verwenden ist heutzutage nämlich nicht mehr so sinnvoll wie noch vor einigen Jahren. Vielleicht hilft es ja zu wissen, dass John Carmack den BSP-Baum für das Rendern bei der *Doom 3*-Engine aufgegeben hat. Aber schauen wir uns jetzt erst einmal an, wie dieser Algorithmus in Bezug auf Computergrafik funktioniert.

Einsatz des BSP Die magischen drei Buchstaben BSP stehen für *Binary Space Partitioning*, was etwa so viel heißt wie »binäre Unterteilung des Raums«. Der BSP-Algorithmus sorgt also dafür, dass ein Raum binär unterteilt wird. Der BSP-Baum hingegen dient dazu, diese Unterteilung in einer sinnvollen Datenstruktur zu speichern, während sich der Algorithmus unaufhaltsam durch den zu teilenden Raum arbeitet. Dieses Verfahren kann mal sowohl für zweidimensionale als auch für dreidimensionale Räume anwenden, aber wir konzentrieren uns hier auf den dreidimensionalen Fall. Der zweidimensionale Fall ist ganz analog, nur eben ein wenig einfacher. Werfen wir zunächst einen Blick auf Abbildung 13.10, die uns einen Mini-Level im dreidimensionalen Raum zeigt, mit dem wir im Folgenden arbeiten werden. Zur besseren Übersichtlichkeit habe ich die Deckenpolygone hier entfernt und die normalerweise unsichtbaren Rückseiten der Wände mit einer diagonal gestreiften Textur überzogen, um sie sichtbar zu machen.

Ziel des BSP-Algorithmus Das erklärte Ziel des BSP-Algorithmus ist es, die Geometrie im Raum so aufzuteilen, dass man sie zu *konvexen* Untereinheiten gruppiert. Abbildung 13.11 zeigt auf der linken Seite ein konvexes Polygon und auf der rechten Seite ein konkaves Polygon. Ich denke, ihr seht, worauf das hinausläuft.

2 Die ersten 3D-Shooter waren aufgrund einfacherer Berechnungen mehr oder weniger nur 2,5D, denn die Kamera konnte nicht auf der X- oder Z-Achse, also hoch/runter bzw. rechts/links gekippt werden, sondern nur nach rechts oder links gedreht.

3 Bei früheren Spielen verwendete man dazu noch animierte 2D-Grafiken als Billboard-Sprites, um Rechenleistung zu sparen.

**Abbildung 13.10:**

Ein kleiner Beispiel-Level, dem wir mit dem BSP-Algorithmus zu Leibe rücken werden. Man beachte, dass die nach außen weisenden Polygone mit der quer gestreiften Textur hier nur zur besseren Visualisierung verwendet werden. Sie wären kein Bestandteil echter Level-Daten!

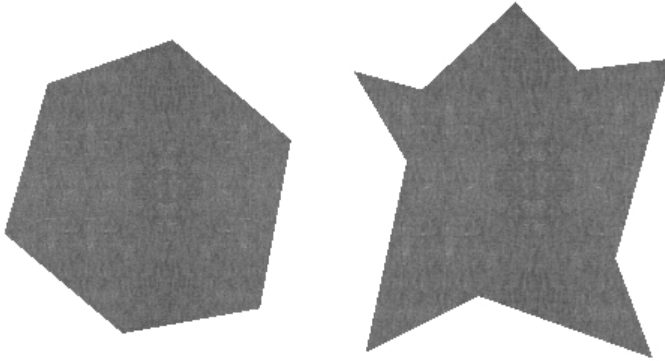
Ein konvexes Polygon ist, simpel ausgedrückt, ein Polygon, das keine Delle hat. Das konkave Polygon hingegen hat auch Innenwinkel, die größer oder gleich 180° sind und daher wie Dellen im Polygon wirken. Diese Definitionen kann man auch auf den 3D-Raum übertragen. Eine Anordnung von Polygonen im 3D-Raum ist dann konvex, wenn man von jeder Stelle innerhalb dieses Raums aus Polygonen in jede beliebige Richtung schauen kann und kein Polygon ein anderes auch nur teilweise verdeckt. In einem konkaven Stück Level-Geometrie kann es sehr wohl zu Überdeckungen kommen. Jede Delle in dem Raum sorgt dafür, dass es dahinter einen Punkt gibt, in den man sich zwängen kann und wo die in den Raum ragende Delle dann andere Polygone verdeckt. Oh, mein Mathe-Prof würde mich für diese legeren Definitionen loben! Aber so sieht man auch gleich, warum der BSP-Algorithmus historisch gesehen so nützlich war: Durch diese Aufteilung der Geometrie erhielt man immer kleine Pakete von Polygonen, die man auch ohne Einsatz eines Depth-Buffers rendern konnte. Man wusste ja genau, dass sie – egal aus welchem Winkel und von welcher Position aus man sie betrachtet – sich nicht gegenseitig verdecken können. Nun brauchte man alle konvexen Polygon-Pakete, die der BSP-Algorithmus berechnet und im BSP-Baum gespeichert hatte, nur noch nach ihrer Distanz zu sortieren und konnte sie dann von hinten nach vorne rendern. Aber in Wirklichkeit war es noch besser: Diese Pakete waren in dem BSP-Baum schon sortiert. Man konnte also eine gesamte Szene vollkommen ohne Depth-Buffer rendern und hatte trotzdem keine Pixelfehler.

Beim Rendern ohne Depth-Buffer und beim Rendern von (teil-) transparenten Polygonen muss man auch heute noch Back-To-Front rendern. Das bedeutet, man rendert die in der Szene am weitesten entfernt gelegenen Polygone zuerst und danach die näher gelegenen. Auf modernen Grafikkarten mit einem Depth-Buffer ist es aber wesentlich effizienter, diesen auch zu

!!
STOP

Abbildung 13.11:

Links ein konvexes
Polygon und rechts
ein konkaves
Polygon



benutzen und Front-To-Back zu rendern – also die nächstgelegenen Polygone zuerst. Diese bevölkern dann den Depth-Buffer und sorgen durch Early-Z-Culling dafür, dass die weiter entfernten Polygone bzw. Pixel gar nicht erst in den Frame-Buffer gerendert werden, nur um dann wieder überschrieben zu werden. Man verhindert also dadurch den so genannten Overdraw.

BSP-Algorithmus

BSP-Algorithmus

Nun haben wir schon so viel geschwätzt und sind eigentlich noch kein Stück weiter, was den BSP-Algorithmus angeht. Wir werden jetzt kurz an einem Beispiel durchsprechen, wie er genau funktioniert, und dann ein wenig code-orientierter zur Sache gehen. Das »B« in der Abkürzung steht ja für »binär«. Binäre Bäume an sich nun nichts Spektakuläres. Dabei handelt es sich einfach um Bäume, die an jedem Node genau zwei Kinder hängen haben. An der Wurzel des BSP-Baums hat man den gesamten 3D-Raum erfasst. Es ist ja ein offenes Geheimnis, dass man in einem BSP-Baum die Geometrie eines Levels speichert, also sagen wir gleich, dass man an der Wurzel eines BSP-Baums damit beginnt, den gesamten Level zu betrachten. Nun muss man für die Wurzel zwei Kinder erzeugen, also den Level in zwei Teile zerlegen. Das Ziel des BSP-Baums sind aber konvexe Pakete von Polygonen. Wie zerlegt man also den Level aus Abbildung 13.10 so, dass wir am Ende eines jeden Astes des BSP-Baums einen konvexen Teil des Levels hängen haben?

Splitter und Partitionierungsebene

Der BSP-Algorithmus geht dazu nach einem ebenso einfachen wie genialen Prinzip vor: Er wählt einfach ein einzelnes Polygon aus dem Level aus, das man als so genannten *Splitter* bezeichnet, und verwendet dessen Ebene als Partitionierungsebene im Raum. Alle Polygone, die hinter dieser Ebene liegen, also auf der Seite, von der der Normalenvektor der Eben wegzeigt, kommen in einen Ast des BSP-Baums, und die Polygone, die vor der Teilungsebene liegen, kommen in den anderen Ast des BSP-Baums. Wenn ein Polygon die Teilungsebene schneidet, dann wird es an der Schnittkante geteilt, und die beiden Teile werden in den entsprechenden Ast einsortiert.

Die beiden Äste bezeichnet man auch als *Frontlist* und als *Backlist*. Sollte nun ein Polygon genau in der Ebene liegen, was natürlich auf alle Fälle für den Splitter der Fall ist, so prüft man, in welche Richtung sein Normalenvektor schaut. Wenn er in die gleiche Richtung wie die Teilungsebene schaut, dann kommt er in die Frontlist (so wie z.B. der Splitter) und andernfalls in die Backlist. Und das ist schon fast der ganze Algorithmus. Abbildung 13.12 zeigt die erste Iteration für unser kleines Beispiel. Als Splitter wurde hier das Wand-Polygon auf der linken Seite des schmalen Gangs gewählt, das nun genau in der Mitte relativ frei dasteht. Der L-förmige Teil im rechten unteren Teil des Bildes enthält alle Polygone der Frontlist. In die Backlist kommen alle anderen Polygone, die auf der Rückseite des Splitters liegen. Hier sind das zwei fast quadratische, nicht miteinander verbundene Teile.

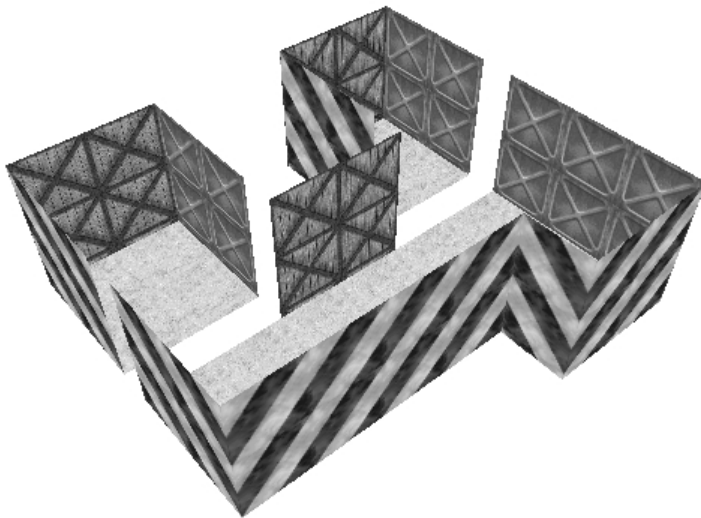


Abbildung 13.12:
Hier wurde die erste Iteration des BSP-Algorithmus auf den Level angewendet. Als Splitter wurde das unscheinbare Polygon in der Mitte gewählt.

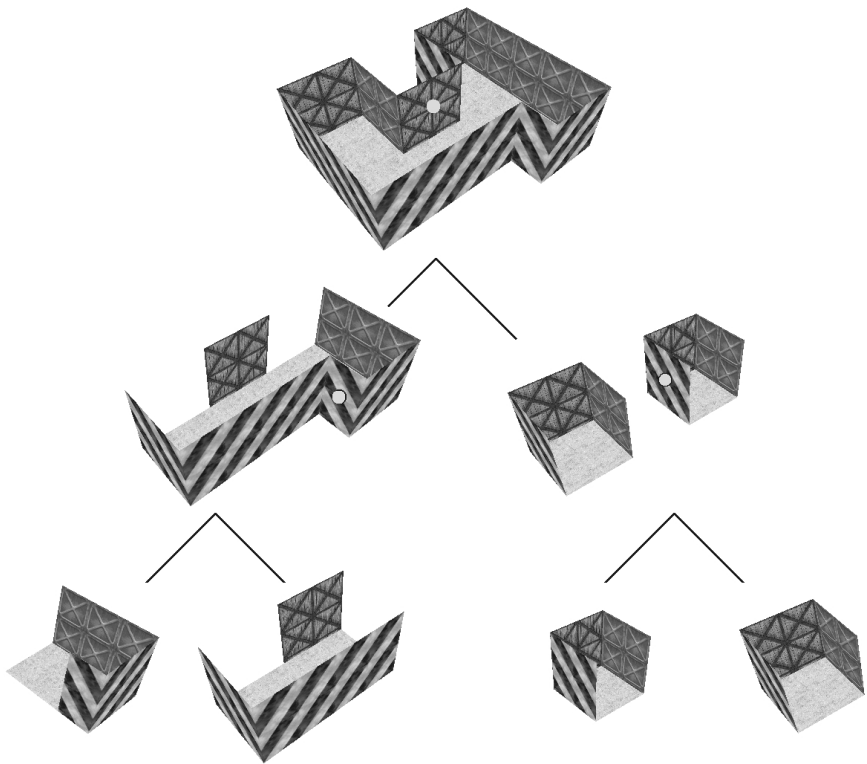
Dasselbe Spielchen geht immer so weiter. Die beiden Listen im BSP-Baum werden jeweils unabhängig voneinander weiter aufgeteilt, und zwar so lange, wie sich noch ein Splitter finden lässt. Das ist der Trick dabei, um die Konvexität zu erzielen: Ein Polygon kann nur zu einem Splitter werden, wenn mindestens je ein Polygon auf der Frontseite und ein Polygon auf der Backseite seiner Ebene liegt. Wenn man ein solches Polygon nicht mehr findet, dann hat man die Geometrie in diesem Ast so weit aufgeteilt, dass sie konvex ist. Alle Polygone an diesem Node des Baums sind damit so angeordnet, dass sie sich jeweils gegenseitig nur die Frontseite zudrehen. Abbildung 13.13 zeigt die absolut komplette Weiterführung des BSP-Algorithmus für unser Beispiel. Ganz oben starten wir mit dem gesamten Level. An jedem Node wird die dort angekommene Geometrie dann jeweils in eine Frontlist (linker Ast) und eine Backlist (rechter Ast) zerlegt. Das Polygon, das an einem Node als Splitter ausgewählt wird, ist immer mit einem hellen Punkt

Mach nur so weiter!

markiert. Zufällig haben wir hier einen ausgewogenen Baum erhalten, bei dem alle Äste die gleiche Länge haben. Das muss aber nicht so sein. Auf der untersten Stufe dieses Beispiels haben wir vier Leaves des BSP-Baums, die jeweils einen konvexen Happen des ursprünglichen Levels enthalten. Dort kann man keine Ebene mehr finden, die den Raum weiter aufteilen würde. Außerdem gibt es noch ein interessantes Detail: Nach der ersten Aufteilung haben wir in der Backlist zwei nicht miteinander verknüpfte Mengen an Polygonen, nämlich die beiden kleinen, fast quadratischen Räume. Es ist nicht notwendig, dass die Geometrie in einer Liste zusammenhängend ist, wobei sie das in den Leaves i.d.R. sein wird. Trotzdem gehören alle Polygone dort in dieselbe Liste, die weiter zerlegt wird.

Abbildung 13.13:

Der vollständige BSP-Baum für den Level. Die Front-Polygone wandern jeweils in den linken Ast, die Back-Polygone in den rechten Ast. Der an jedem Node ausgewählte Splitter ist mit einem hellen Punkt markiert.



Geschichte der BSP-Bäume und ihr Weg in die Computerspiele

Der Algorithmus, der in einem BSP-Baum mündet, wurde ursprünglich Ende der siebziger Jahre an der University of Texas in Dallas entwickelt. Dort waren Henry Fuchs und der als Theoretiker stigmatisierte Zvi Kedem als Professoren tätig. Sie stießen auf den Studenten Bruce Naylor, der nach einem Thema für seine PhD Thesis suchte. Der Rest ist Geschichte, und die Zusammenarbeit der drei mündete in zwei SIGGRAPH-Papers und Naylors PhD-Dissertation:

- Fuchs, Kedem, Naylor: »Predetermining Visibility Priority in 3-D Scenes«, SIGGRAPH '79, pp 175-181.
- Fuchs, Kedem, Naylor: »On Visible Surface Generation by A Priori Tree Structures«, SIGGRAPH '80, pp124-133.
- Bruce F. Naylor: »A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes«, PhD Thesis, University of Texas at Dallas, Mai 1981.

Im Dezember 1993 wurde dann das erste Spiel veröffentlicht, das einen BSP-Baum einsetzte, um Performance zu gewinnen. Das Spiel des Grafikers Adrian Carmack, des Game-Designers Tom Hall und der Programmierer John Romero und eines gewissen John Carmack war natürlich Doom [1]. Die beiden Carmacks des Teams sind übrigens nicht miteinander verwandt.



Abbildung 13.14:
Doom [1] von id
Software

Die diversen Vorgänger von *Doom*, die das Team schon entwickelt hatte, z. B. *Hovortank* und *Wolfenstein 3D*, waren an sich schon kleine Sensationen für die damalige Zeit. Sie definierten jedoch einige nicht unerhebliche Restriktionen für das Design eines Levels. Bei *Wolfenstein 3D* mussten alle Räume eines Levels dieselbe Höhe für Böden und dieselbe Höhe für Decken haben. Des Weiteren mussten alle Wände in 90-Grad-Winkeln zueinander und zu den Decken und Wänden stehen und vertikal sein. Folglich mussten alle Wände und Böden horizontal sein. Für *Doom* wollte man diese Beschränkung aufheben, daher benötigte man eine neue Methode, um die Entfernung verdeckter Flächen (engl. hidden surface removal) schnell zu ermöglichen. Die oben genannten Restriktionen erleichterten diese Aufgabe bei *Wolfenstein 3D* ungemein, weil viele allgemeine und daher komplexe Fälle von Anordnungen der Geometrie im 3D-Raum in den Berechnungen einfach untersagt waren.

John Carmack stieß dabei auf den BSP-Baum, der bereits in den *Bell Labs* für die Darstellung von 3D-Objekten verwendet wurde. Für Echtzeit-Anwendungen im 3D-Raum war diese Methode aber auf der damaligen Hardware viel zu langsam.

Abbildung 13.15:
Wolfenstein 3D von
id Software



Daher verwendet *Doom* in der Tat nur einen zweidimensionalen BSP-Baum. Die Wände in *Doom* konnten so in einem beliebigen Winkel zueinander stehen, und die Böden und Decken der Räume konnten unterschiedliche Höhen haben. Die Wände müssen zwar weiterhin vertikal und die Böden und Decken horizontal sein, aber das war bereits ein großer Fortschritt. Von oben betrachtet, erscheinen so alle Wände des Levels als zweidimensionale Linien. Die Böden und Decken sind weiterhin nicht von Bedeutung, da sie garantiert horizontal liegen und die Level-Geometrie ihres Raums folglich nicht aufteilen.

Das Splitten von zweidimensionalen Linien im 2D-Raum war damals in der Tat schon in Echtzeit möglich, im Gegensatz zum Splitten von Polygonen im 3D-Raum. Spiele wie *Doom* verwenden daher einen zweidimensionalen BSP-Baum, der quasi nur Linien splittet und sortiert. Daher bezeichnet man solche Spiele auch als 2,5D, weil alle Berechnungen im 2D-Raum stattfinden, das Spiel aber nach 3D aussieht.

Abbildung 13.16:
Quake [1] von id
Software



Der nächste Evolutionsschritt war das im Juni 1996 veröffentlichte Spiel *Quake* [1], das ebenfalls von demselben Team, also *id Software*, ist. Dieses Spiel verwendet auch einen BSP-Baum, allerdings nun einen BSP-Baum im 3D-Raum. Damit war es erstmals möglich, beliebige dreidimensionale Geometrie ohne Restriktionen in einem Level zu verwenden.

Seither ist der BSP-Baum eine feste Größe in nahezu allen 3D-Spielen. Auch wenn BSP-Bäume nun dank schneller Grafikkarte zum Rendern nicht mehr nötig sind, werden sie doch insbesondere in der Kollisionsabfrage nach wie vor verwendet.

Den besten Splitter finden

Der BSP-Algorithmus ist also nicht sonderlich schwierig. Im Gegenteil: Wie bei allen Algorithmen, die dazu dienen, eine baumartige Datenstruktur zu erstellen, ist die Umsetzung des Ganzen in Quellcode sehr einfach, denn man braucht i.d.R. nur einen einzigen rekursiven Funktionsaufruf, der den gesamten Baum erstellt. Aber eine Frage sollte uns nun noch quälen: Wir haben gehört, dass der Algorithmus an jedem Node des Baums ein Polygon aus der Level-Geometrie als Splitter auswählen möchte, um alle anderen Polygone an diesem Node in Relation zu der Ebene des Splitters auf die beiden Children des Nodes aufzuteilen. Nun stellt sich die Frage, welche Konditionen ein solches Polygon erfüllen muss, um als Splitter ausgewählt zu werden? Die überraschende Antwort darauf lautet: Es gibt keine Bedingung, die erfüllt sein muss. Man kann jedes beliebige Polygon auswählen, so lange wenigstens ein anderes Polygon auf seiner Vorderseite und auf seiner Rückseite liegt bzw. ein Polygon entsprechend zerschnitten wird.

Wie wählt man ein Polygon als Splitter aus dem Level aus?

Okay, okay. Ich will den Algorithmus hier nicht einfacher reden, als er ist. Aber es gibt eben keine Heuristik, nach der man den absolut besten Splitter auswählt. Allerdings gibt es verschiedene Ziele, die man durch den BSP-Algorithmus noch erreichen will, und die kann man bei der Auswahl des Splitters berücksichtigen. Das erste Ziel ist, dass der generierte BSP-Baum möglichst ausgewogen ist. Das bedeutet: Die Äste sollten alle nahezu gleich lang sein. Dadurch erreicht man, dass die Suche innerhalb des Baums immer etwa gleich lange dauert, und nicht mal sehr schnell und dann wieder sehr langsam vonstatten geht, je nachdem wie lang der Ast ist, in dem man suchen muss. Dieses Kriterium kann man erfüllen, indem man den Splitter so auswählt, dass er die Menge an Polygonen möglichst in zwei gleich große Teile aufspaltet. Das zweite bedeutende Kriterium für die Qualität eines Splitters ist die Anzahl der Polygone, die er wirklich splittet, also diejenigen Polygone, die seine Ebene schneiden und entsprechend zerlegt werden müssen. Jede dieser Zerlegungen (eines konvexen Polygons) erzeugt zwei neue Polygone. Die Menge der Daten steigt also an. Das ist heutzutage nicht mehr so kritisch, es sei denn, man möchte den BSP-Baum wirklich zum Rendern verwenden.

So einfach kann es doch nicht sein?

*Heuristik zur
Auswahl des
Splitters*

Aus diesen beiden Kriterien kann man nun eine Heuristik kombinieren, die für jedes Polygon an einem Node einen Punktwert vergibt. Das Polygon mit den wenigsten *Miesen* macht das Rennen und wird der Splitter an diesem Node. Eine solche Heuristik könnte wie folgt aussehen:

$$\text{Score} = \text{abs}(\text{NumFront} - \text{NumBack}) + (\text{NumSplits} * 3)$$

Dabei bezeichnen `NumFront` und `NumBack` die Anzahl an Polygonen, die jeweils vor bzw. hinter dem als Splitter betrachteten Polygon liegen; und `NumSplits` ist die Anzahl an Polygon-Zerschneidungen, die notwendig werden, wenn man diesen Splitter verwendet. Die Konstante 3 kann man noch weiter erhöhen. Sie gibt die Gewichtung an, mit der die Splits in den Punktwert eingehen. Je höher dieser Wert ist, desto schlechter schneidet ein Polygon als Splitter ab, wenn es zu vielen Splits führen würde.

Mit dieser Heuristik kann man an jedem Node die Liste von Polygonen durchlaufen. Für jedes Polygon durchläuft man dann in einer inneren Schleife alle anderen Polygone aus der Liste und klassifiziert sie. Dabei zählt man mit, wie viele Polygone jeweils in die Frontlist und Backlist fallen und wie viele gesplittet werden müssen. Danach vergibt man den Punktwert und merkt sich den Splitter mit dem niedrigsten Punktwert. Diesen wählt man dann aus und verwendet ihn an diesem Node als Splitter. Hier ist der Pseudocode dazu:

```
PLANE FindBestSplitter(polygonlist) {
    PLANE BestSplitter, CurrentSplitter;
    int    nClass=0,
           nFront=0,
           nBack=0,
           nSplits=0
           nBestScore=99999,
           nCurrentScore=0;

    for ( i < num polygons at node )
    {
        CurrentSplitter = Polygon[i].GetPlane();
        for ( j < num polygons in node )
        {
            if ( i == j ) continue;

            nClass = CurrentSplitter.Classify(Polygon[i]);
            if (nClass == BACKSIDE)    nBack++;
            else if (nClass == CLIPPED) nSplits++;
            else if (nClass == FRONTSIDE) nFront++;
            else {
                if (NormalsSameDirection(Plane,Polygon[i]))
                { nFront++; }
                else

```

```

        { nBack++; }
    }
    nCurrentScore = abs(nFront - nBack)
                  + (nSPlits*3);

    if (nCurrentScore < nBestScore) {
        BestSplitter = CurrentSplitter;
        nBestScore   = CurrentScore;
    }
} // for
} // for
return BestSplitter;
} // function

```

Nun sollte uns langsam ein Licht aufgehen, wozu wir in unserer Mathe-LIB die ganzen Klassifizierungsmethoden und die Methoden zum Zerschneiden eines Polygons implementiert haben. Mit diesen Funktionen ist das Programmieren eines BSP-Algorithmus zur Erstellung eines BSP-Baums ein Klacks.

Klassifizierungen

Node-Based-, Leavy- und solide BSP-Trees

Wenn man sich ein wenig eingehender mit der Materie der BSP-Bäume befasst, wird man auf drei verschiedene Varianten von BSP-Bäumen stoßen. Diese nennen sich *Node-Based BSP-Tree*, *Leavy BSP-Tree* und *Solid Leaf BSP-Tree*. Dabei handelt es sich nur um minimale Variationen des Algorithmus, die sich hauptsächlich auf die Art und Weise beziehen, wie und wo man die Polygone in dem Baum letzten Endes speichert.

*Varianten des
BSP-Baums*

Die einfachste Variante des BSP-Baums ist die Node-Based-Variante. Sie heißt so, weil man einfach alle Polygone an den einzelnen Nodes des Baums abspeichert. Wenn man die Geometrie an einem Node des Baums teilt, speichert man das Polygon, dessen Ebene man als Splitter verwendet hat, direkt in dem Node und entfernt es aus der Liste, die an die Children weitergereicht wird. Das macht man so lange, bis man in einem Ast des Baums nur noch einen konvexen Teil Geometrie übrig hat. Nun wählt man jeweils ein beliebiges Polygon aus dieser Menge aus und verwendet es als Splitter. Logischerweise bleiben die Back-Listen ab diesem Node leer, denn alle Polygone des konvexen Stücks Geometrie liegen ja jeweils im Front-Bereich von jeweils allen anderen Polygonen desselben Stücks. Die Backlist ist also leer oder NULL, wie wir Programmierer sagen würden. Die Frontlist hingegen unterteilt man immer weiter, bis man die Bedingung erfüllt hat, dass noch genau ein Polygon in der Frontlist am Ende des Astes übrig bleibt und alle anderen vorher als Splitter verwendet wurden.

*Node-Based BSP-
Trees*

Das hat den Sinn, dass man so genannten »leeren Raum« (engl. empty space) und »soliden Raum« (engl. solid space) unterscheiden kann. Der Clou des BSP-Baums: Der leere Raum ist freier Raum innerhalb des Levels,

in dem man sich bewegen kann. Abgegrenzt durch die Polygone der Level-Geometrie, ist der solide Raum hingegen ungültiger Raum, also Raum, in dem sich nichts befinden darf, weil es sonst durch eine Wand des Levels gegangen wäre und außerhalb der Level-Geometrie stehen würde. Natürlich ist uns sofort klar, dass der Raum innerhalb der konvexen Geometrie-Stücke leerer, gültiger Raum ist. Wenn man nun einen Punkt in den BSP-Baum schickt, dann prüft man an jedem Node, auf welcher Seite des Splitters er sich befindet, und schickt ihn in den entsprechenden Front- oder Backlist-Ast des Nodes weiter. Gelangt der Punkt schließlich bis in ein Front-Leaf mit einem Polygon, dann befindet er sich innerhalb des Levels an einer gültigen Stelle. Ist der Punkt aber irgendwann gezwungen, in einen soliden Node abzubiegen, also in eine `NULL`-Backlist, dann befindet er sich eindeutig nicht an einer gültigen Position innerhalb des Levels, sondern hinter einer Wand, hinter der auch kein anderer Raum des Levels mehr liegt.

Wie durch Zauberhand erhält man so die Möglichkeit, Kollisionsabfragen für Objekte in einem Level ohne Berechnung auszuführen. Man muss nur den BSP-Baum durchlaufen.

Leavy BSP-Trees

Die Leaf-Based BSP-Bäume speichern an den inneren Nodes des Baums nun nicht mehr das als Splitter verwendete Polygon, sondern einfach nur dessen Ebene. Das Polygon wird dann durch ein Flag markiert, das angibt, dass das Polygon bereits als Splitter verwendet wurde, weil das nur einmal passieren darf. Es bleibt jedoch in der Liste der Polygone bzw. wird in die Frontlist einsortiert. Wenn man nun keinen Splitter mehr findet, dann erstellt man ein Leaf, und dieses enthält dann eine Liste mit allen Polygonen, die das konvexe Stück Geometrie in diesem Leaf ausmachen. Der Nachteil des Node-Based BSP-Baums ist nämlich, dass man die Objekte dort nur sortiert hat, aber man hat nicht automatisch eine Liste mit Polygonen für konvexe Stücke der Geometrie. Der Nachteil des Leaf-Based BSP-Baums ist jedoch, dass man dort keine `NULL`-Leaves hat, die soliden Raum repräsentieren.

Solid Leaf BSP-Trees

Wie so oft im Leben kann man auch hier die besten Elemente verschiedener Optionen kombinieren. Ein Solid Leaf BSP-Baum ist die Erweiterung des Leavy BSP-Baumes um die Informationen über soliden Raum. Dazu beendet man die Erstellung des Baums nicht in einem Leaf mit der konvexen Geometrie, sondern verwendet dort aus der Liste der Polygone des Leafs alle Polygone als Splitter, die diese Aufgabe bisher noch nicht erfüllt haben. Auch hier werden alle Polygone der konvexen Geometrie wiederum in die jeweilige Frontlist einsortiert, und die Backlist ist jeweils `NULL`, also solider Raum.

Portal-Engines

Portale kontra BSP-Baum

Während John Carmack den BSP-Baum im Bereich der 3D-Grafik salonfähig machte, erblickte noch ein anderes Verfahren das Licht der dreidimensionalen Grafikwelt. Dabei handelte es sich um die Portal-Engine, die

beispielsweise in dem Megahit *Duke Nukem 3D* von 3D Realms und Apogee Anwendung fand. Auch wenn *Duke Nukem 3D* noch nicht wirklich vollkommen dreidimensional war, so war es doch schon um einiges dreidimensionaler als *Doom 1* und *Doom 2*. Zwar wurden auch hier noch zweidimensionale Sprites für die Darstellung der Gegner verwendet, aber man konnte mit ein bisschen Erfahrung bereits übereinander liegende Räume mit dem Level-Editor *build!* erzeugen, der dem Spiel beilag.⁴ Auch wenn dies dem Spieler nicht unbedingt auffiel, so war das doch eine kleine Revolution, denn mit den rein zweidimensionalen BSP-Bäumen von Carmack war dies nicht möglich. Das Geheimnis hinter der Fassade war die Verwendung einer Portal-Engine.

Seinerzeit habe ich mit build! einen Level gebaut, der etwas ermöglichte, was sicherlich nicht einmal die Entwickler des Tools bedacht hatten, denn die Original-Level des Spiels boten nichts Vergleichbares. Man konnte in einem großen Außenareal herumtoben und durch die Fenster in ein zweistöckiges Hauptgebäude schauen und dieses durch Fenster und Türen betreten. Dort konnte man dann in die zweite Etage gelangen, die ebenfalls die gesamte Grundfläche des Gebäudes mit begehbaren Räumen bedeckte. Auch aus den Fenstern der zweiten Etage konnte man nach draußen in das Außenareal schauen und sogar durch die Fenster wieder in das Außenareal springen.

Heutzutage ist das absolut nichts Besonderes mehr. Man beachte jedoch die Restriktion der build!-Engine, dass man von einem Raum nicht in einen über oder unter diesem liegenden Raum blicken konnte, weil das Grafikfehler erzeugte. Mein kleiner Level ermöglichte eine selbst mit dem Original-Spiel nicht mögliche 3D-Erfahrung für den Spieler. Wer sich mit dem build!-Editor ein wenig auskennt und auch weiß, wie man das realisieren kann, der schreibe mir eine E-Mail. Er erhält ein Lob von mir für besondere Verdienste in der Ausreizung beschränkter Tools to the max ... :-)

Und die Moral von der Geschicht': Ausgereizt sind auch die kommerziellen Tools in den meisten Spielen nicht.

Die Idee der Portal-Engine

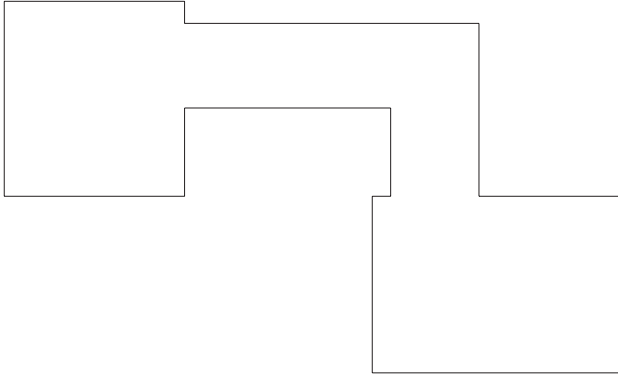
Doch wie konnte eine Portal-Engine nun etwas realisieren, was vorher nicht so einfach möglich schien? Wir werfen dazu einen Blick auf Abbildung 13.17, die einen kleinen Level in der Draufsicht zeigt.

Portal-Engine

⁴ Zu den weiteren Neuerungen zählte hier u.a., dass der Spieler auch seinen Blickwinkel heben und senken konnte, was bei *Doom* nicht möglich war, dass Böden und Decken nicht mehr horizontal sein mussten, dass es in Echtzeit reflektierende Spiegel gab, dass man in Fahrzeugen fahren und Wände einstürzen lassen konnte und noch etliches mehr.

Abbildung 13.17:

Ein einfacher dreidimensionaler Level mit zwei Räumen und einem Verbindungsgang aus der Vogel-Perspektive

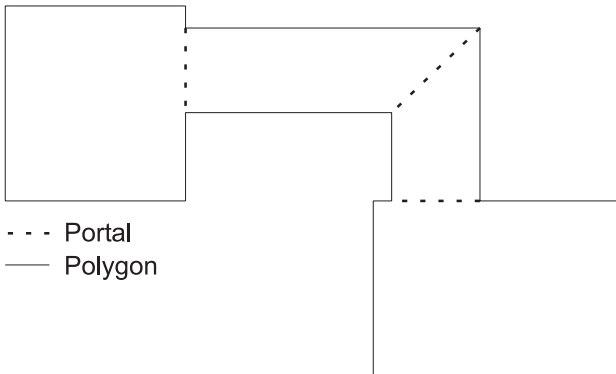


Wo war noch mal das Problem?

Die große Frage (wie auch bei den BSP-Bäumen) lautet immer noch: Wie kann man so schnell wie möglich die Untermenge an Polygonen herausfinden, die wirklich im Frustum eines Spielers liegen? Der BSP-Baum dient in seiner ursprünglichen Form dazu, dass man ihn durchlaufen kann und dabei eine Liste an Polygonen aus den Leaves zusammensucht, die sichtbar sind. Mit der Portal-Engine ist das ein wenig anders, denn dabei handelt es sich nicht um eine Baumstruktur. Abbildung 13.18 zeigt den Level, in den bereits die Portale eingefügt wurden. Man erkennt, dass der Level auch hier in konvexe Stückchen Geometrie aufgeteilt wird. An den Schnittkanten der konvexen Stückchen sitzen die Portale, die dieser Technik ihren Namen geben. Die konvexen Stückchen Geometrie nennt man im Jargon der Portal-Engine entweder »Sektor«, »Zelle« oder neuerdings auch »Area«.

Abbildung 13.18:

Derselbe Level, allerdings nun in konvexe Stücke unterteilt. Die konvexen Stücke (auch »Sektoren«, »Zellen« oder »Areas« genannt) werden an ihren Schnittstellen durch die Portale miteinander verbunden.



Sinn der Portale

Der Witz an der ganzen Geschichte ist nun, dass man den Level gar nicht als zusammenhängende Geometrie betrachten muss. Man muss eigentlich nur die einzelnen Sektoren speichern. In den Sektoren sind jeweils die Portale gespeichert, die zu ihnen gehören. Dabei hat ein Portal immer zwei Seiten, wobei jede Seite nur zu dem Sektor gehört, in den die Normale dieser Seite zeigt. Jedes Polygon hat also je eine Seite in zwei verschiedenen Sektoren. Und genau das ist die Information, über die man die Sektoren miteinander

in Beziehung setzt. Nun muss man nur wissen, in welchem Sektor sich ein Spieler am Anfang befindet; dieser muss auf alle Fälle gerendert werden. Dann durchläuft man die Liste mit Portalen dieses Sektors. Liegt ein Portal dieses Sektors im Frustum des Spielers, so kann der Spieler auch den Sektor sehen, zu dem die Rückseite dieses Portals gehört. Auch dieser Sektor muss gerendert werden. Und nun kommt ein kleiner Trick der Portal-Engine: Man verkleinert jetzt das Frustum des Spielers so, dass es genau durch das betrachtete Portal passt. Für den dahinter liegenden Sektor hat man also ein verengtes Frustum, mit dem man dann in diesem Sektor wiederum die dortigen Portale auf Sichtbarkeit prüft.

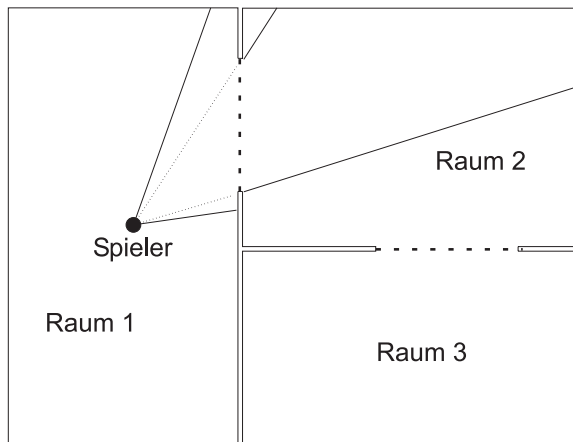


Abbildung 13.19: Frustum und durch ein Portal verkleinertes Frustum des Spielers in einem kleinen Level mit drei konvexen Räumen und zwei Portalen, die diese Räume miteinander verbinden

Abbildung 13.19 zeigt einen kleinen Level mit drei Räumen aus der Vogel-Perspektive. Der Spieler befindet sich in Raum 1, der folglich schon mal gerendert werden muss, und schaut nach oben rechts. Dabei liegt das Portal, das die Räume 1 und 2 verbindet, in seinem View-Frustum. Damit muss also auch der Raum 2 gerendert und weiter betrachtet werden. Dazu wird das Frustum des Spielers so verkleinert, dass es genau auf das Portal zugeschnitten ist. Das andere Portal, das von Raum 2 zu Raum 3 führt, liegt nicht im neuen Frustum des Spielers und muss daher nicht gerendert und auch nicht weiter betrachtet werden. Befände sich auch dieses Portal im verkleinerten Frustum, so hätte man auch den Raum 3 rendern müssen. Dann hätte man das verkleinerte View-Frustum wiederum verkleinert, so dass es genau durch das Portal zwischen Raum 2 und Raum 3 passen würde.

*Beschnittene
Frustums*

Wenn ihr meine bisherigen Ausführungen über die Historie der 3D-Grafik verfolgt habt, solltet ihr jetzt einen kleinen Haken erkannt haben: Die BSP-Bäume dienten ja dazu, dass man konvexe Stückchen Geometrie Back-to-Front rendern konnte, ohne dass es dabei zu fehlerhaften Überlagerungen von Polygonen kam, weil weiter hinten gelegene Polygone über weiter vorne gelegene Polygone gezeichnet wurden. Ein Depth-Buffer, wie man ihn heute verwendet, war damals nicht realisierbar, u.a. wegen der großen Menge an

Zero Overdraw

benötigtem Speicher. In der Portal-Engine rendern wir aber auf alle Fälle mehr oder weniger Front-to-Back, müssten also ohne einen Depth-Buffer ein total fehlerhaftes Bild erhalten, weil die entfernten Polygone durch die vorderen scheinbar durchscheinen. Sie werden ja auch erst später über diese gerendert, weil entferntere Sektoren später gerendert werden als die näher gelegenen. Doch hier greift der zweite Trick der Portal-Engine: Während der BSP-Baum eine perfekte Back-to-Front-Sortierung ermöglichte, lieferte die Portal-Engine die Möglichkeit, mit **Zero Overdraw** zu rendern. Damit ist gemeint, dass man jeden Pixel auf dem Bildschirm nur ein einziges Mal rendert und es überhaupt nicht zu Übermalungen kommt.

Weiteres Clipping

Das rührt daher, dass man zuerst den Sektor mit dem Spieler rendert. Damit ist der gesamte Bildschirm vollkommen belegt, mit Ausnahme des Bereichs, in dem das Portal sitzt. Nur hier dürfen wir jetzt noch rendern, wenn wir Zero Overdraw haben wollen. Wir nehmen also den Sektor, zu dem das Portal führt, greifen uns seine Polygonliste und – **tataaaa** – zerschneiden die Polygone dieses Sektors so, dass sie genau in das verkleinerte Frustum passen. Wenn wir diese geclippten Polygone nun rendern, belegen wir auf dem Bildschirm alle Pixel, die bisher noch nicht bemalt worden sind, natürlich wiederum mit Ausnahme des Bereichs, in dem ein Portal dieses Sektors liegt, das sichtbar ist. Hier machen wir dasselbe Spiel wieder.

Portale kontra BSP-Baum, die Zweite

Das klingt doch perfekt – oder? Damit sind wir wieder bei dem erbitterten Kampf von Portal-Engine gegen BSP-Baum, den die Anhänger des einen Ansatzes gegen die Anhänger des anderen Ansatzes geführt haben. Kritiker warfen dem BSP-Baum vor, dass man dort unter Umständen eine ganze Menge an Overdraw hat, was auf den damaligen Grafikkarten alles andere als optimal war. Im Gegenzug warfen Kritiker der Portal-Engine vor, dass man dort extrem viel Polygon-Clipping an den Portalen zur Laufzeit hat, was für die damaligen CPUs alles andere als optimal war. Natürlich haben beide Seiten recht, aber mit beiden Ansätzen wurden auch erstklassige Spiele realisiert. Einen absolut besseren Ansatz gibt es also nicht, und heutzutage ist man dank der guten Hardware sowieso i.d.R. nicht mehr gezwungen, perfekt Back-to-Front oder mit Zero Overdraw zu rendern. Von daher greifen wir die kritische Diskussion der Ansätze hier auch gar nicht mehr auf.

Sieh's locker

Heutzutage wird man den Ansatz der BSP-Bäume in einer lockereren Form sehen. So gibt es beispielsweise Varianten, die gar keine Splits mehr in den Polygonen auslösen, weil sie den BSP-Baum eben nicht dazu brauchen, konvexe Geometrie-Happen zu erzeugen. Und bei einer Portal-Engine kann man auf den Zero Overdraw sowieso verzichten. Letzten Endes dürfte aber eine Portal-Engine zum Rendern auf heute moderner Hardware die Nase vorn haben, denn das Front-to-Back-Rendern ist heutzutage durch das Early-Z-Culling der Grafikkarten effizienter. Aber das Clipping der Polygone sichtbarer Sektoren kann man sich getrost schenken und brute-force rendern. Die Grafikkarte ist dabei schneller, als die CPU es beim Clipping der Polygone wäre.

Auf die Portal-Engine gehen wir hier nicht weiter ein. Wir werden aber garantiert auf sie zurückkommen, denn in unserem Spiel am Ende dieses Buches verwenden wir auch eine moderne Variation einer Portal-Engine – natürlich ohne Polygon-Clipping und ohne konvexe Sektoren. Schließlich soll sie ja performant laufen.

Special-Effects mit Portalen

Im Gegensatz zu einem BSP-Baum haben Portal-Engines aber einen ganz entscheidenden Vorteil: Mit ihnen kann man nämlich eine ganze Reihe von Special-Effects leicht implementieren. Während ein normaler BSP-Baum beispielsweise die Daten für den Level als zusammenhängende Menge von Polygonen erwartet, ist es der Portal-Engine vollkommen egal, ob die Daten zusammenhängen oder nicht. Entscheidend ist das allenfalls, wenn man die Portale automatisch generieren möchte. Wenn man aber die Geometrie bereits in (gern auch konkaven) Sektoren vorliegen hat und diese Sektoren Portale haben, die zu anderen Sektoren führen, dann ist es herzlich egal, ob ein Sektor auch physisch an denjenigen Sektor angrenzt, zu dem man durch eines seiner Portale gelangt. Es handelt sich hierbei um das alte *Star Gate*-Prinzip. Wenn man in einem Raum in Berlin ist, dann kann man einfach durch eine Tür gehen und gelangt in den Sektor, in den das Portal, das im Berliner Türrahmen sitzt, einen führt. Ob das der physikalisch in Berlin gelegene Nachbarraum ist, ein Office in New York oder der Transporterraum der *Voyager* spielt für die Portal-Engine keine Rolle. So kann man auch ein *Star Gate* mitten in einem Raum platzieren, durch das man auf einen fremden Planeten blicken und diesen auch betreten kann. Auf der Rückseite des Portals ist weiterhin nichts bzw. nur der *Star Gate*-Raum zu sehen.

*Special-Effects
mit Portalen*

Ebenso kann man ein Portal erzeugen, das nicht durchschritten werden kann, sondern einfach denselben Sektor aus einem invertierten Sichtwinkel heraus auf eine Textur rendert und sich dann selbst mit dieser Textur rendert. Eine ziemlich komplizierte Umschreibung für einen ordinären Spiegel, nicht wahr?

Spiegel

Oder wie wäre es mit *Alice im Wunderland*? Ein nettes Labyrinth-Spiel wäre eine Portal-Engine mit Zufallsauswahl der Sektoren beim Durchschreiten der Portale. Mal kommt man durch die Tür von Raum 1 in Raum 37, ein anderes Mal gelangt man durch dieselbe Tür per Zufallsauswahl in den Raum 92. Zugegebenermaßen ist es schwierig, so ein sinnvolles Spiel zu erstellen. Aber man beachte, dass man normalerweise durch ein Portal den angrenzenden Sektor sehen kann. Wenn man nun durch den Rahmen eines *Star Gates* alle fünf Sekunden einen anderen Planeten sieht, so ist das schon nicht mehr so abwegig. Man muss halt so lange warten, bis der richtige Sektor wieder drankommt. Wenn man also nicht ganz per Zufall den Sektor auswählt, sondern einen Zyklus von vier oder fünf möglichen Zielorten

Zufalls-Portale

kontinuierlich durchläuft, dann kann man damit schon etwas anfangen. Schon mal etwas von fluktuierenden Wurmlöchern gehört?⁵ :-)

Catch 21 Eine Sache müssen wir allerdings bei der ganzen Portal-Bastelei beachten: Wenn wir einen Sektor A gerade gerendert haben und durch eines seiner Portale in einen Nachbarsektor B blicken können und diesen auch rendern, dann müssen wir darauf achten, dass wir nicht durch ein Portal des Sektors B wieder im Sektor A landen. Dann wären wir in einer Endlosschleife gefangen und würden beim Rendern immer zwischen den beiden Sektoren hin- und herwandern. Also muss das Portal ein Flag erhalten, dass es bereits durchwandert wurde und nicht wieder rückwärts durchwandert werden darf, richtig? Falsch, das hilft leider nur in der Mehrzahl der Fälle. Aber wenn wir einen Special-Effect haben, dass wir durch ein anderes Portal in der gegenüberliegenden Wand vom Sektor B wieder in Sektor A kommen, dann durchschreiten wir ein neues Portal und gelangen zu einem schon besuchten Sektor. Man muss also dem Sektor ein Flag geben, das aussagt, ob er in einem Frame schon besucht wurde. Ähnliches gilt für Spiegel. Denn wenn zwei Spiegel sich sehen können, dann bricht die Hölle los so wie bei *Duke Nukem 3D* und die Engine hängt sich in einer Endlosschleife auf. Es wäre aber denkbar, dass man eine gewisse Anzahl an Iterationen für einen Spiegel zulässt. Wir könnten erlauben, dass sich zwei Spiegel ein Bild beispielsweise zehnmal gegenseitig zuwerfen, und dann würde der Algorithmus die Schleife brechen. So hätte man auch einen netten Spiegel im Spiegel-Effekt in seiner Engine.

*Die virtuelle Welt
hat keine
Grenzen!*

Denkt immer daran, dass die Grenzen in der Virtualität dort sind, wo man sie selbst hinlegt. Und in diesem Fall simuliert man ja auch nur die abnehmende Energie der Photonen bei der Reflexion in einem Spiegel. Würden die Photonen in der Realität auch in einer Endlosschleife laufen, dann könnte man so ein Perpetuum mobile konstruieren und die Stoßenergie der Photonen zur Energie-Transformation nutzen. Aber ich glaube, wir schweifen vom Thema ab.

Automatische Portal-Generierung

*Autopilot oder per
Hand?*

Nachdem wir uns nun selbst davon überzeugt haben, dass eine Portal-Engine eine nette Sache ist, stellt sich die Frage, wie man eine Menge von Polygonen eines Levels in Datenstrukturen bringen kann, die den Sektoren und Portalen einer Portal-Engine entsprechen. Dazu gibt es zwei Verfahren. Das erste besteht darin, die Level-Daten automatisch in Sektoren aufzuteilen und automatisch Portale für diesen Datensatz zu generieren. Die andere Alternative ist es, den Level-Designer mit der Aufgabe zu betrauen, den Level per Hand im Level-Editor in einzelne Sektoren aufzuteilen und Portale zwischen diesen anzulegen. Das zweite Verfahren ist ein sehr sinnvoller

5 Für die wissenschaftliche Abhandlung der Thematik »Wurmlöcher« vgl. z.B. Thorne, K.S.: »Gekrümmter Raum und verbogene Zeit.«, München, 1994, oder Halpern, P.: »Wurmlöcher im Kosmos«, München u.a., 1994

Weg, den wir hier auch gehen werden, und zwar wiederum aus genau zwei Gründen: Zunächst einmal ist die automatisierte Generierung der Portale eine knifflige Angelegenheit, und ein entsprechender Compiler für einen Level würde den Umfang dieses Buches bei weitem sprengen (wenn er bugfrei und auch allgemein nutzbar sein soll). Der zweite Grund ist vielleicht viel einsichtiger: Ein solcher Automatismus, den ich gleich skizzieren werde, erzeugt viel zu viele Sektoren und Portale in einem Level. Das wiederum führt auf heute aktueller Grafik-Hardware zu einem Overhead an Processing Time auf der CPU und zur Stückelung der Render-Aufträge an die Grafikkarte, so dass man dadurch einfach viel zu viel an Scene-Management hat und sich damit wieder selbst ausbremst und keinen Performance-Gewinn erzielt.

Aber wer sein (Un-)Glück dennoch versuchen möchte, der frage sich nun, woran uns der Begriff konvexe Sektoren sofort erinnert? Na klar, an den Urvater der Konvexität namens BSP-Baum. Man zerlegt die ursprüngliche Level-Geometrie also zunächst in einen Leavy BSP-Baum. Nun hat man bereits die konvexen Sektoren, nämlich die konvexen Leaves. Jetzt bleibt nur noch die Kleinigkeit, die Portale zwischen diesen Leaves zu erzeugen. Potenziell kann auf jeder Ebene, die als Splitter verwendet wurde, ein Portal liegen. Denn allein die Splitter-Ebenen bilden ja die Schnittstellen zwischen den konvexen Leaves. Wir müssen also für **jeden** Node des BSP-Baums, der kein Leaf ist, ein rechteckiges *Initial-Portal* erstellen und dieses weiter bearbeiten. Die weitere Bearbeitung besteht darin, dass wir dieses Initial-Portal von seinem Node an abwärts durch den Baum schicken. Und was bei dieser Reise durch die Tiefen des Baums mit dem Initial-Portal geschieht, sehen wir uns gleich an.

*Woran erinnern
uns konvexe
Sektoren?*

Das Initial-Portal an einem Node muss so groß sein, dass es einmal durch die gesamte Geometrie reicht, die unterhalb dieses Nodes im BSP-Baum gespeichert ist. Als einzige Information für diesen Zweck haben wir die Bounding-Box des Nodes und die dort gespeicherte Splitter-Ebene. Dann berechnet man die Entfernung des Mittelpunktes der Box zu der Ebene und läuft anschließend über diese Distanz von dem Mittelpunkt aus in Richtung der Normalen der Ebene bzw. ihrer invertierter Normalen, wenn der Mittelpunkt auf der Rückseite der Ebene liegt. Nun hat man den Mittelpunkt des Initial-Portals berechnet, und der Normalenvektor der Ebene ist auch gleichzeitig der Normalenvektor des Portals. Um die Vertices des Portals berechnen zu können, brauchen wir aber noch den Rechtsvektor und den Hochvektor des Portals. Diese können wir über das Kreuzprodukt bestimmen. Mit dem Normalenvektor bilden wir das normalisierte Kreuzprodukt mit dem Welt-Hochvektor $(0,1,0)$, dem Welt-Richtungsvektor $(0,0,1)$ oder aber mit dem Welt-Rechtsvektor $(1,0,0)$. Wir wählen denjenigen der drei Vektoren, der bei einem Punktprodukt mit dem Normalenvektor den kleinsten absoluten Wert erzeugt und folglich am wenigsten an der Normalen ausgerichtet ist. Idealerweise steht er bereits in einem 90-Grad-Winkel

*Das Initial-Portal
erstellen*

zu ihr. Damit haben wir den zweiten Vektor U der drei Achsen des Portals, den wir mit dem Normalenvektor kreuzen. Das Ergebnis normalisieren wir ebenfalls, um den dritten und letzten Vektor V des Systems zu erhalten.

Nun haben wir unsere Vertices schon so gut wie berechnet. Wir kalkulieren die Länge fL der Strecke vom Mittelpunkt der Box zu ihrem Maximalpunkt. Das ist quasi die halbe Diagonale der Box und beschreibt ihre maximale Ausdehnung in eine Richtung vom Mittelpunkt aus. Mit dieser Länge fL skalieren wir die Vektoren U und V und erhalten die Positionen der vier Vertices wie folgt:

```
vc[0] = Box.vcCenter + U + V;
vc[1] = Box.vcCenter + U - V;
vc[2] = Box.vcCenter - U + V;
vc[3] = Box.vcCenter - U - V;
```

Clipping des Initial-Portals

Nun haben wir unser Initial-Portal und senden es von seinem Node an abwärts durch den Baum. An jedem weiteren Node klassifizieren wir das potenzielle Portal genau so, wie wir es mit den Polygonen gemacht haben. Im Folgenden ist aufgelistet, wie wir das Ergebnis der Klassifizierung jeweils handhaben müssen:

- ➔ FRONT
Sende das Portal in den Ast mit der Frontlist dieses Nodes.
- ➔ BACK
Sende das Portal in den Ast mit der Backlist dieses Nodes.
- ➔ PLANE
Sende das Portal zuerst in den Ast mit der Frontlist. So erhält man eine Liste von geclippten Portalen als Rückgabe. Diese sendet man dann nach und nach in den Ast mit der Backlist des Nodes.
- ➔ CLIP
Zerschneide das Portal und zwei Portal-Fragmente, und sende sie in den zugehörigen Ast mit der Front- bzw. Backlist.

Ein Initial-Portal wird also bei der Jagd durch den Baum mit den Splitter-Ebenen der Nodes in viele Fragmente zerteilt. Irgendwann kommt ein solches Fragment dann in einem Leaf des BSP-Baums an. Hier entscheidet sich, ob es zunächst weiterleben darf oder nicht. Ein Fragment eines Initial-Portal hat nur dann das Potenzial, ein valides Portal zu sein, wenn es sich innerhalb des konvexen Bereichs eines Leafs befindet. Wenn ein Fragment also ein Leaf erreicht, dann prüfen wir es dort gegen die Liste von Polygonen. Das Fragment muss innerhalb der Box des Leafs liegen, es darf kein Polygon der Liste schneiden, und es darf sich nicht auf der Rückseite eines der Polygone befinden. Treffen alle drei Bedingungen zu, dann speichern wir das

Fragment als potenzielles Portal. Anderenfalls verwerfen wir es, weil es sich außerhalb des Levels befindet.

Nachdem wir die Initial-Portale aller Nodes des BSP-Baums betrachtet haben, befinden sich in unserer Liste potenzieller Portale eine ganze Menge Einträge. Doch nicht alle sind auch wirklich valide Portale. Wir haben zwei Typen schwarzer Schafe dabei. Ein gültiges Portal muss immer zu zwei Leaves gehören, die es miteinander verbinden soll. Wir durchlaufen also die Liste potenzieller Portale und schmeißen alle Portale heraus, die nicht exakt in zwei verschiedenen Leaves vorkommen. Ein gültiges Portal muss zweimal mit denselben Vertices und einer gegenläufigen Normalen in zwei verschiedenen Leaves vorkommen. Alle anderen Einträge in der Liste werden gelöscht.

*Entfernung
invalider Portale*

Der zweite Typ schwarzer Schafe bezieht sich auf redundante Portale. Der oben beschriebene Mechanismus generiert unter Umständen auch gültige Portale, die in unterschiedlicher Größe an derselben Stelle im Raum liegen. Man muss nun also aus der bisherigen Liste eine neue Liste erstellen, der man nach und nach die Einträge aus der bisherigen Liste hinzufügt. Zwei Sektoren können immer nur durch ein gültiges Portal (mit Vorderseite und Rückseite) verbunden sein, nicht durch mehr. Wenn wir in unserer neuen Liste beim Hinzufügen eines Portals bereits eines in der Liste finden, das dieselben Leaves miteinander verbindet, dann ist eines dieser beiden Portale überflüssig und wird aus der Liste entfernt. Dabei löscht man immer das Portal, das das kleinere ist.

Nun haben wir eine Liste von Portalen, bei der jedes Portal aus zwei Seiten besteht. In unserer Liste werden sich aber extrem viele Portale befinden. Wie bereits weiter oben erwähnt wurde, führt die Erstellung eines BSP-Baums zu einer großen Anzahl an Splits in der ursprünglichen Geometrie, und die Leaves mit den konvexen Stücken Geometrie sind sehr klein. Und jedes Leaf hat mindestens ein Portal. Ich überlasse es an dieser Stelle euch zu entscheiden, ob Ihr diesen Aufwand für die automatische Generierung der Portale investieren wollt. Ich jedoch plädiere für weitaus weniger Portale, die man manuell setzt.

Fertig!

Potential Visibility Set

Eine triviale, aber eiserne Regel bei der Computerspiele-Entwicklung ist die folgende: Alles, was man vorberechnen kann, braucht man nicht zur Laufzeit zu berechnen! So trivial das auch erscheinen mag, so sensationell muten doch die ersten Schritte an, diese Regel auch auf die Sichtbarkeit eines Indoor-Levels zu übertragen. Dazu machen wir wieder eine kleine Exkursion in die Historie und landen erneut bei einem guten alten Bekannten – John Carmack. In der Artikelserie *Inside Quake* von Michael Abrash, einem der Programmierer von *Quake*, findet sich auch ein Abschnitt über die *Visible-Surface Determination*. Hier beschreibt Michael Abrash, auf welche

*Und wieder dieser
Carmack!*

Art und Weise in den frühen Stadien der Entwicklung des Spiels versucht wurde, die Performance von Leveln mit einer vergleichsweise hohen Anzahl an Polygonen im spielbaren Bereich zu halten. John Carmack, der hauptsächlich für die Entwicklung der Engine verantwortlich war, probierte viele verschiedene Ansätze aus. Insbesondere konzentrierte er sich darauf, die Sichtbarkeitsinformationen einzelner Leaves direkt aus dem BSP-Baum zu extrahieren. Nach einem langen Arbeitswochenende von John Carmack kam Michael Abrash am Montag in die Firma. Aber lassen wir ihn selbst sprechen:

»The first time John showed me his working prototype, I went to the most complex scene I knew of, a place where the frame rate used to grind down into the single digits, and spun around smoothly, with no perceptible slowdown.«⁶

*PVS does the
Trick – BSP-Baum
plus Portale*

Nachdem er das gesamte Wochenende am dem direkten BSP-Ansatz gearbeitet hatte, kam John Carmack am Montagmorgen im Bett die Idee. Anstatt umständlich zu versuchen, die Information über Sichtbarkeiten zur Echtzeit aus dem BSP-Baum zu extrahieren, wollte er nun die Sichtbarkeit als Vorberechnung realisieren, die mit dem Level abgespeichert werden konnte. Wenn man die einzelnen Leaves des BSP-Baums über Portale miteinander verband, dann konnte man für jedes Leaf exakt berechnen, welche anderen Leaves von diesem aus überhaupt potenziell sichtbar waren. Potenziell deshalb, weil man bei einem Raum mit zwei offenen Türen, die sich gegenüberliegen, nun mal nicht gleichzeitig aus beiden Türen heraus schauen kann. Potenziell sind von diesem Leaf aus aber die beiden jeweils an den Türen angrenzenden anderen Leaves sichtbar. Zur Laufzeit kann man erst entscheiden, in welche Richtung der Betrachter schaut und durch welche Tür er folglich blickt. Ein Durchlaufen des BSP-Baums wurde damit zur Laufzeit überflüssig. Jedes Leaf speichert in einer Liste Pointer auf diejenigen anderen Leaves, die man durch seine Portale potenziell sehen kann. Ausgehend von dem Leaf, in dem sich die Kamera befindet, muss man dann nur die entsprechende PVS-Liste des Leafs durchlaufen, die dort enthaltenen Leaves einem Frustum-Culling unterziehen und diese dann einfach rendern.

*Der Trick ist aber
tricky*

So schön dieser Ansatz auch klingt, so viele Probleme bringt er auch mit sich. Der BSP-Baum muss ganz bestimmte Bedingungen erfüllen, damit die Berechnung des PVS erfolgreich ist. Dazu muss die Level-Geometrie wiederum bestimmte Restriktionen beachten, die vergleichsweise schwer umzusetzen sind. Auch mit den offiziellen Tools, mit denen man für solche Spiele Level erstellen kann, ist es möglich, Level zu erzeugen, die den BSP- und PVS-Compiler zum Absturz bringen. Wir werden auf die Berechnung eines PVS hier nicht weiter eingehen, da dies ein sehr aufwändiges Verfahren ist, wenn man es wasserdicht haben will. Zum anderen ist ein PVS in der heuti-

⁶ Abrash, Michael: »Ramblings in Realtime, Chapter 1: Inside Quake – Visible Surface Determination«, u.a. <http://www.bluesnews.com/abrash/>

gen Hardware-Welt mit schnellen Grafikkarten und Prozessoren nicht mehr von so entscheidender Bedeutung wie noch vor wenigen Jahren. Abschließend sei auch noch erwähnt, dass man ein PVS natürlich auch für Outdoor-Level vorberechnen kann und man sich nicht selbst auf den Portal-Ansatz von Quake beschränken sollte, wenn man Nachforschungen in diese Richtung betreibt.

Einen kompletten Kurs zum Thema BSP-Bäume mit PVS kann man bei der Online-Akademie *Game Institute* für 95 US-\$ belegen.⁷ Dort erhält man entsprechendes Material und Source-Code für einen stabilen Compiler.

*Weitere
Informationen*

13.3 Implementierung eines BSP-Baums

Wir haben inzwischen viel über Scene-Management-Techniken gehört, doch noch nicht gesehen, wie man so etwas wirklich im Code umsetzen kann. In diesem Abschnitt werden wir eine Klasse für BSP-Bäume implementieren und zu unserer Mathe-Lib hinzufügen. Wir werden dazu auch ein kleines Demo-Programm entwickeln, das die Funktion unseres BSP-Baums demonstriert. Wir werden hier übrigens die Variante des Leavy BSP-Trees implementieren.

*Erweiterung der
Mathe-Lib um
Leavy BSP-Trees*

Die Performance dieser Demo wird sehr bescheiden sein. Wir konstruieren unseren BSP-Baum ganz ausdrücklich **nicht** zum Rendern, sondern für Kollisionsabfragen. In einer echten Life-Fire-Anwendung würden wir die Daten der Geometrie für einen Level in statische Buffer auf die Grafikkarte packen und von dort aus einfach rendern. Im System-RAM haben wir dann einen BSP-Baum für die Geometrie, also quasi eine Kopie des Levels ohne Texturkoordinaten und den ganzen Klimbim. Mit dieser Kopie, die beispielsweise in Form eines BSP-Baums vorliegt, führen wir dann die Kollisionsabfragen im Level durch.

*Performance der
Implementierung*

Damit wir in diesem Kapitel aber auch ein Beispiel sehen, implementiere ich eine Methode, mit der wir die Geometrie aus dem BSP-Baum heraus rendern können. Das sieht nicht besonders schön aus und läuft auch nur mit einer Hand voll Frames pro Sekunde, aber so kann man sich den Level wenigstens einmal anschauen.

Für eben dieses Rendern implementieren wir eine kleine Hilfsklasse. Unsere Mathe-Lib hat ja keine Ahnung von der Render-DLL, also können wir nicht einfach ein Render-Device in den BSP-Baum schieben und dort jedes Leaf rendern lassen. Stattdessen nehmen wir unsere Hilfsklasse, die nicht viel mehr ist als ein halber Stack, auf den man Polygone legen kann – »halb« deswegen, weil wir die Liste nur ganz löschen, aber nicht einzelne Elemente aus ihr entfernen können. In jedem Frame werden wir dann den BSP-Tree

*Hilfsklasse für
Polygonlisten*

⁷ <https://www.gameinstitute.com/gi/courses/courses.asp>

mit dieser Polygonliste durchlaufen und dort alles einsammeln, was gerendert werden muss. Das rendern wir dann extern in unserer Applikation und löschen die Liste wieder für den nächsten Frame. Im Folgenden seht ihr den Code für diese Klasse, der wohl keiner Erklärung bedarf.



```
class ZFXPolylist {
public:
    ZFXPolylist(void);
    ~ZFXPolylist(void);
    bool        AddPolygon(const ZFXPolygon&);
    void        Reset(void);
    ZFXPolygon* GetPolylist(void){return m_pPolys;}
    unsigned int GetNum(void)    {return m_Num; }
private:
    ZFXPolygon  *m_pPolys;
    unsigned int m_Num;
    unsigned int m_Max;
    bool        CheckMem(void);
}; // class

ZFXPolylist::ZFXPolylist(void) {
    m_pPolys = NULL;
    m_Num    = 0;
    m_Max    = 0;
} // constructor

/*-----*/

ZFXPolylist::~ZFXPolylist(void) {
    if (m_pPolys) {
        free(m_pPolys);
        m_pPolys = NULL;
    }
} // destructor

/*-----*/

bool ZFXPolylist::AddPolygon(const ZFXPolygon &Poly) {
    if ( !CheckMem() ) return false;
    m_pPolys[m_Num].CopyOf( Poly );
    m_Num++;
    return true;
} // AddPolygon

/*-----*/

void ZFXPolylist::Reset(void) {
    if (m_pPolys) {
        free(m_pPolys);
        m_pPolys = NULL;
    }
    m_Num = 0;
}
```

```

    m_Max = 0;
} // Reset
/*-----*/

bool ZFXPolylist::CheckMem(void) {
    if (m_Num < m_Max) return true;
    m_Max += 100;
    int nSize = sizeof(ZFXPolygon)*m_Max;
    m_pPolys = (ZFXPolygon*)realloc( m_pPolys, nSize );
    memset(&m_pPolys[m_Num], 0, sizeof(ZFXPolygon)*100);
    return (m_pPolys != NULL);
} // Reset

```

Wie gesagt, es geht hier nicht um Performance oder Eleganz. Die Klasse war nur einfach nötig, um ein exemplarisches Rendern des BSP-Baums zu ermöglichen. Diese Klasse werden wir auch sonst nicht weiter verwenden, sondern nur für die Beispiele dieses Kapitels. Im folgenden Abschnitt sehen wir uns den BSP-Baum an.

Was wirklich zählt

Klassendeklaration

Ehrlich gesagt, sieht so ein BSP-Baum doch schon ganz schön kompliziert aus, oder? Wir haben Ebenen, Frontlists und Backlists, wir müssen Polygone klassifizieren und splitten sowie Splitting-Ebenen auswählen. Aber keine Panik. Es liegt in der Natur von baum-ähnlichen Datenstrukturen aus der Informatik, dass sie sich alle sehr simpel erstellen und durchlaufen lassen. Das Zauberwort der Stunde heißt hier »Rekursion«.

Eine Klasse für sich

Rekursion

*Als **Rekursion** bezeichnet man das Verfahren, bei dem eine Funktion sich in ihrem Rumpf wieder selber aufruft und dadurch rekursiv wird. Ein triviales Beispiel verdeutlicht dieses Vorgehen hoffentlich ein wenig, falls es noch nicht klar sein sollte, was gemeint ist:*

```

void Countdown(int n) {
    if (n <= 0) return;
    printf("%d and counting", n);
    n--;
    Countdown(n);
} // Countdown

```

Man kann nun die Funktion beispielsweise als `Countdown(10)` aufrufen und bekommt den entsprechend herunterzählenden Countdown am Bildschirm zu sehen. Das Gegenstück zur Rekursion nennt man **Iteration**, und die *iterative* Variante der Funktion sieht wie folgt aus:

```
void Countdown(int n) {  
    while (n > 0) {  
        printf("%d and counting", n);  
        n--;  
    }  
} // Countdown
```

Natürlich könnte man auch eine `for`-Schleife verwenden, aber der Unterschied dürfte, glaube ich, klar geworden sein. Wenn wir uns nun etwas komplexere Beispiele, also Funktionen mit mehr Sinn, ansehen würden, so würden wir erkennen, dass die iterative Variante oftmals einen Nachteil hat. Man muss i.d.R. komplexere Mechanismen verwenden, um mehrere Unter-elemente wie zum Beispiel zwei, drei oder acht Äste eines Baum-Nodes jeweils in einer Schleife durchzugehen. Bei strikt gradlinigen Datenstrukturen wie beispielsweise verketteten Listen ist das nicht so von Bedeutung, weil man im Inneren der Schleife nur ein einziges Element bearbeiten muss.

In der rekursiven Variante hat man den Vorteil, dass man einfach die Funktion zur Behandlung eines Objekts schreibt und diese rekursiv für alle Objekte eines Nodes der Struktur aufrufen kann. Natürlich hat auch die Rekursion einen Nachteil: Bevor die Funktion an die Aufrufstelle zurückspringt, ruft sie sich selbst auf und immer so weiter. Das bedeutet, dass die Funktion immer mehr und mehr Daten auf dem Stack ablegt. Der Stack ist ja der Speicherbereich, auf dem der Computer alle lokalen Variablen und die Parameter beim Aufruf einer Funktion anlegt, bis die Funktion zu ihrem Aufrufpunkt zurückkehrt. Dynamisch allozierter Speicher hingegen wird im Heap-Speicher bereitgestellt.

Nun gibt es den berüchtigten Stack-Overflow, wenn der Stack nämlich voll ist. Normalerweise kommt man kaum in die Bereiche solcher Mengen an belegtem Speicher. Aber bei einer Rekursion kommt man in Gefilde, in denen so ein Problem schon einmal auftreten kann. Entweder hat man eine fehlerhafte Endlosrekursion, oder die Rekursion läuft einfach viel zu lange, und es werden in der Funktion zu viele und zu große Parameter und lokale Variablen verwendet. Einen Stack-Overflow zu erzeugen ist trotzdem nicht ganz einfach, aber die Rekursion wird auch schneller, wenn man die Grundregel beachtet, so wenige Parameter und lokale Variablen wie möglich zu verwenden.

BSP-Baum-Klasse

Dank der Rekursion brauchen wir also gar nicht so viele Zeilen Code, wie wir uns schon in unseren schrecklichsten Alpträumen über BSP-Bäume ausgemalt haben. Eher im Gegenteil: Die Klasse ist für die Funktionalität, die sie uns bietet, geradezu lächerlich simpel.



```

class ZFXBspTree {
public:
    ZFXBspTree(void);
    virtual ~ZFXBspTree(void);

    void    BuildTree(const ZFXPolygon*, UINT);
    void    TraverseBtF(ZFXPolylist*, ZFXVector,
                      const ZFXPlane*);
    void    TraverseFtB(ZFXPolylist*, ZFXVector,
                      const ZFXPlane*);

    ZFXAabb GetAabb(void) { return m_Aabb; }

    bool    LineOfSight(const ZFXVector&,
                      const ZFXVector&);

    bool    TestCollision(const ZFXRay&, float,
                      float*, ZFXVector*);
private:
    ZFXAabb    m_Aabb;           // Bounding-Box
    ZFXPlane    m_Plane;        // Splitting-Plane
    ZFXBspTree *m_pBack;        // Backlist
    ZFXBspTree *m_pFront;       // Frontlist
    ZFXBspTree *m_pRoot;        // Root-Node
    ZFXBspTree *m_pParent;      // Parent-Node
    ZFXPolygon *m_pPolys;       // bei Leaf-Node
    UINT        m_NumPolys;     // bei Leaf

    static UINT m_sNum;         // finaler Poly-Count

    void CreateChildren(void);
    bool FindBestSplitter(void);
    void AddPolygon(const ZFXPolygon&);
    void CalcBoundingBox(const ZFXPolygon*, UINT);
    void SetRelationship(ZFXBspTree *R, ZFXBspTree *D)
        {m_pParent = D; m_pRoot = R;}
    bool IsLeaf(void)
        {return (m_pFront==NULL)&&(m_pBack==NULL);}
}; // class

```

Diese eine Klasse ist tatsächlich alles, was wir benötigen. Es gibt Leute, die bevorzugen ein Design, bei dem man eine Klasse für den BSP-Baum und dann noch eine Struktur oder Klasse für einen Node des Baums hat. Das halte ich jedoch für unnötigen Overkill, und so fungiert unsere Klasse primär als Node. Der Root-Node kann also von außen als BSP-Baum interpretiert werden, und daher trägt die Klasse den entsprechenden Namen.

Node und Tree

Die Methoden der Klasse sind größtenteils durch ihre Namen selbsterklärend. Den Code von `ZFXBspTree::CalcBoundingBox` und `ZFXBspTree::AddPolygon` werde ich hier jedoch nicht mehr abdrucken. Wie man aus einer Menge an

Methoden

Polygonen eine achsen-ausgerichtete Bounding-Box berechnet, sollte nun wirklich jeder wissen. Das Hinzufügen eines Polygons zu einem Node ist ebenso trivial. Wir erweitern bei Bedarf den dynamisch allokierten Speicher des Attributs `m_pPolys` des Nodes und kopieren das Polygon in diese Liste. Unser Code sieht nur schlanker und verständlicher aus, wenn wir diesen Dreizeiler in eine eigene Methode auslagern. Dann gibt es noch zwei `TraverseXY`-Methoden. Die eine macht einen Back-to-Front-Durchlauf durch den Baum und die andere dann entsprechend einen Front-to-Back-Durchlauf. Was es damit auf sich hat, klären wir, wenn wir die Methoden implementieren.

Erzeugen und Freigeben einer Instanz

Konstruktor und Destruktor

Die Konstruktion und Destruktion einer Instanz der Klasse `ZFXBspTree` führen wir wie gewohnt durch. Alle Attribute erhalten Initialwerte im Konstruktor, und potenziell allozierter Speicher wird im Destruktor freigegeben. Man beachte hierbei, dass wir für die Polygonliste auch weiterhin die C-Funktionen `malloc()`, `realloc()` und `free()` verwenden, da wir den Speicher im Erstellungsprozess des Baums dynamisch erweitern müssen. Unserer Engine fehlt leider aufgrund des begrenzten Platzes in diesem Buch ein eigener Speicher-Manager, der natürlich effizienter und eleganter wäre.



```
ZFXBspTree::ZFXBspTree(void) {
    m_NumPolys = 0;
    m_pBack    = NULL;
    m_pFront   = NULL;
    m_pRoot    = NULL;
    m_pParent  = NULL;
    m_pPolys   = NULL;
} // constructor

/*-----*/

ZFXBspTree::~ZFXBspTree(void) {
    m_NumPolys = 0;

    SAFE_FREE(m_pPolys);
    SAFE_DELETE(m_pFront);
    SAFE_DELETE(m_pBack);
} // destructor
```

Rekursives Erstellen des Baums

Kick-Off-Methode

Da unsere Klasse ja intern sozusagen als Schablone für einen Node des BSP-Baums dient und extern nur als ganzer BSP-Baum in Erscheinung tritt, bieten wir dem Anwender eine Methode an, mit der er den Erstellungsprozess des Baums anstoßen kann. Dazu muss der Anwender lediglich ein Array von `ZFXPolygon`-Objekten und die Anzahl der Einträge in dem Array bereithalten. Die Klasse `ZFXBspTree` erstellt dann daraus den BSP-Baum für ihn. Die

Kick-Off-Methode für diesen Prozess, also `ZFXBspTree::BuildTree` allokiert daraufhin dynamisch den Speicher für das Attribut `m_pPolys` und kopiert die Polygonliste in dieses Attribut. Man beachte, dass unsere Klasse `ZFXPolygon` selbst Attribute mit dynamisch allokiertem Speicher enthält, daher können wir die gesamte Liste nicht auf einen Rutsch kopieren, sondern müssen Polygon für Polygon vorgehen.

Danach beginnt der eigentliche Erstellungsprozess. Die Kick-Off-Methode ruft die Methode `ZFXBspTree::CreateChildren` auf. Diese übernimmt die Erstellung des Baums ausgehend von der Polygonliste, die nun in dem Wurzel-Objekt gespeichert ist.

```
void ZFXBspTree::BuildTree(const ZFXPolygon *pPolys,
                           UINT Num)
{
    m_pRoot = this;
    m_pParent = NULL;

    if (Num<1) return;

    // Speicher für die Polygonliste allokieren
    int nSize = sizeof(ZFXPolygon)*Num;
    m_pPolys = (ZFXPolygon*)malloc(nSize);
    memset(m_pPolys, 0, nSize);
    m_NumPolys = Num;

    for (UINT i=0; i<Num; i++)
        m_pPolys[i].CopyOf( pPolys[i] );

    // Rekursion starten
    CreateChilds();
} // BuildTree
```



Nach dem Kick-Off durch `ZFXBspTree::BuildTree` haben wir eine Instanz der Klasse `ZFXBspTree`, die nach außen hin als einzige sichtbar bzw. verwendbar ist. Ausgehend von diesem Objekt beginnen wir nun die Erstellung des Baums. Als Erstes berechnen wir für die Polygonliste des Nodes die Bounding-Box. Danach versuchen wir, mit `ZFXBspTree::FindBestSplitter` einen Splitter in der Liste der Polygone des Nodes zu finden. Liefert diese Methode `false` zurück, dann konnte kein geeigneter Splitter gefunden werden. In diesem Fall können wir einen Ast der Rekursion beenden, denn diese Node ist dann ein Leaf mit konvexer Geometrie. Liefert die Methode jedoch `true` zurück, dann hat sie einen geeigneten Splitter gefunden, markiert und dessen Ebene im Attribut `m_Plane` des Nodes gespeichert. Wir müssen also die Geometrie an diesem Node noch weiter auf die beiden Children in der Frontlist und Backlist aufteilen. Dazu erzeugen wir die entsprechenden Instanzen in den Attributen `m_pFront` und `m_pBack`, die von derselben Klasse `ZFXBspTree` sind.

*Rekursive
Erstellung*

Teilen der Geometrie

Das Aufteilen der Geometrie auf die beiden Children ist nun sehr einfach. Wir laufen durch die Liste von Polygonen an dem Node und klassifizieren jedes Polygon in Relation zu der Ebene des ausgewählten Splitters. Liegt das Polygon vollkommen auf der Front-Seite der Ebene, dann kommt es in die Frontlist. Dasselbe gilt analog für die Backlist. Wenn das Polygon genau in der Ebene liegt, dann prüfen wir den Winkel zwischen seiner Normalen und der Normalen der Ebene. Zeigen beide in dieselbe Richtung, kommt das Polygon in die Frontlist, sonst in die Backlist. Der letzte mögliche Fall ist nun, dass das Polygon die Ebene schneidet. Dann schneiden wir auch, und zwar das Polygon an der Ebene in zwei Teile. Der Teil auf der Front-Seite kommt als neues Polygon in die Frontlist, der Teil auf der Back-Seite als neues Polygon in die Backlist.

Polygonliste freigeben

Das war es auch schon. Jetzt löschen wir die Polygonliste des Nodes, denn für innere Nodes brauchen wir ja später nur die Splitter-Ebene und keine Polygone. Anschließend rufen wir für die beiden Children des Nodes die Methode `ZFXBspTree::CreateChilds` rekursiv auf und klopfen uns auf die Schulter, denn nun wird der BSP-Baum komplett für uns erstellt.



```
void ZFXBspTree::CreateChilds(void) {
    float      fDot=0.0f;
    ZFXPolygon plyFront, plyBack;
    int        nFront=0, nBack=0, nClass=0;

    CalcBoundingBox(m_pPolys, m_NumPolys);

    // wenn kein Splitter, dann ist das ein Leaf
    if ( !FindBestSplitter() ) {
        ZFXBspTree::m_sNum += m_NumPolys;
        return;
    }

    // erzeuge Objekte für die beiden Children
    m_pFront = new ZFXBspTree;
    m_pBack  = new ZFXBspTree;
    m_pFront->SetRelationships(m_pRoot, this);
    m_pBack->SetRelationships(m_pRoot, this);

    // sortiere Polygone in Children ein
    for (UINT i=0; i<m_NumPolys; i++) {

        nClass = m_Plane.Classify( m_pPolys[i] );

        if (nClass == ZFXFRONT) {
            m_pFront->AddPolygon( m_pPolys[i] );
        }
        else if (nClass == ZFXBACK) {
            m_pBack->AddPolygon( m_pPolys[i] );
        }
    }
}
```



```

else if (nClass == ZFXCLIPPED) {
    // splitte Polygon an der Split-Ebene
    m_pPolys[i].Clip(m_Plane, &plyFront, &plyBack);

    m_pFront->AddPolygon( plyFront );
    m_pBack->AddPolygon( plyBack );
}
else if (nClass == ZFXPLANAR) {

    fDot = m_Plane.m_vcN *
           m_pPolys[i].GetPlane().m_vcN;

    if ( fDot >= 0.0f ) {
        m_pFront->AddPolygon( m_pPolys[i] );
    }
    else {
        m_pBack->AddPolygon( m_pPolys[i] );
    }
}
} // for

// lösche Polygonliste in inneren Nodes
SAFE_FREE(m_pPolys);

// REKURSIVE AUFRUFE
m_pFront->CreateChilds();
m_pBack->CreateChilds();
} // CreateChilds

```

Diese Methode verwendet keine Parameter, was für die Rekursion gut ist, weil wir so den Stack schonen. Allerdings haben wir doch eine Anzahl an lokalen Variablen. Ich lasse das mal bewusst so stehen, um ein wenig Optimierungspotenzial aufzuzeigen. Der `float`-Wert wäre beispielsweise nicht unbedingt nötig, da wir die Berechnung direkt in die `if`-Abfrage stecken könnten. Alternativ könnte man die Deklaration der Variablen auch in den entsprechenden Fall des planaren Polygons verschieben, so dass sie wirklich nur erzeugt wird, wenn sie notwendig ist. Die drei `int`-Zähler und die beiden Instanzen von `ZFXPolygon` könnte man als `static`-Attribute der Klasse umsetzen. Damit hätten wir dem Stack schon wieder ein wenig Arbeit erspart.

*Optimierungen für
eine schnelle
Rekursion*

So ganz sind wir damit noch nicht fertig, was die Erstellung des Baums angeht. Bisher haben wir die Auswahl eines geeigneten Splitters noch nicht betrachtet bzw. nur als Pseudo-Code gesehen. Also erledigen wir das auch noch schnell.

*Auswahl des
Splitters*

Auswahl des besten Splitters

Piece of Cake

Wenn wir eine Liste von Polygonen haben, dann finden wir den Splitter ja wie folgt: Wir laufen durch die Liste, und in jedem Durchlauf der Schleife nehmen wir das entsprechende Polygon quasi testweise als Splitter. Dann starten wir eine innere Schleife nochmals über die gesamte Liste von Polygonen und klassifizieren alle Polygone für den entsprechenden Test-Splitter. Dabei zählen wir mit, wie viele Polygone jeweils direkt in die Frontlist und die Backlist rutschen würden und wie viele Polygone wir an der Ebene des Splitters splitten müssten. Daraus errechnen wir nach der weiter oben beschriebenen Heuristik eine Punktbewertung für den potenziellen Splitter und merken uns während der ganzen Schleiferei immer den Splitter mit der besten Punktbewertung, also der niedrigsten Punktzahl. Denn die Punkte sind sozusagen Strafpunkte. Insbesondere werden viele Splits hoch bestraft.

Splitter-Ebene speichern

Am Ende der Methode prüfen wir noch einmal, ob wir überhaupt eine Aufteilung erreichen konnten. Es muss in der Frontlist und der Backlist jeweils mindestens ein Polygon geben. Sonst ist kein geeigneter Splitter zu finden, weil die Geometrie konvex ist. Wenn wir aber einen Splitter aufgetrieben haben, dann speichern wir dessen Ebene im Attribut `m_Plane` des Nodes und sind mit der Arbeit fertig.



```
bool ZFXBspTree::FindBestSplitter(void) {
    ZFXPolygon *pBestSplitter=NULL, *pSplitter=NULL;
    ZFXPlane    Plane;
    LONG        lFront  = 0,        // Wie viele Polys liegen
                lBack   = 0,        // bei jedem potenziellen
                lPlanar = 0,        // Splitter in Front, Back
                lSplits = 0;        // Planar oder spannend?

    int         nClass;
    LONG        lScore, lBestScore = 1000000;
    bool        bFound = false;

    for (UINT i=0; i<m_NumPolys; i++) {

        pSplitter = &m_pPolys[i];
        Plane      = pSplitter->GetPlane();

        // Zähler zurücksetzen
        lFront = lBack = lPlanar = lSplits = 0;

        // Schon mal gesplittet?
        if ( pSplitter->GetFlag() == 1 ) continue;

        // teste alle Polygone mit potenziellem Splitter
        for (UINT j=0; j<m_NumPolys; j++) {
            if (i==j) continue;

            nClass = Plane.Classify( m_pPolys[j] );
```

```

        if ( nClass == ZFXFRONT ) lFront++;
        else if ( nClass == ZFXBACK ) lBack++;
        else if ( nClass == ZFXPLANAR ) lPlanar++;
        else
            lSplits++;
    } // for

    // BERECHNE DEN PUNKTWERT
    lScore = abs(lFront - lBack) + (lSplits * 3);

    if (lScore < lBestScore) {
        if ( ((lFront > 0) && (lBack > 0)) ||
            (lSplits > 0) ) {
            lBestScore = lScore;
            pBestSplitter = pSplitter;
            bFound = true;
        }
    } // if [ulScore]
} // for

// kein Splitter mehr zu finden
if ( !bFound ) return false;

// markiere Polygon, speichere Split-Ebene
pBestSplitter->SetFlag(1);
m_Plane = pBestSplitter->GetPlane();
return true;
} // FindBestSplitter

```

Den Punktwert eines Splitters kann man natürlich auch anders berechnen. Wenn man einen strikt ausgewogenen Baum für die Stabilisierung der Suchzeiten haben möchte, dann muss man die Anzahl der Splits einfach ignorieren. Wenn man jedoch Splits vermeiden möchte, um den Baum zwar unausgewogen, dafür aber insgesamt kleiner zu halten, dann muss man die Anzahl der Splits noch stärker bestrafen, also den Faktor von 3 noch weiter erhöhen. Einen idealen Kompromiss gibt es aber nicht.

*Andere
Heuristiken*

Durchlaufen des Baums

Hoppla, da ist unser BSP-Baum doch schon fertig. Na gut, nun brauchen wir eine Methode, mit der wir den Baum durchlaufen können. Dabei wollen wir in einer Instanz der Klasse `ZFXPolylist` alle Polygone aus den Leaves sammeln, die von Bedeutung sind. Die Bedeutung oder Nicht-Bedeutung der konvexen Polygon-Happen ist dabei von zwei Dingen abhängig: zum einen von der Position des Betrachters und zum anderen vom View-Frustum. Mit Letzterem kann man natürlich Nodes und damit alle Leaves und Geometrie frühzeitig cullen, die nicht im View-Frustum des aktuellen Frames liegen. Anhand der Position des Betrachters kann man die Polygone des BSP-Baums beim Traversieren ordnen, und zwar von hinten nach vorne (engl. back to

BSP-Tricks

front) oder von vorne nach hinten (engl. front to back), also nach ihrer Entfernung in Relation zum Betrachter. Wie wir in unserer Demo-Applikation später sehen werden, kann man die Polygone eines Levels vollkommen ohne Depth-Buffer rendern, wenn man die Back-to-Front-Sortierung verwendet. Das war ja der Sinn, den der BSP-Baum früher einmal hauptsächlich erfüllen sollte. Wenn man beispielsweise ein Modell mit Transparenzeffekten rendern möchte, kann man diese Technik natürlich auch verwenden, um die Polygone entsprechend zu sortieren, damit keine Alphablending-Fehler durch falsche Sortierung entstehen.

Front-to-Back

Wenn man Front-to-Back rendert, spart man heutzutage hingegen viel an Overdraw ein, da die ganz vorn zum Betrachter hin liegenden Teile des Levels zuerst gerendert werden. Die verdeckten Teile des Levels werden dann vom Depth-Buffer einer modernen Grafikkarte sofort abgeschmettert. Heute macht es also mehr Sinn, eine Szene front-to-back zu rendern, weil das eine bessere Performance ergibt. Auch dafür kann man einen BSP-Baum verwenden.

Rekursive Suche in Bäumen

Aber genug geredet. Wir schreiben nun eine Methode für die Front-to-Back-Traversion des BSP-Baums. Als Erstes führen wir natürlich ein Culling mit dem Frustum durch. Ist der Node nicht vollkommen außerhalb, dann testen wir, ob es sich um ein Leaf handelt. Wenn es ein Leaf ist, dann fügen wir alle Polygone des Leafs zu der `ZFXPolylist` hinzu, die der Aufrufer angegeben hat. Anderenfalls sind wir an einem inneren Node, und hier wird es interessant, denn wir müssen nun darauf achten, in welches der beiden Children wir zuerst abtauchen, um die gewünschte Sortierung zu erhalten. Wir klassifizieren die Position des Betrachters gegen die Splitter-Ebene des Nodes. Wenn sie auf der Back-Seite liegt, bedeutet das, dass mindestens ein Polygon auf der Back-Seite der Ebene näher an dieser Position des Betrachters liegen muss als das am nächsten liegende Polygon auf der Front-Seite der Ebene. Also tauchen wir zuerst rekursiv in die Backlist des Nodes ab. Liegt die Position des Betrachters auf der Front-Seite, so starten wir die Rekursion natürlich in der Frontlist.

Den Fall der Position, die planar zu der Ebene ist, behandeln wir beliebig, hier analog zu dem Fall Front.



```
void ZFXBspTree::TraverseFtB(ZFXPolylist *pList,
                             ZFXVector vcPos,
                             const ZFXPlane *Frustum) {
    // Frustum-Culling für diesen Node
    if (m_Aabb.Cull(Frustum,6 ) == ZFXCULLED) return;

    // Leaves enthalten die Polygone
    if ( IsLeaf() ) {
        for (UINT i=0; i<m_NumPolys; i++) {
            pList->AddPolygon( m_pPolys[i] );
        }
    }
}
```

```

    }
}
else {
    int nClass = m_Plane.Classify( vcPos );

    if (nClass == ZFXBACK) {
        m_pBack->TraverseFtB(pList, vcPos, Frustum);
        m_pFront->TraverseFtB(pList, vcPos, Frustum);
    }
    else {
        m_pFront->TraverseFtB(pList, vcPos, Frustum);
        m_pBack->TraverseFtB(pList, vcPos, Frustum);
    }
}
} // TraverseFtB

```

Nun drehen wir die Sortierung einfach um. Wenn wir eine Back-to-Front-Sortierung haben möchten, dann müssen wir natürlich die von der Position des Betrachters am weitesten entfernten Polygone zuerst aufspüren. Das machen wir, indem wir natürlich jeweils in den entgegengesetzten Ast eines Nodes abtauchen. Wenn die Position des Betrachters auf der Front-Seite liegt, dann gehen wir zuerst rekursiv in die Backlist des Nodes und vice versa.

Back-to-Front

```

void ZFXBspTree::TraverseBtF(ZFXPolylist *pList,
                             ZFXVector vcPos,
                             const ZFXPlane *Frustum) {
    // Frustum-Culling für diesen Node
    if (m_Aabb.Cull(Frustum,6 ) == ZFXCULLED) return;

    // Leaves enthalten die Polygone
    if ( IsLeaf() ) {
        for (UINT i=0; i<m_NumPolys; i++) {
            pList->AddPolygon( m_pPolys[i] );
        }
    }
    else {
        int nClass = m_Plane.Classify( vcPos );

        if (nClass == ZFXBACK) {
            m_pFront->TraverseBtF(pList, vcPos, Frustum);
            m_pBack->TraverseBtF(pList, vcPos, Frustum);
        }
        else {
            m_pBack->TraverseBtF(pList, vcPos, Frustum);
            m_pFront->TraverseBtF(pList, vcPos, Frustum);
        }
    }
} // TraverseBtF

```



Kollisionsabfragen

Und ewig kollidiert der Strahl

Wenn wir den BSP-Baum schon nicht zum Rendern verwenden, dann führen wir doch wenigstens eine ordentliche Kollisionsabfrage mit dem Baum durch. Nach dem Erstellungsprozess repräsentiert der BSP-Baum eine Darstellung eines Indoor-Levels, die wir optimal für Kollisionsabfragen einsetzen können. Wir nehmen uns hier wieder unseren ZFXRay-Strahl vor und möchten von der ZFXBspTree-Klasse wissen, ob dieser Strahl innerhalb einer bestimmten definierbaren Länge mit irgendeinem Polygon der Level-Geometrie kollidiert. Wir gehen dabei davon aus, dass der Ursprungspunkt des Strahls innerhalb des Levels liegt.

Kollisionen in den Leaves

Eine Kollision können wir natürlich nur in einem Leaf feststellen, denn nur dort sind ja die Polygone des Baums gespeichert. Doch wie gehen wir mit dem Strahl in ein Leaf? Nun, das Ganze ist wieder einmal sehr einfach, denn es funktioniert genauso wie das Einsortieren eines Polygons in den BSP-Baum. Wir schieben den Strahl in die Wurzel des Baums und klassifizieren ihn an jedem Node mit der dort gespeicherten Splitter-Ebene. Wenn der Strahl die Ebene schneidet, zerlegen wir ihn in zwei Teile und senden diese jeweils in das entsprechende Child des Nodes mit der Frontlist bzw. der Backlist. Wenn der Strahl die Ebene nicht schneidet, dann kommt er nur in die eine entsprechende Liste, je nachdem auf welcher Seite der Ebene er komplett liegt. Das ist schon das ganze Geheimnis. Wenn wir dann mit einem Stück des Strahls in einem Leaf landen, so testen wir dort jedes einzelne Polygon darauf, ob der Strahl es schneidet. Dann liegt eine Kollision vor.



```
bool ZFXBspTree::TestCollision(const ZFXRay &Ray,
                               float fL, float *pFD,
                               ZFXVector *pvcN) {
    ZFXRay rayFront, rayBack;
    int    nFront=0;

    // WIR SIND IN EINEM LEAF
    if ( IsLeaf() ) {

        for (UINT i=0; i<m_NumPolys; i++) {
            // Kollision mit einem Polygon?
            if ( m_pPolys[i].Intersects(Ray,false,fL,0) )
            {
                if (pvcN)
                    *pvcN = m_pPolys[i].GetPlane().m_vcN;
                return true;
            }
        } // for
        return false;
    }
}
```

```

// SONST SIND WIR IN EINEM INNEREN NODE
int nClass = m_Plane.Classify( Ray.m_vcOrig );

// Schneidet der Strahl die Split-Plane?
if ( m_Plane.Clip(&Ray, fL, &rayFront, &rayBack) ) {
    // Suche in Front-to-Back-Reihenfolge
    if ( nClass == ZFXBACK )
        return m_pBack->TestCollision(rayBack, fL,
                                       pfD, pvcN)
        || m_pFront->TestCollision(rayFront, fL,
                                   pfD, pvcN);
    else
        return m_pFront->TestCollision(rayFront, fL,
                                       pfD, pvcN)
        || m_pBack->TestCollision(rayBack, fL,
                                   pfD, pvcN);
}
else {
    if ( nClass == ZFXBACK )
        return m_pBack->TestCollision(Ray,fL,pfD,pvcN);
    else
        return m_pFront->TestCollision(Ray,fL,pfD,pvcN);
}
} // TestCollision [ray]

```

Wichtig ist hierbei auch, dass wir wie bei der Traversal des Baums die Reihenfolge beachten, in der wir die beiden Teilstücke des Strahls in die beiden Children schieben, wenn der Strahl die Ebene geschnitten hat. Wir wollen ja i.d.R. den Kollisionspunkt finden, der am dichtesten am Ursprung des Strahls liegt. Folglich bemühen wir dazu das Front-to-Back-Verfahren.

*Erster
Kollisionspunkt*

Mit dieser Methode zum Testen des BSP-Baums auf eine Kollision mit einem Strahl können wir auch eine andere hilfreiche Funktion in den BSP-Baum integrieren. Wenn wir beispielsweise zwei Entities im Level haben, etwa den Spieler und ein Monster, dann ist es insbesondere für die künstliche Intelligenz wichtig herauszufinden, ob das Monster den Spieler sehen kann. Dazu muss es eine gültige Sichtlinie (engl. line of sight) zwischen den Positionen der beiden Entities geben. Unsere Klasse ZFXBspTree bietet dem Anwender nun die folgende Methode an, um zu evaluieren, ob es zwischen zwei Punkten im Level eine Sichtlinie gibt. Ist das nicht der Fall, weil ein Teil der Geometrie die Sicht blockiert, dann gibt die Methode `false` zurück.

*Sichtlinie
zwischen zwei
Punkten*

```

bool ZFXBspTree::LineOfSight(const ZFXVector &vcA, const ZFXVector &vcB) {
    ZFXRay Ray;

```

```

    // Strahl von A nach B
    ZFXVector vcDir = vcB - vcA;
    vcDir.Normalize();
    Ray.Set( vcA, vcDir );

```



```
// Strahl auf Kollision testen  
return !TestCollision(Ray,vcDir.GetLength(),0,0);  
} // LineOfSight
```

Sichtlinie Wir berechnen also einen Vektor, der von Punkt A zu Punkt B läuft. Dies wird der Richtungsvektor für einen Strahl, den wir von der Position A aus zum Punkt B schießen. Die Länge des Strahls ist natürlich die Distanz zwischen den beiden Punkten. Nun können wir die eben entwickelte Methode `ZFXBspTree::TestCollision` einsetzen, um zu überprüfen, ob man vom Punkt A aus eine freie Sichtlinie auf Punkt B hat oder nicht.

13.4 Implementierung eines Octrees

Nichts als Bäume Nachdem wir nun den BSP-Baum gemeistert haben, mausern wir uns langsam zu einem echter Gärtner und nutzen den uns gegebenen grünen Daumen gleich richtig aus. Wir pflanzen in unsere Mathe-Lib nun auch gleich einen *Achterbaum*, von einigen auch vornehm Octree genannt, ein. Im Anschluss daran konstruieren wir eine Demo-Applikation, um die beiden Bäume in Action zu sehen.

Praxis Im ersten Teil dieses Kapitels haben wir ja schon einiges zur Theorie und zum Sinn und Zweck von Octrees im Rahmen des Scene-Managements gehört. In diesem Teil geht es nun darum, eine Klasse für den Octree zu implementieren, die wir später in unserem Spiel nutzen können. Ich möchte aber betonen, dass wir den Octree ausdrücklich nicht dazu verwenden werden, für einen gegebenen Frame die Menge an Polygonen zu bestimmen, die sichtbar sind und gerendert werden müssen. Das würde darauf hinauslaufen, dass wir in jedem Frame wieder dynamisch eine Menge an Vertices – oder wenigstens Indices – über den Bus jagen müssten. Und das wollen wir vermeiden. Wir verwenden den Octree in unserem späteren Spiel ausschließlich für Kollisionsabfragen. Es wird dort eine strikte Trennung zwischen der Geometrie zum Rendern auf der Grafikkarte und einer Kopie der Geometrie in einem Octree im System-RAM für die Kollisionsabfragen geben. Das erhöht zwar den Speicherverbrauch, macht aber das Rendern um einiges effizienter.

Richtungsweisend Wie wir wissen, hat ein Octree-Node acht Kinder, falls es sich bei dem Node nicht um ein Leaf handelt, das dann gar keine Kinder mehr hätte. Unsere Klasse für den Octree hat entsprechend ein Array von acht Nodes, in dem wir die Kinder speichern können. Damit wir aber nicht mit Indices von 0 bis 7 um uns werfen müssen und durcheinander kommen, wenn es auf einen bestimmten Index ankommt, definieren wir die folgenden sprechenden Begriffe für den Zugriff auf dieses Array.


```
#define POLYS_PER_LEAF 10
```

```
// Benenne die Kinder
#define UP_NE 0 // oberes Nordost-Kind
#define UP_NW 1 // oberes Nordwest-Kind
#define UP_SE 2 // oberes Südost-Kind
#define UP_SW 3 // oberes Südwest-Kind
#define LW_NE 4 // unteres Nordost-Kind
#define LW_NW 5 // unteres Nordwest-Kind
#define LW_SE 6 // unteres Südost-Kind
#define LW_SW 7 // unteres Südwest-Kind
```



*Anzahl der
Polygone je Leaf*

Eine Geometrie zerlegen wir nun in einen Octree, indem wir sie immer weiter gleichmäßig in acht Würfel zerlegen, bis – ja, bis was eigentlich geschieht? Es gibt zwei populäre Möglichkeiten, diesen schier unendlichen Algorithmus terminieren zu lassen. Die erste Möglichkeit ist, eine vorgegebene Anzahl an Zerlegungen zu bestimmen. Man kann beispielsweise sagen, dass ein Ast des Octrees maximal eine Tiefe von n haben soll, wobei n ein beliebiger Wert ist. Nachdem man die Geometrie also n -mal zerlegt hat, stoppt man dort einfach. Damit kann man die Komplexität des Octrees direkt steuern, allerdings ignoriert man auf diese Weise die Komplexität der Geometrie vollkommen. Ein Leaf des Octrees kann also ein einziges Polygon beinhalten oder zehntausend. Wir wählen daher die zweite Möglichkeit, die geometrie-orientiert arbeitet. Wir beenden die Unterteilung eines Nodes in acht weitere Children genau dann, wenn an dem Node nur noch n Polygone vorhanden sind. Wie oben zu sehen ist, definieren wir dafür den Bezeichner `POLYS_PER_LEAF`. So können wir während der Kompilierung des Programms festlegen, wie viele Polygone maximal in einem Leaf sein dürfen.

Performance

Je weiter man den Octree unterteilt, desto größer werden natürlich die Suchzeiten im Octree, bis man einen bestimmten Node erreicht hat. Wenn man jedoch mit der Geometrie eines Leafs komplexere Berechnungen durchführen möchte, so ist es besser, eine längere Suchzeit zu einem Leaf in Kauf zu nehmen und dort weniger komplexe Berechnungen mit weniger Polygonen durchzuführen. Die andere Option wäre eine schnelle Suche des Leafs, die aber dann viele komplexe Berechnungen erfordern würde, weil das Leaf sehr viele Polygone enthält. Durch unser `#define` können wir später ohne Probleme verschiedene Szenarien ausprobieren und schauen, wie sich das auf die Performance auswirkt. Jetzt implementieren wir den Octree aber erst einmal.

Klassen-Deklaration

Im Folgenden ist die Klassen-Deklaration für den Octree abgedruckt. Die öffentlichen Methoden können wir an einer Hand abzählen. Neben dem Konstruktor haben wir zwei Methoden für Kollisionsabfragen: eine, um die

*Wofür Render-
Methoden?*

Höhe des Bodens für eine bestimmte Position zu ermitteln und eine Methode, um den Tree überhaupt zu erstellen. Dazu kommt dann noch eine Traversions-Methode, um den Baum zu durchlaufen und die sichtbaren Polygone für ein gegebenes View-Frustum einzusammeln. Wie bei dem BSP-Baum auch verwenden wir die so generierte Liste dann zum Rendern des Octrees. Die Render-Methode der Demo-Applikation ist aber explizit nicht dazu da, große Level zu rendern. Sie dient ausschließlich dem Zweck, Teile eines Levels über den Octree zu rendern, um die Erstellung des Octrees zu überprüfen. Sie ist extrem langsam, weil sie jedes einzelne Polygon getrennt zum Rendern schickt, und bei der Menge an Polygonen verhilft uns selbst das Batching in unserer Render-DLL nicht mehr zu guter Performance. Hier ist die Klassen-Deklaration für ZFXOctree:



```
class ZFXOctree {
public:
    ZFXOctree(void);
    virtual ~ZFXOctree(void);

    void    BuildTree(const ZFXPolygon*, UINT);
    void    Traverse(ZFXPolylist*, ZFXPolylist*,
                    const ZFXPlane*);
    ZFXAabb GetAabb(void) { return m_Aabb; }

    bool    GetFloor(const ZFXVector&, float*,
                    ZFXPlane*);
    bool    TestCollision(const ZFXAabb&, ZFXPlane*);
    bool    TestCollision(const ZFXRay&, float,
                    float*);

private:
    ZFXAabb    m_Aabb;           // Bounding-Box
    ZFXPolygon *m_pPolys;       // Falls Leaf
    UINT       m_NumPolys;      // Falls Leaf
    ZFXOctree *m_pChild[8];     // 8 Child-Nodes
    ZFXOctree *m_pRoot;         // Root-Node
    ZFXOctree *m_pParent;       // Root-Node
    int        m_Pos;           // NO, NW, ...

    void CalcBoundingBox(const ZFXPolygon*, UINT);
    void InitChildObject(int ChildID, ZFXOctree *pP);
    void ChopListToMe(ZFXPolygon*, UINT);
    void CreateChilds(ZFXOctree *pRoot);
    void GetAabbAsPolygons(ZFXPolylist*);
    bool IntersectsDownwardsRay(const ZFXVector&,
                                float);

    bool IsLeaf(void) { return (m_pChild[0]==NULL); }
    void SetBoundingBox(const ZFXAabb &Aabb)
        { memcpy(&m_Aabb, &Aabb, sizeof(ZFXAabb)); }
}; // class
```

Auch an Attributen braucht die Klasse nicht viel. Es gibt eine achsen-ausgerichtete Bounding-Box, eine Liste von Polygonen zusammen mit einem Zähler, die aber nur bei Leaves verwendet werden, sowie ein Array von acht Children, die nur bei Leaves nicht verwendet werden. Hier können wir auch schon sehen, dass wir unseren Octree nicht in eine Klasse für den Octree und eine Klasse für die Nodes unterteilen. Das ist bei einer so einfachen Struktur vollkommen unnötig, da die Nodes mit Ausnahme der Methode `ZFXOctree::BuildTree` exakt dieselben Methoden verwenden würden wie die Dachklasse. Von daher ist eine solche Trennung nicht wirklich erforderlich. Zu rein informativen Zwecken hat die Klasse noch einen Pointer auf die Wurzel des Octrees sowie einen Pointer auf den Vater (engl. Parent), sofern es sich nicht um die Wurzel handelt. Im Attribut `m_Pos` merkt sich ein Node zudem, welche Child-Position er aus Sicht seines Vaters einnimmt.

Attribute

Den Sinn und Zweck der meisten privaten Methoden der Klasse werden wir im Folgenden ergründen. Zu der hier implementierten Abfrage, ob es sich bei der Instanz um ein Leaf handelt, sei nochmals erwähnt, dass ein Node entweder acht Children hat oder gar keins. Wenn also der Pointer auf eines seiner Children `NULL` ist, dann kann man davon ausgehen, dass die anderen auch `NULL` sind und es sich bei dem Node um ein Leaf handelt.

Private Methoden

Erzeugen und Freigeben einer Instanz

Das Erzeugen einer Instanz über den Konstruktor ist genauso unspektakulär wie das Zerstören einer Instanz über den Destruktor. Im ersten Fall setzen wir die Attribute auf sinnvolle Startwerte, und im zweiten Fall geben wir einfach sämtlichen Speicher frei, den wir im Verlauf des Octrees potenziell dynamisch belegt haben. Dies ist bei den Leaves die Liste von Polygonen und bei allen anderen Nodes der Speicher für die acht Child-Objekte derselben Klasse.

Nichts zu tun

```
ZFXOctree::ZFXOctree(void) {
    m_NumPolys = 0;
    m_Pos      = -1;
    m_pPolys   = NULL;
    m_pRoot    = NULL;
    m_pParent   = NULL;

    for (int i=0; i<8; i++)
        m_pChild[i] = NULL;

    memset(&m_Aabb, 0, sizeof(ZFXAabb));
} // constructor

/*-----*/

ZFXOctree::~ZFXOctree(void) {
    m_NumPolys = 0;
```



```
SAFE_DELETE_A(m_pPolys);

for (int i=0; i<8; i++) {
    SAFE_DELETE(m_pChild[i]);
}
} // destructor
```

Initialisieren eines Child-Nodes

*Ordnung muss
sein*

Bevor wir nun endlich, endlich mit der Implementierung anfangen, schreiben wir erst mal eine kleine Hilfsmethode, damit wir sie nachher gleich zur Hand haben. Die folgende Methode dient einer Instanz unserer Klasse dazu, eines der acht Children von sich zu initialisieren. Die Initialisierung betrifft dabei die Berechnung der Bounding-Box. Wir kennen ja die Bounding-Box des Nodes, und da die Children diese brüderlich gleichmäßig unter sich aufteilen, können wir für ein gegebenes Child exakt vorherberechnen, wie seine Bounding-Box aussehen wird. Mehr tun wir in dieser Methode nicht. Zusätzlich wird in jedem Child der Pointer auf seinen Vater gespeichert und angegeben, welche Position er aus dessen Sicht innehat.



```
void ZFXOctree::InitChildObject(int ChildID,
                                ZFXOctree *pParent) {

    ZFXAabb aabb;

    float xmin = m_Aabb.vcMin.x,
          xcen = m_Aabb.vcCenter.x,
          xmax = m_Aabb.vcMax.x;

    float ymin = m_Aabb.vcMin.y,
          ycen = m_Aabb.vcCenter.y,
          ymax = m_Aabb.vcMax.y;

    float zmin = m_Aabb.vcMin.z,
          zcen = m_Aabb.vcCenter.z,
          zmax = m_Aabb.vcMax.z;

    switch(ChildID) {
        case UP_NW:
            aabb.vcMax = ZFXVector(xcen, ymax, zmax);
            aabb.vcMin = ZFXVector(xmin, ycen, zcen);
            break;
        case UP_NE:
            aabb.vcMax = m_Aabb.vcMax;
            aabb.vcMin = m_Aabb.vcCenter;
            break;
        case UP_SW:
            aabb.vcMax = ZFXVector(xcen, ymax, zcen);
            aabb.vcMin = ZFXVector(xmin, ycen, zmin);
```

```

        break;
    case UP_SE:
        aabb.vcMax = ZFXVector(xmax, ymax, zcen);
        aabb.vcMin = ZFXVector(xcen, ycen, zmin);
        break;
    case LW_NW:
        aabb.vcMax = ZFXVector(xcen, ycen, zmax);
        aabb.vcMin = ZFXVector(xmin, ymin, zcen);
        break;
    case LW_NE:
        aabb.vcMax = ZFXVector(xmax, ycen, zmax);
        aabb.vcMin = ZFXVector(xcen, ymin, zcen);
        break;
    case LW_SW:
        aabb.vcMax = m_Aabb.vcCenter;
        aabb.vcMin = m_Aabb.vcMin;
        break;
    case LW_SE:
        aabb.vcMax = ZFXVector(xmax, ycen, zcen);
        aabb.vcMin = ZFXVector(xcen, ymin, zmin);
        break;
    default: break;
}
aabb.vcCenter = (aabb.vcMax + aabb.vcMin) / 2.0f;
m_pChild[ChildID] = new ZFXOctree();
m_pChild[ChildID]->SetBoundingBox(aabb);
m_pChild[ChildID]->m_Pos      = ChildID;
m_pChild[ChildID]->m_pParent = pParent;
} // InitChildObjects

```

Rekursives Erstellen des Baums

Und nun legen wir los. Erschreckenderweise sind die Methoden zum rekursiven Erstellen eines algorithmischen Baums immer verdammt kurz und einfach. Aber ich werde mir die größte Mühe geben, es so kompliziert wie möglich zu halten, damit ihr hinterher nicht so deprimiert seid, weil das alles so einfach ist. Die folgende Methode wird einmal für die Wurzel aufgerufen und ist der Kick Off für die Erstellung des Baumes. Die Applikation, die einen Octree nutzen möchte, gibt hier eine Liste von Polygonen für den Octree an. Der Root-Node übernimmt diese Liste, indem er sie sich kopiert, und delegiert die Arbeit des Erstellens dann einfach die Methode `ZFXOctree::CreateChilds`. Dort findet die eigentliche Arbeit statt. Diese Trennung ist notwendig, weil diese Methode dann auch von den weiteren Children einfach weiterverwendet werden kann.

Run Forrest, run!



```
void ZFXOctree::BuildTree(const ZFXPolygon *pPolys,
                          UINT Num) {
    m_pRoot = this;

    if (Num<1) return;

    // Berechne die AABB für die Wurzel
    CalcBoundingBox(pPolys, Num);

    m_pPolys = new ZFXPolygon[Num];
    m_NumPolys = Num;

    for (UINT i=0; i<Num; i++)
        m_pPolys[i].CopyOf( pPolys[i] );

    // Berechne die Kinder rekursiv
    CreateChilds(this);

    SAFE_DELETE_A(m_pPolys);
} // BuildTree
```

Rekursive Erstellung

Die Adresse des Root-Nodes schleifen wir einfach unverändert durch den Erstellungsprozess durch, so dass jeder Node sich den Pointer merken kann. Danach prüfen wir in der entsprechenden Member-Methode `ZFXOctree::CreateChilds` unser Abbruchkriterium für die Erstellung, also die Anzahl der Polygone, die sich in einem Node befinden. Müssen wir den Node weiter zerlegen, dann sind drei Schritte notwendig:

- ➔ Initialisiere die acht Child-Objekte.
- ➔ Beschneide die Polygonliste für die Child-Nodes.
- ➔ Erzeuge die Children der Child-Nodes rekursiv.

Der interessante Teil

Die Initialisierung der Children haben wir ja schon gesehen. Dort legen wir einfach nur deren Bounding-Box fest. Der dritte Schritt ist auch trivial, denn das ist einfach der rekursive Aufruf der Methode, um den Erstellungsprozess für jeden der acht Child-Nodes fortzusetzen. Der interessante Teil ist nun das Beschneiden der am Node vorliegenden Liste (ausgehend von der gesamten Liste am Root-Node) auf das jeweilige Child. Wir nehmen also die gesamte Geometrie des aktuellen Nodes und laufen durch seine acht Children. Jedem Child-Node geben wir diese Geometrie an, und die Nodes müssen diese Geometrie jeweils so beschneiden, dass die Geometrie nicht aus ihrer eigenen Bounding-Box herausragt. So ist sichergestellt, dass ein Child-Node exakt die Geometrie enthält, die er mit seiner Bounding-Box umschließt. Hier ist die Methode, die die rekursive Erstellung des Baums durchführt.



```
void ZFXOctree::CreateChilds(ZFXOctree *pRoot) {
    // speichere Adresse der Wurzel
    m_pRoot = pRoot;

    // weiter zerlegen?
    if ((pRoot == this) || (m_NumPolys > POLYS_PER_LEAF))
    {
        // initialisiere Children
        for (int i=0; i<8; i++) {
            InitChildObject( i, this );

            // erstelle Polygonliste für die Children
            m_pChild[i]->ChopListToMe(m_pPolys,
                                     m_NumPolys);
            m_pChild[i]->CreateChilds(pRoot);
        }

        // Polygonliste innerer Nodes freigeben
        SAFE_DELETE_A(m_pPolys);
    }
    // nein, das ist ein Leaf
    else return;
} // CreateChilds
```

Auch hier verstecken wir die Arbeit wieder in einer Unterfunktion, nämlich in `ZFXOctree::ChopListToMe`. Mit dieser Methode schneidet sich ein Child-Node aus der Liste von Polygonen seines Parents exakt diejenige Geometrie heraus, die er umschließt. Und dabei sage ich bewusst nicht »diejenigen **Polygone**«, sondern die »**Geometrie**«. Aber dazu gleich mehr. Es sei noch erwähnt, dass wir an jedem Node die Liste von Polygonen wieder löschen, nachdem die Children erstellt sind – es sei denn, es handelt sich um ein Leaf. Leaves dürfen und müssen die Liste von Polygonen behalten, sonst hätten wir am Ende nur eine Bounding-Box-Struktur für die Input-Geometrie erschaffen und könnten lediglich auf der Ebene von Bounding-Boxen bis zu einem gewissen Detailgrad Kollisionsabfragen u.Ä. durchführen.

*Schon wieder
vertröstet*

Polygonliste auf einen Node beschneiden

Wenn ein Node sich aus einer Liste von Polygonen diejenige Geometrie herausuchen soll, die er umschließt, so gibt es zwei triviale Fälle. Erstens: Wenn ein Polygon vollkommen außerhalb der Bounding-Box des Nodes liegt, dann wird es einfach ignoriert. Der zweite Fall ist genau umgekehrt: Wenn ein Polygon vollkommen innerhalb der Bounding-Box liegt, dann kopiert der Nodes sich das in seine eigene Liste von Polygonen. Interessant wird es dort, wo wir ein Polygon haben, das teilweise innerhalb und teilweise außerhalb der Bounding-Box liegt, sie also schneidet. Hier gibt es verschiedene Strategien von Octrees. Man kann ein Polygon mit einem Flag versehen und dann komplett in jedem Node speichern, den es schneidet.

*Zuschneiden
einzelner Polgone*

Wenn man Abfragen mit diesem Polygon durchführt, muss man nur darauf achten, dass man es nicht mehrmals in verschiedenen Nodes anspricht. Das ist aber nur interessant, wenn man über einen Octree rendert oder wenn man andere Objekte wie beispielsweise Characters in einem Octree speichert. Wir folgen hier der Strategie des exakten Octrees und beschneiden das Polygon kurzerhand so, dass der verbleibende Teil genau in die Bounding-Box passt und kein Stückchen mehr übersteht.

Datenberg

So erzeugen wir viel mehr Polygone, als ursprünglich in der Geometrie vorhanden waren, insbesondere wenn wir den Octree bis auf die Ebene einer sehr kleinen Menge an Polygonen pro Leaf erstellen. Aber dafür umgehen wir unschöne Kontrolleffekte, wenn wir ein Polygon mehrfach in den Octree einsortieren. Dieser Weg ist für unseren Kollisions-Octree also der beste. Nun lassen wir noch schnell einen Jubelschrei über unsere Mathematik-Bibliothek los, die uns die Implementierung dieser wuchtigen Aufgabe vollkommen abnimmt, und implementieren die Methode wie folgt:



```
void ZFXOctree::ChopListToMe(ZFXPolygon *pList,
                             UINT Num)
{
    ZFXPolygon ChoppedPoly;
    int nClass=0;

    if (Num < 1) return;

    // Falls hier schon was ist ...
    SAFE_DELETE_A(m_pPolys);
    m_NumPolys = 0;

    ZFXPolygon *TempMem = new ZFXPolygon[Num];

    // laufe durch die Liste
    for (UINT i=0; i<Num; i++) {
        if (pList[i].GetFlag() == 1) continue;

        ChoppedPoly.CopyOf( pList[i] );

        nClass = ChoppedPoly.Cull(m_Aabb);

        // Polygon total außerhalb der AABB
        if (nClass == ZFXCULLED) continue;

        // Polygon in der AABB oder schneidet AABB
        else {
            // falls total enthalten, setze Flag
            if ( nClass != ZFXCLIPPED) pList[i].SetFlag(1);

            // sonst beschneide das Polygon für AABB
            else ChoppedPoly.Clip(m_Aabb);
        }
    }
}
```



```

        // füge es der temporären Liste hinzu
        TempMem[m_NumPolys].CopyOf(ChoppedPoly);
        m_NumPolys++;
    }

    // kopiere die temporäre Liste in diesen Node
    m_pPolys = new ZFXPolygon[m_NumPolys];
    for (UINT j=0; j<m_NumPolys; j++)
        m_pPolys[j].CopyOf( TempMem[j] );

    delete [] TempMem;
} // ChopListToAabb

```

Als Parameter erhält die Methode die Polygonliste aus dem Parent des Nodes, über den wir die Methode aufrufen. Die Methode geht dann durch die Liste und legt von jedem Polygon eine Kopie an. Dann wird die Culling-Methode für dieses Polygon mit der Bounding-Box des Nodes aufgerufen. Wenn sie angibt, dass dieses Polygon außerhalb der Box, also gecullt ist, dann untersucht die Methode das nächste Polygon. Ist das Ergebnis des Cullings jedoch, dass das Polygon vollständig in der Bounding-Box liegt, dann setzen wir das Flag des Polygons in der Originalliste auf 1 und fügen es einer temporären neuen Liste hinzu. Das Flag dient dazu, in den Brüder-Nodes dieses Nodes das Polygon gleich verwerfen zu können, weil es bereits vollständig in diesem Node hier enthalten war. Wenn das Culling jedoch das unerfreuliche Ergebnis `ZFXCLIPPED` liefert, also meldet, dass das Polygon von der Bounding-Box geschnitten wird, dann rufen wir die `ZFXPolygon::Clip`-Methode unserer Mathe-LIB auf. Diese beschneidet das Polygon genau so, dass es korrekt in die Bounding-Box passt und alle überstehenden Ecken und Kanten abgeschnitten werden. Das beschnittene Polygon kommt dann in die temporäre Liste.

Und so funktioniert's

Am Ende der Funktion kopieren wir die temporäre Liste in das Attribut des Nodes und sind damit fertig. Nun enthält dieser Node genau den Ausschnitt der ursprünglichen Geometrie, der in seiner Bounding-Box liegt. Damit ist unser Octree tatsächlich schon komplett. Das Kopieren der Liste gestaltet sich allerdings etwas schwierig: Die Klasse `ZFXPolygon` hat ja selbst dynamisch Speicher allokiert, und wenn wir nun einfach per `memcpy()` die Daten von der Adresse `TempMem` nach `m_pPolys` kopieren würden, so wäre das eigentlich auch in Ordnung, denn die Pointer auf den dynamischen Speicher würden ja korrekt mitkopiert. Aber eben nur die Pointer. Wenn wir den Speicher für `TempMem` dann per `delete` freigeben, wird aber der Destruktor der Klasse `ZFXPolygon` für jede Instanz der Liste aufgerufen und der dynamische Speicher freigegeben. Die Pointer auf diesen Speicher in den Instanzen in `m_pPolys` würden dann beim ersten Zugriff einen Fehler auslösen. Daher müssen wir die Polygone in diesem Fall einzeln kopieren.

Schon fertig

Kollisionsabfragen im Octree

Kollidierende Kisten

Da wir unseren Octree nur für Kollisionsabfragen einsetzen möchten, benötigen wir dort natürlich auch entsprechende Methoden, um diese Abfragen durchzuführen. Als Erstes implementieren wir daher die Kollisionsabfrage zwischen einer achsen-orientierten Bounding-Box und der Geometrie, die in dem Octree steckt. Dazu schieben wir die Box oben in den Octree ein. In dem Octree prüfen wir dann an jedem Node erst einmal, ob die Box überhaupt die Box des Nodes schneidet. Wenn sie das nicht tut, dann können wir die weitere Suche nach einer Kollision in diesem Ast sofort abbrechen, denn alle acht Kinder dieses Nodes sind ja ebenfalls nur innerhalb des Raumes, den der aktuelle Node auch einnimmt. Ist aber an diesem Node eine Kollision aufgetreten, dann müssen wir die Bounding-Box auf eine Kollision mit den acht Kindern dieses Nodes testen, indem wir dieselbe Methode rekursiv aufrufen.

Der erste gewinnt

Wenn die Box sich dann irgendwann einmal zu einem Leaf durchgearbeitet hat, testen wir alle dort wartenden Polygone auf eine Kollision mit dieser Box. Treffen wir im Verlauf dieser Tests auf eine Kollision, so geben wir die erstbeste Kollision zurück, die wir finden. Das bedeutet: Wir schießen gleich den Rückgabewert `true` raus und beenden das Testen damit. Gleichzeitig geben wir auch noch die Ebene des kollidierenden Polygons mit aus, falls der Aufrufer diese Information zur Weiterverarbeitung in Reaktion auf die Kollision benötigt. Diese Methode prüft also nur, ob überhaupt eine Kollision stattgefunden hat. Sie kann in der jetzigen Form nicht dazu eingesetzt werden, den ersten Kollisionspunkt der Box mit der Geometrie zu finden. Dazu müsste man alle Nodes des Octrees hundertprozentig durchtesten.



```
bool ZFXOctree::TestCollision(const ZFXAabb &aabb,
                             ZFXPlane *pP) {
    // Teste Kollision mit diesem Node
    if (this != m_pRoot) {
        if (!m_Aabb.Intersects(aabb))
            return false;
    }

    // Keine Geometrie, nur ein Node
    if ( !IsLeaf() ) {
        for (int i=0; i<8; i++) {
            // Suche irgendeine Kollision
            if ( m_pChild[i]->TestCollision(aabb, pP) )
                return true;
        } // for

        // Keines der Kinder ist kollidiert
        return false;
    } // if [!leaf]
```

```

// Das ist ein Leaf mit Geometrie
else {
    if (!m_pPolys) return false;

    // Teste alle Polygone auf Kollision
    for (UINT i=0; i<m_NumPolys; i++) {
        if (m_pPolys[i].GetAabb().Intersects(aabb)) {
            if (pP) *pP = m_pPolys[i].GetPlane();
            return true;
        }
    }
    // Keine Kollision in diesem Leaf
    return false;
}
} // TestCollision [aabb]

```

Die zweite Kollisionsabfrage, die wir neben der Kollision einer Bounding-Box implementieren, ist die zwischen der Geometrie in dem Octree und einem Strahl. Diese Abfrage kann man später dazu verwenden, in einem Level freie Sichtlinien zu berechnen, indem man Strahlen von dem Objekt, für das die Sichtlinie geprüft werden soll, in Richtung der Objekte schießt, die man auf Sichtbarkeit hin testen will. Die Implementierung dieser Methode entspricht fast der mit den Bounding-Boxen, weist aber einen feinen, kleinen Unterschied auf: Wenn wir mit dem Strahl bis zu einem Leaf vorgestoßen sind, dann suchen wir dort innerhalb des Leafs den ersten Kollisionspunkt heraus und nicht einfach einen beliebigen. Wir nehmen zwar das erstbeste Leaf, in dem eine Kollision stattfindet, aber innerhalb dieses Leafs suchen wir schon den Punkt der ersten Kollision auf dem Strahl.

**Strahlen-
beschuss**

```

bool ZFXOctree::TestCollision(const ZFXRay &Ray,
                             float fL, float *pfD)
{
    bool bInCollision=false;
    float _fD=0.0f;

    // Teste Kollision mit diesem Node
    if (this != m_pRoot) {
        if ( !m_Aabb.Intersects(Ray, fL, pfD) &&
            !m_Aabb.Contains(Ray, fL) )
            return false;
    }

    // Keine Geometrie, nur ein Node
    if ( !IsLeaf() ) {
        for (int i=0; i<8; i++) {
            // Suche irgendeine Kollision in einem Child
            if ( m_pChild[i]->TestCollision(Ray, fL, pfD) )
                return true;
        } // for
    }
}

```



```

        // Keines der Kinder ist kollidiert
        return false;
    } // if [!leaf]

    // Das ist ein Leaf mit Geometrie
    else {
        if (!m_pPolys) return false;

        // Teste alle Polygone auf Kollision
        for (UINT i=0; i<m_NumPolys; i++) {
            if (m_pPolys[i].Intersects(Ray,false,fL,&_fD))
            {
                blnCollision = true;
                if (!pfD) return true;

                // Finde den dichtesten Kollisionspunkt
                if ( (*pfD<=0.0f) || (_fD < *pfD))
                    *pfD = _fD;
            }
        }
        // Keine Kollision in diesem Leaf
        return blnCollision;
    }
    return false;
} // TestCollision [ray]

```

Höhe des Spielers im Octree

Auch nur eine Kollision

Eine besondere Art der Kollisionsabfrage ist das Evaluieren des nächstgelegenen Polygons aus dem Octree, ausgehend von einer gegebenen Position. Diese Position kann beispielsweise die eines Spielers in einem Level sein. Wenn wir wissen wollen, auf welcher Höhe der Spieler sich aktuell befindet, dann müssen wir ausgehend von seinem Kopf einen Strahl nach unten in den Octree schießen. Dadurch finden wir die Höhe des nächstgelegenen Polygons unter dem Spieler und können ihn entsprechend auf die korrekte Höhe setzen, wenn er beispielsweise eine Treppe oder eine Schräge hinauf oder hinunter geht. Das ist eine besondere Art der Kollisionsabfrage, weil wir hier genau wissen, dass der zu prüfende Strahl senkrecht nach unten verläuft.

Vertikaler Strahl in AABB?

Dadurch können wir uns einige Tests vereinfachen. Wir brauchen beispielsweise keinen vollwertigen Test »Strahl versus Bounding-Box eines Octree-Nodes« durchzuführen. Stattdessen schreiben wir eine Methode, die einen einfacheren Test für diesen Spezialfall durchführt. Wenn der Ausgangspunkt des Strahls auf der X- oder der Z-Achse schon außerhalb der Box liegt, dann schneidet er die Box mit Sicherheit nicht, weil er ja senkrecht nach unten läuft. Und wenn der Ausgangspunkt des Strahls bereits unter-

halb des minimalen Y-Wertes der Box liegt, dann kann er diese auch nicht mehr schneiden. Der dritte Fall, den wir prüfen müssen, ist, ob wir nicht bereits einen Kollisionspunkt gefunden haben, dessen Y-Wert höher liegt als der maximale Y-Wert der Bounding-Box. In diesem Fall könnte der Strahl die Box zwar schneiden, der Schnittpunkt liegt aber tiefer als ein bereits von uns gefundener und ist damit uninteressant, um die Höhe für ein Objekt in der Geometrie des Octrees zu bestimmen. Dazu benötigen wir den höchsten Y-Wert unter dem ursprünglichen Punkt, der beispielsweise der Kopf des Spielers ist.

Treffen alle diese Fälle nicht zu, dann schneidet der Strahl die Box. Die folgende Methode übernimmt den Ausgangspunkt für den Strahl und den Wert für die bisher gefundene Entfernung vom Ursprung des Strahls zu einem Kollisionspunkt im Octree, also die bisherige Höhe über Grund des angegebenen Punktes:

```
bool ZFXOctree::IntersectsDownwardsRay(const ZFXVector
                                       &vcOrig, float f)
{
    // Strahl startet erst unterhalb dieses Nodes
    if (vcOrig.y < m_Aabb.vcMin.y) return false;

    // auf der X-Achse außerhalb dieses Nodes
    if (vcOrig.x < m_Aabb.vcMin.x) return false;
    if (vcOrig.x > m_Aabb.vcMax.x) return false;

    // auf der Z-Achse außerhalb dieses Nodes
    if (vcOrig.z < m_Aabb.vcMin.z) return false;
    if (vcOrig.z > m_Aabb.vcMax.z) return false;

    // minimal mögliche Entfernung zu einem Polygon
    // bereits größer als bisher gefundene "f"
    if (f < (fabs(m_Aabb.vcMax.y - vcOrig.y)))
        return false;

    return true;
} // IntersectsDownwardsRay
```



Mit dieser Hilfsmethode können wir nun auch die Methode schreiben, die eine Applikation von außen über ihren Octree aufrufen kann, um die Höhe über Grund für eine Position im Raum zu bestimmen. Die nun folgende Methode namens `ZFXOctree::GetFloor` ist sozusagen eine etwas aufgepeppte Variante von `ZFXOctree::TestCollision` für einen Strahl. Diese Methode ist zwar für unsere Zwecke hinreichend genau, aber eben nicht hundertprozentig genau, weil sie gleich nach der ersten Kollision in einem Leaf den Erfolg zurückmeldet und nicht nach dem am nächsten liegenden Kollisionspunkt auch in anderen Leaves sucht. Die Methode zum Finden der korrekten Höhe muss dies aber tun. Ihr Grundaufbau ist also derselbe wie der für die

*Höhe im Octree
finden*

Kollision mit einem beliebigen Strahl. Es gibt nur die zwei Unterschiede, dass wir einerseits Vereinfachungen vornehmen können, weil der Strahl in diesem Fall garantiert senkrecht ist, und andererseits nicht nach der ersten gefunden Kollision in einem Child zurückkehren können. Stattdessen müssen wir wirklich alle Children durchlaufen und bei mehreren gefundenen Kollisionenpunkten den mit der kleinsten Distanz vom Ursprungspunkt des Strahls auswählen.



```
bool ZFXOtree::GetFloor(const ZFXVector &vcPos,
                        float *pf, ZFXPlane *pPlane)
{
    float  fAabbDist=0, fHitDist=0;
    bool   bHit=false;
    ZFXAabb aabb;
    ZFXRay  Ray;

    // falls das die Wurzel ist
    if ( this == m_pRoot) *pf = 99999.0f;

    // keine Geometrie, das ist nur ein Node
    if ( !IsLeaf() ) {
        for (int i=0; i<8; i++) {
            // Schneidet der Strahl überhaupt eines der Children?
            if ( m_pChild[i]->IntersectsDownwardsRay(
                                    vcPos, *pf) )
            {
                // Schnitt dichter als bisherige Distanz?
                if (m_pChild[i]->GetFloor(vcPos, pf,
                                           pPlane))
                    bHit = true;
            }
        } // for
        return bHit;
    } // if [!leaf]

    // sonst ist das ein Leaf mit Geometrie
    else {
        if (!m_pPolys) return false;

        Ray.Set(vcPos, ZFXVector(0.0f,-1.0f,0.0f));

        // laufe durch alle Polys
        for (UINT i=0; i<m_NumPolys; i++) {
            aabb = m_pPolys[i].GetAabb();

            // Schnelltest Strahl neben dem Poly
            if ((Ray.m_vcOrig.x < aabb.vcMin.x) ||
                (Ray.m_vcOrig.x > aabb.vcMax.x) ||
                (Ray.m_vcOrig.z < aabb.vcMin.z) ||
```

```

        (Ray.m_vcOrig.z > aabb.vcMax.z) ||
        (Ray.m_vcOrig.y < aabb.vcMin.y) )
    continue;

    // Volltest Strahl versus Polygon
    if (m_pPolys[i].Intersects(Ray, true, *pf,
                               &fHitDist))
    {
        *pf = fHitDist;
        bHit = true;
    }
} // for
return bHit;
}
} // GetFloor

```

Den eigentlichen Strahl müssen wir dank unserer Hilfsmethode für den Schnelltest des senkrechten Strahls mit einer Bounding-Box erst in einem Leaf erstellen. Und auch die Bounding-Box eines Polygons im Leaf können wir erst einmal entsprechend durch einen Mini-Schnelltest abhandeln, bevor wir den richtigen Test aus der Tasche ziehen. Der ist zwar schnell, aber der Schnelltest ist noch schneller. Daher der Name. :-)

Schnelltests

Nun ist unser Octree absolut einsatzbereit und ist geradezu dafür prädestiniert, für Kollisionsabfragen in einem Indoor-Level eingesetzt zu werden.

Durchlaufen des Baums

War das Durchlaufen des Baums zum Einsammeln der sichtbaren Polygone im BSP-Baum noch wenigstens deswegen etwas spannend, weil man die Front-to-Back- und Back-to-Front-Sortierung beachten musste, so ist die Traversal des Octrees so einfach, wie sie nur sein kann. Wenn ein Node nicht im angegebenen View-Frustum liegt, wird der rekursive Ast an dieser Stelle abgebrochen. Anderenfalls gehen wir weiter in alle acht Children des Octrees und führen dort jeweils einen weiteren rekursiven Ast aus. Schaffen wir es irgendwann in ein Leaf, dann fügen wir alle dort gelagerten Polygone einfach der angegebenen Liste hinzu.

*Node für Node für
Node für Node für
...*

Bei dem Octree ist es aber auch noch interessant, den Tree an sich zu visualisieren. Das kann man bei dem BSP-Baum nicht ganz so gut machen. Für den Octree kann man aber die Bounding-Boxen ebenfalls rendern, weil diese die Zerlegung des Raums gut sichtbar machen. Je mehr und je kleinere Boxen man an einer Stelle des Raums vorfindet, umso tiefer ist der Octree dort logischerweise aufgebaut. Unserer Methode für das Einsammeln der Polygone geben wir also eine zusätzliche Instanz der Klasse `ZFXPolyListe` an, in der wir für jedes sichtbare Leaf die Polygone seiner Bounding-Box speichern.

Eye Candy



```
void ZFXOctree::Traverse(ZFXPolylist *pList,
                        ZFXPolylist *pAabbList,
                        const ZFXPlane *pFrustum)
{
    if (m_Aabb.Cull(pFrustum, 6) == ZFXCULLED) return;

    if ( IsLeaf() ) {

        if (pList) {
            for (unsigned int i=0; i<m_NumPolys; i++)
                pList->AddPolygon(m_pPolys[i]);
        }
        if (pAabbList) GetAabbAsPolygons(pAabbList);
    }
    else {
        m_pChild[0]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[1]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[2]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[3]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[4]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[5]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[6]->Traverse(pList, pAabbList, pFrustum);
        m_pChild[7]->Traverse(pList, pAabbList, pFrustum);
    }
} // Traverse
```

AABB als Box von Polygonen

Die Konstruktion eines Würfels aus einer achsen-ausgerichteten Bounding-Box ist nur ein wenig Fleißarbeit. Wir kennen den Mittelpunkt der Box, und wir kennen ihre Ausdehnung auf allen drei ebenfalls bekannten Achsen. Daraus basteln wir nun die acht Positionen im Raum, die die Eckpunkte der Box ausmachen, und konstruieren sechs Rechtecke für die Box.



```
void ZFXOctree::GetAabbAsPolygons(ZFXPolylist *pList) {
    ZFXPolygon    Poly;
    ZFXVector     vcPoints[24];
    unsigned int  nIndis[6] = { 0, 1, 2, 2, 3, 0 };

    float fW = m_Aabb.vcMax.x - m_Aabb.vcMin.x;
    float fH = m_Aabb.vcMax.y - m_Aabb.vcMin.y;
    float fD = m_Aabb.vcMax.z - m_Aabb.vcMin.z;

    // oberes Rechteck
    vcPoints[0].Set( m_Aabb.vcCenter.x - (fW / 2.0f),
                    m_Aabb.vcCenter.y + (fH / 2.0f),
                    m_Aabb.vcCenter.z - (fD / 2.0f) );
    vcPoints[1].Set( m_Aabb.vcCenter.x - (fW / 2.0f),
                    m_Aabb.vcCenter.y + (fH / 2.0f),
                    m_Aabb.vcCenter.z + (fD / 2.0f) );
    vcPoints[2].Set( m_Aabb.vcCenter.x + (fW / 2.0f),
```



```

        m_Aabb.vcCenter.y + (fH / 2.0f),
        m_Aabb.vcCenter.z + (fD / 2.0f) );
vcPoints[3].Set( m_Aabb.vcCenter.x + (fW / 2.0f),
        m_Aabb.vcCenter.y + (fH / 2.0f),
        m_Aabb.vcCenter.z - (fD / 2.0f) );
Poly.Set( &vcPoints[0], 4, nIndis, 6 );
pList->AddPolygon( Poly );

// rechtes Rechteck
vcPoints[4] = vcPoints[3];
vcPoints[5] = vcPoints[2];
vcPoints[6].Set( m_Aabb.vcCenter.x + (fW / 2.0f),
        m_Aabb.vcCenter.y - (fH / 2.0f),
        m_Aabb.vcCenter.z + (fD / 2.0f) );
vcPoints[7].Set( m_Aabb.vcCenter.x + (fW / 2.0f),
        m_Aabb.vcCenter.y - (fH / 2.0f),
        m_Aabb.vcCenter.z - (fD / 2.0f) );
Poly.Set( &vcPoints[4], 4, nIndis, 6 );
pList->AddPolygon( Poly );

// linkes Rechteck
vcPoints[8] = vcPoints[0];
vcPoints[9] = vcPoints[1];
vcPoints[10].Set( m_Aabb.vcCenter.x - (fW / 2.0f),
        m_Aabb.vcCenter.y - (fH / 2.0f),
        m_Aabb.vcCenter.z + (fD / 2.0f) );
vcPoints[11].Set( m_Aabb.vcCenter.x - (fW / 2.0f),
        m_Aabb.vcCenter.y - (fH / 2.0f),
        m_Aabb.vcCenter.z - (fD / 2.0f) );
Poly.Set( &vcPoints[8], 4, nIndis, 6 );
pList->AddPolygon( Poly );

// hinteres Rechteck
vcPoints[12] = vcPoints[2];
vcPoints[13] = vcPoints[1];
vcPoints[14] = vcPoints[10];
vcPoints[15] = vcPoints[6];
Poly.Set( &vcPoints[12], 4, nIndis, 6 );
pList->AddPolygon( Poly );

// vorderes Rechteck
vcPoints[16] = vcPoints[0];
vcPoints[17] = vcPoints[3];
vcPoints[18] = vcPoints[7];
vcPoints[19] = vcPoints[11];
Poly.Set( &vcPoints[16], 4, nIndis, 6 );
pList->AddPolygon( Poly );

// unteres Rechteck
vcPoints[20] = vcPoints[7];

```

```

vcPoints[21] = vcPoints[6];
vcPoints[22] = vcPoints[10];
vcPoints[23] = vcPoints[11];
Poly.Set( &vcPoints[20], 4, nIndis, 6 );
pList->AddPolygon( Poly );
} // GetAabbAsPolygons

```

*Nicht schön, aber
glücklicherweise
selten*

Was wir allein in dieser Methode an versteckten dynamischen Allokationen von Speicher haben, geht kaum auf eine Kuhhaut – erst recht nicht, wenn man bedenkt, dass wir das für jedes sichtbare Leaf durchführen müssen. Aber diese Methode erfüllt wie die Visualisierung des Octrees für unsere kleine Demo ihren Zweck. Sie ist für nichts anderes gedacht, und ihr solltet sie nach dem Durcharbeiten dieses Kapitels gleich aus der Klasse löschen. Nachdem wir das geklärt haben, schreiben wir unsere kleine Demo-Applikation, damit wir auch etwas zum Vorzeigen haben.

13.5 Demo-Applikation BSP-Tree und Octree

*Was wollen wir
demonstrieren?*

Nun kommen wir endlich zu der lange versprochenen Demo-Applikation. Dazu sollten wir zunächst klären, was wir mit dieser Applikation demonstrieren wollen. Wir haben nun in unserer Mathe-Lib die Klassen `ZFXBspTree` und `ZFXOctree`. Beide können wir dazu nutzen, eine beliebig aufgebaute Menge an Polygonen, beispielsweise Level-Daten, zu verwenden, um einen entsprechenden Baum zu konstruieren. Diesen Baum wiederum können wir für Kollisionsabfragen verwenden. Wir werden also im Folgenden eine kleine Demo schreiben, die Level-Daten in Form einer Liste von Polygonen aus einer Datei einliest und am Bildschirm anzeigt. Der Betrachter kann in diesem Level dann mit einer freien Kamera manövrieren, und die Wände des Levels blockieren durch die Kollisionsabfrage den Weg des Betrachters. Das Rendern der Geometrie erledigen wir auch durch die `Traverse ... ()`-Methoden der jeweiligen Klasse, indem wir die dort zusammengetragene Liste an Polygonen in jedem Frame zu Vertices umwandeln und rendern. Wie ihr an der Frame-Rate sehen könnt, ist dies ein Weg, den man keinesfalls gehen sollte. Aber für diese Demo wird das ausreichend sein. Am Ende dieses Buches werden wir noch ein umfangreiches Spiel entwickeln, in dem ihr dann sehen könnt, wie man das richtig macht.

*BSP und Octree in
einer Demo*

Die Applikation wird für den geladenen Level auch gleich je einen BSP-Baum und einen Octree erstellen. Zur Laufzeit der Demo kann man dann zwischen drei verschiedenen Varianten umschalten. Bei den ersten beiden Varianten erledigen wir die Kollisionsabfrage über den BSP-Baum, und das Rendern erfolgt einmal über den BSP-Baum in Front-to-Back-Sortierung und das andere Mal über den BSP-Baum in Back-to-Front-Sortierung. Bei letzterer Variante schalten wir auch den Depth-Buffer aus, um zu demonstrieren, dass es hier zu keinem Fehler durch fehlerhaften Overdraw kommt. Die dritte Variante verwendet schließlich den Octree für das Rendern und

die Kollisionsabfrage. Zusätzlich gibt es noch zwei verschiedene Render-Modi für jede Variante, nämlich einmal den soliden Level und einmal den Level als Wireframe. Im Wireframe-Modus werden bei dem Octree auch zusätzlich die Bounding-Boxen der Octree-Leaves gerendert, und man kann so sehr schön den Octree erkennen.

Nun haben wir noch ein Problem: Die Polygone in den Bäumen enthalten ja lediglich die Positionen der Vertices. Sie enthalten aber eindeutig nicht die Vertex-Normalen, die Texturkoordinaten und eine Information über die verwendete Textur. Wenn wir nun die Polygone rendern wollen, so sähe der Level sehr unschön aus, weil alles dieselbe Farbe hätte und man keine Struktur erkennen könnte. Wir verwenden daher im gesamten Level dieselbe Textur für alle Polygone. Doch wir brauchen auch Texturkoordinaten, da diese ansonsten als (0,0) an jedem Vertex angenommen würden und das gesamte Polygon wieder überall dieselbe Farbe hätte. Wenn man nun einen Zufalls-wert für die Texturkoordinaten zwischen 0.0f und 1.0f berechnet, so ist dies eine gute Lösung. Dann haben wir jedoch in jedem Frame andere Texturkoordinaten, und das sieht natürlich ziemlich wüst und flackerig aus. Wir müssen also für die Vertices einen Weg finden, wie wir in jedem Frame dieselben Texturkoordinaten berechnen können, die die Textur wenigstens halbwegs über das Polygon ziehen, so dass das Polygon nicht überall dieselbe Farbe hat.

*Texturierung des
Levels*

Rendern von ZFXPolygon-Instanzen

Beginnen wir doch gleich damit, dann haben wir das hinter uns. Den vollständigen Code der Demo-Applikation findet ihr natürlich auf der CD-ROM zu diesem Buch. Wir haben inzwischen so viele Demos gesehen, dass ich mir den Teil der Initialisierung unserer Engine, die WinMain()-Funktion usw. hier spare. Wir spendieren unserer Applikation eine Methode, mit der wir ein Objekt der Klasse ZFXPolylist über unser Render-Device rendern können. Ein bool-Parameter gibt dabei an, ob wir im Wireframe-Modus rendern wollen, und ein weiterer Parameter legt die Farbe für den Wireframe-Modus fest.

*Polygone zu
Vertices machen*

```
void RenderPolylist(ZFXPolylist &List, bool bWired,
                   const ZFXCOLOR *pClr) {
    ZFXPolygon *pCP = NULL;
    UINT        i=0, j=0;
    ZFXAabb      aabb;
    float        fMaxX=0.0f, fMaxY=0.0;
    ZFXPolygon *pPolys = List.GetPolylist();
    UINT         Num = List.GetNum();
    VERTEX       Verts[256];
    WORD         Indis[256];

    for (UINT p=0; p<Num; p++) {
```



```

pCP = &pPolys[p];
aabb = pCP->GetAabb();
fMaxX = aabb.vcMax.x - aabb.vcMin.x;
fMaxY = aabb.vcMax.y - aabb.vcMin.y;

for (i=0; i<pCP->GetNumPoints(); i++) {
    memset(&Verts[i], 0, sizeof(VERTEX));
    Verts[i].x = pCP->m_pPoints[i].x;
    Verts[i].y = pCP->m_pPoints[i].y;
    Verts[i].z = pCP->m_pPoints[i].z;

    // TEXTURKOORDINATEN BERECHNEN
    Verts[i].tu = Verts[i].x - aabb.vcMin.x / fMaxX;
    Verts[i].tv = Verts[i].y - aabb.vcMin.y / fMaxY;
}

if (!bWired) {
    g_pDevice->SetBackfaceCulling(RS_CULL_CCW);
    g_pDevice->SetShadeMode(RS_SHADE_SOLID,0,NULL);

    for (j=0; j<pCP->GetNumIndis(); j++)
        Indis[j] = pCP->m_pIndis[j];
}
else {
    g_pDevice->SetBackfaceCulling(RS_CULL_NONE);

    g_pDevice->SetShadeMode(RS_SHADE_LINES,0,pClr);
    for (j=0; j<pCP->GetNumPoints(); j++) {
        Indis[(j*2)] = j;
        Indis[(j*2)+1] = j+1;

        if (j==(pCP->GetNumPoints()-1))
            Indis[(j*2)+1] = 0;
    }
    j = 2 * pCP->GetNumPoints();
}
g_pDevice->GetVertexManager()->Render(VID_UU, i,
                                     j, Verts, Indis, 0);
} // for
} // RenderPolylist

```

*Und so
funktioniert's*

Wir kopieren zuerst die Positionen der Punkte des Polygons in Vertices. Danach berechnen wir Texturkoordinaten für die Vertices, wobei wir eine sehr vereinfachte Form der planaren Projektion auf die XY-Ebene verwenden. Bei der Implementierung des 3D-Level-Editors in einem späteren Kapitel werden wir natürlich eine korrekte Projektion durchführen, aber die Variante hier ist ausreichend. Wenn sich ein Polygon jedoch entlang der XY-Achse erstreckt, dann wird die Textur auf diesem Polygon entsprechend mit der ersten Reihe von Pixeln aus der Textur entlang der Z-Achse überzo-

gen. Das sieht nicht sonderlich schön aus, aber es reicht hier aus, um die Form der Geometrie zu erkennen. Wenn auch das erledigt ist, dann erzeugen wir die Indices zum Rendern dieses Polygons. Im Wireframe-Modus konstruieren wir diese so, dass wir das Polygon als geschlossenen Linestrip rendern können.

Laden der Level-Daten

Die interessante Frage bei solchen Beispielen ist ja immer, woher man eigentlich die Testdaten bekommt. Lustigerweise ist es einfacher, einen BSP-Baum oder einen Octree zu implementieren. Aber woher bekommt man dann einen Level in einem Rohformat, den man verwursten kann? Die Antwort lautet ganz einfach: *aus dem Off*. Ich stelle hier für die Demo eine Datei namens `Testlevel.pbp` zur Verfügung. Doch so einfach drücke ich mich nicht vor der Verantwortung. Die Dateieindung `*.pbp` steht für das Dateiformat, das unser eigener Editor für PreFabs verwendet. Da stellen sich gleich zwei neue Fragen: Welcher Editor, und was sind PreFabs?

Level-Format

In einem späteren Kapitel implementieren wir unseren eigenen Low-Polygon-Level-Editor mit grafischem Interface, um vollwertig eigene Level und Modelle erstellen zu können. Dieser wird von Haus aus zwei Formate beherrschen: Zum einen kann er echte Level-Daten laden und speichern und zum anderen PreFabs. Während echte Level auch Portale, Spawn-Points, Lichtquellen usw. enthalten, bestehen PreFabs lediglich aus Informationen über die Geometrie und deren Texturen. Salopp gesagt, handelt es sich um ein normales Format für 3D-Modelle. So können wir später in unserem Editor kleine Teile des Levels als Modelle speichern, um sie in anderen Levels auch einbauen zu können. Hier werden wir diese PreFabs verwenden, um sie in den BSP-Baum oder den Octree zu schubsen.

PreFabs

Das Dateiformat ist dabei so aufgebaut, dass man es sehr einfach laden kann. Als Erstes steht in der Binärdatei die Anzahl der Polygone. Danach folgen diese Polygone eines nach dem anderen in der Datei. Vor jedem Polygon steht allerdings noch ein Zähler, gefolgt von einem String **ohne terminierendes \0-Zeichen**. Der Zähler gibt die Länge des Strings an, und der String enthält den Namen der Grafikdatei, die als Textur für das folgende Polygon verwendet wird. Danach folgen die Daten für das Polygon. Die Textur werden wir hier nicht berücksichtigen, da es uns nicht um die optische Schönheit geht. Die Funktion zum Laden eines solchen PreFabs sieht also wie folgt aus:

Dateiformat

```
bool LoadLevel(const char *chFile) {
    ZFXPolygon *pList=NULL, Poly;
    char        *buffer[2048];
    UINT        Num=0, n=0;
```



```

FILE *pFile = fopen(chFile, "rb");
if (!pFile) return false;

// lies die Anzahl der Polygone in der Datei
fread(&Num, sizeof(UINT), 1, pFile);
if (Num == 0) return false;

pList = new ZFXPolygon[Num];

// lade alle Polygone
for (UINT i=0; i<Num; i++) {

    // lies den Namen der Textur
    fread(&n, sizeof(UINT), 1, pFile);
    fread(buffer, n, 1, pFile);
    buffer[n] = '\0';

    // lade das Polygon aus der Datei
    LoadPolygon(pFile, &Poly);
    pList[i].CopyOf( Poly );
} // for

// erzeuge den BSP-Baum
g_pBSPTree = new ZFXBspTree;
g_pBSPTree->BuildTree(pList, Num);

// erzeuge den Octree
g_pOctree = new ZFXOctree;
g_pOctree->BuildTree(pList, Num);

fclose(pFile);
if (pList) { delete [] pList; }
return true;
} // LoadLevel

```

Erzeugen der Bäume

Nachdem alle Polygone in eine entsprechende Liste geladen worden sind, erzeugen wir aus dieser Liste einmal einen BSP-Baum und dann noch einen Octree. Die Demo-Applikation kann ja beide wahlweise verwenden. Ihr erkennt auch schon, dass die Instanzen der Klassen ZFXBspTree und ZFXOctree globale Variablen sind.

Dateiformat des Polygons

Doch noch haben wir nicht gesehen, wie man die Daten für die Polygone lädt. Diese Funktionalität versteckt sich in der eben gezeigten Funktion in dem Aufruf von LoadPolygon(). Ein einzelnes Polygon in der Datei hat dabei zunächst ein paar Informationen gespeichert, analog einer Art Header-Datei. Dort finden sich die folgenden Informationen in dieser Reihenfolge:

Header eines Polygons

- ➡ UINT – Anzahl der Vertices
- ➡ UINT – Anzahl der Indices

- ➔ UINT – hier nicht verwendet, später ID der verwendeten Skin
- ➔ ZFXAabb – hier nicht verwendet
- ➔ bool – hier nicht verwendet
- ➔ float[4] – hier nicht verwendet, später Texturrepeat und -offset

Danach folgen die Vertices vom Typ `VERTEX` und die Indices vom Typ `WORD` sowie eine weitere Indexliste. Diese zweite Indexliste ist dabei die Liste der Indices, die man für einen geschlossenen Linestrip für dieses Polygon verwenden muss. Mit diesem Wissen ist es natürlich ein Leichtes, so ein Polygon aus der Datei zu ziehen und in einer Instanz der Klasse `ZFXPolygon` zu speichern.

```
void LoadPolygon(FILE *pFile, ZFXPolygon *pPoly) {
    UINT    NumVerts=0,
           NumIndis=0,
           ID=0;
    bool    bln=false;
    ZFXAabb aabb;
    float    f[4] = { 0.0f,0.0f,0.0f,0.0f };

    // Zähler einlesen
    fread(&NumVerts, sizeof(UINT), 1, pFile);
    fread(&NumIndis, sizeof(UINT), 1, pFile);
    fread(&ID,      sizeof(UINT), 1, pFile);
    fread(&aabb,    sizeof(ZFXAabb), 1, pFile);
    fread(&bln,    sizeof(bool), 1, pFile);
    fread(f,       sizeof(float), 4, pFile);

    // Speicher bereitstellen
    VERTEX *pvVerts = new VERTEX[NumVerts];
    WORD   *pwIndis = new WORD[NumIndis];
    WORD   *pDummy  = new WORD[NumVerts*2];

    ZFXVector *pvcPoints = new ZFXVector[NumVerts];
    UINT      *pnIndis   = new UINT[NumIndis];

    if ( !pvVerts || !pwIndis ||
         !pvcPoints || !pnIndis ) return;

    // Daten einlesen
    fread(pvVerts, sizeof(VERTEX), NumVerts, pFile);
    fread(pwIndis, sizeof(WORD), NumIndis, pFile);
    fread(pDummy, sizeof(WORD), NumVerts*2, pFile);

    // Vertices zu Vektoren konvertieren
    for (UINT i=0; i<NumVerts; i++)
        pvcPoints[i].Set(pvVerts[i].x,
```



```

        pvVerts[i].y,
        pvVerts[i].z);

// WORD zu UINT konvertieren
for (UINT j=0; j<NumIndis; j++)
    pnIndis[j] = pwIndis[j];

// Daten für das Polygon festlegen
pPoly->Set( pvcPoints, NumVerts, pnIndis, NumIndis );

// aufräumen
delete pvVerts;
delete pwIndis;
delete pvcPoints;
delete pnIndis;
delete pDummy;
} // LoadPolygon

```

Level geladen Damit haben wir einen Level erfolgreich geladen. Wir interpretieren den Begriff Level hier ein wenig großspurig, denn natürlich haben wir nur einen Haufen von Polygonen, die zwar nicht ganz zufällig so angeordnet sind, dass sie einem kleinen Indoor-Level zum Verwechseln ähnlich sehen. Aber wir haben hier noch keinerlei Informationen über Lichtquellen, Spaw-Points usw., die zu einer *echten* Level-Datei einfach dazu gehören würden.

Berechnung eines Frames

Kameraführung Da wir nun die Daten für den Level in unseren beiden Bäumen redundant gespeichert haben, können wir uns daran machen, dem Programm ein wenig Interaktivität zu verleihen. Damit wir uns auf das Wesentliche konzentrieren können, verwenden wir für den Input des Users einfach die WinAPI-Methode und reagieren in der WinProc()-Prozedur auf die entsprechenden Nachrichten, woraufhin wir dann unsere eigene Instanz eines Movement-Controllers veranlassen, sich entsprechend zu bewegen.

Update eines Frames Und dann tun wir Folgendes: Der Movement-Controller bleibt ja so lange in seiner bisherigen Position, bis wir seine Update()-Methode aufrufen. Wir speichern zunächst die aktuelle Position des Spielers, dann rufen wir eben dieses Update() auf und speichern auch die neue Position. Dann prüfen wir den Weg des Spielers von der alten Position zu der neuen Position auf eine Kollision, indem wir einen Strahl zwischen den beiden Punkten erzeugen und diesen der TestCollision()-Methode des gerade aktiven Baums übergeben.

Welcher Baum ist aktiv? Die globale Variable g_bBsp steuert dabei, ob der BSP-Baum oder der Octree verwendet werden soll. Der Spieler kann mit der Taste **E** zwischen den drei oben erwähnten Varianten umschalten, also zwischen dem BSP-Baum

in Front-to-Back-Sortierung, dann dem BSP-Baum in Back-to-Front-Sortierung und schließlich dem Octree. Für die Unterscheidung der Sortierreihenfolge sorgt die globale Variable `g_bFtB`.

Wenn der entsprechende Baum nun eine Kollision meldet, verbieten wir dem Spieler einfach die Bewegung in diese Richtung. Wir setzen den Movement-Controller dazu einfach auf seine alte Position zurück, die er vor Ausführung der Bewegung gehabt hat. Wenn das erledigt ist, setzen wir die entsprechende Position und Ausrichtung des Spielers für das Render-Device fest und beginnen mit dem Rendern. Dazu durchlaufen wir den entsprechenden Baum und erhalten somit eine Liste von Polygonen im aktuellen View-Frustum. Diese Liste rendern wir dann einfach über die eingangs gezeigte Funktion `RenderPolylist()`. Und so sieht dann die Funktion zum Update eines Frames unserer Applikation aus:

Kollisionsreaktion

```
HRESULT ProgramTick(void) {
    ZFXPlane    Frustum[6];
    ZFXPolylist List, AabbList;
    ZFXMatrix    mWorld;
    static ZFXCOLOR clrR = {1.0f, 0.3f, 0.3f};
    static ZFXCOLOR clrG = {0.0f, 0.7f, 0.0f};

    mWorld.Identity();

    g_pTimer->Update();

    // aktuelle Position abfragen
    ZFXVector vcOld = g_pMCEgo->GetPos(), vcNew(0,0,0);

    // Movement-Controller updaten
    g_pMCEgo->Update( g_pTimer->GetElapsed() );

    // Movement-Controller zurücksetzen
    g_pMCEgo->SetSpeed(0.0f);
    g_pMCEgo->SetSlideSpeed(0.0f);
    g_pMCEgo->SetRotationSpeedX(0.0f);
    g_pMCEgo->SetRotationSpeedY(0.0f);

    // neue Position abfragen
    vcNew = g_pMCEgo->GetPos();
    ZFXRay Ray; Ray.Set(vcNew, g_pMCEgo->GetDir());

    // falls Kollision auftritt, zurück auf alte Position
    if ( g_bBsp ) {
        if (g_pBSPTree->TestCollision(Ray, 0.5f, 0, 0))
            g_pMCEgo->SetPos(vcOld);
    }
    else {
        if (g_pOctree->TestCollision(Ray, 0.5f, 0))
```



```

        g_pMCEgo->SetPos(vcOld);
    }

    // Position für Renderer festlegen
    g_pDevice->SetView3D(g_pMCEgo->GetRight(),
                        g_pMCEgo->GetUp(),
                        g_pMCEgo->GetDir(),
                        g_pMCEgo->GetPos());

    // aktuelles View-Frustum abfragen
    g_pDevice->GetFrustum( Frustum );

    // Polygonliste aus dem Baum holen
    if ( g_bBsp ) {
        if ( g_bFtB )
            g_pBSPTree->TraverseFtB( &List,
                                     g_pMCEgo->GetPos(), Frustum );
        else
            g_pBSPTree->TraverseBtF( &List,
                                     g_pMCEgo->GetPos(), Frustum );
    }
    else {
        g_pOctree->Traverse( &List, &AabbList, Frustum );
    }

    g_pDevice->BeginRendering(true,true,true);
    mWorld.Translate(0.0f, 0.0f, 0.0f);
    g_pDevice->SetWorldTransform(&mWorld);

    if (g_bBsp) RenderPolylist(List,g_bWired,&clrR);
    else {
        RenderPolylist( List, g_bWired, &clrR );
        if (g_bWired) RenderPolylist(AabbList,
                                     g_bWired, &clrG);
    }
    return g_pDevice->EndRendering();
} // Tick

```

**Screenshot der
Demo**

In Abbildung 13.20 ist ein Screenshot der Demo-Applikation zu sehen, die gerade mit dem BSP-Baum eine Szene rendert. Wenn man sein Augenmerk weg von der bescheidenen Texturprojektion auf die hier als Overlay dargestellte Wireframe-Szene richtet, dann wird man die durch den BSP-Baum erzwungenen Splits in der Geometrie deutlich erkennen, beispielsweise an der Decke direkt oben mittig im Bild.

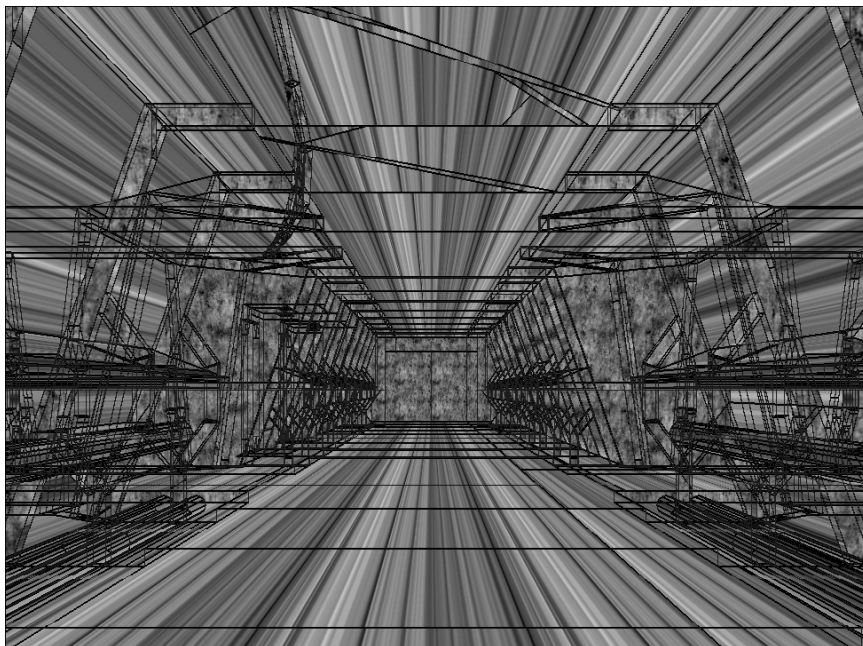


Abbildung 13.20: Screenshot der Demo-Applikation für den BSP-Baum. Die schwarzen Linien zeigen die Szene im Wireframe-Modus. Hier kann man gut einige der willkürlichen Schnitte erkennen, die die Zerlegung in einen BSP-Baum der Geometrie aufgezwungen hat.

Sehenswertes in der Demo

Okay, nun haben wir das Ding programmiert. Doch was können wir damit anfangen? Als Erstes kann man durch Drücken der Taste **[E]** die beiden BSP-Modi und danach den Octree Modus anwählen. Anhand der Texturen, die zwar ziemlich verschmiert aussehen, aber doch ausreichend sind, kann man schon einzelne Polygone ausmachen. Wenn man zwischen dem BSP und dem Octree umstellt, wird man an einigen Stellen Veränderungen an der Texturierung feststellen. Das kommt daher, dass der Octree die Geometrie ganz anders in Polygone zerlegt hat als der BSP-Baum. Dies ist eine interessante Beobachtung.

Was die Demo so tut

Die zweite Beobachtung, die man an dieser Demo machen kann, betrifft die Performance. Mal abgesehen von der insgesamt bescheidenen Frame-Rate im einstelligen Bereich schon bei kleinen Geometrien, fällt auch auf, dass der Octree nochmals um ein Vielfaches langsamer ist als der BSP-Baum. Man beachte dabei den Polygonzähler, der als Text ausgegeben wird. Der Octree stellt dieselbe Szene mit fast der dreifachen Menge an Polygonen dar wie der BSP-Baum. Das liegt daran, dass der Octree den Raum vollkommen willkürlich bzw. ohne Rücksicht auf die Geometrie entlang der Weltachsen aufteilt. Der BSP-Baum nimmt zunächst Rücksicht auf die Ebenen, an denen die Polygone ausgerichtet sind. Zum anderen können wir über die Heuristik zur Auswahl des Splitters sehr genau bestimmen, dass wir viele Splits von

Octree langsamer als BSP-Baum

Polygonen sehr hoch bestrafen und der BSP-Algorithmus gefälligst Splitter suchen soll, die zu möglichst wenig Splits führen.

Man beachte hierbei aber, dass die schlechte Performance hauptsächlich durch das unoptimierte *Stupid Force*-Rendern kommt. Die Bezeichnung »Brute Force« wäre für diesen Ansatz noch geschmeichelt.

Wireframe-Modus

Durch das Drücken der Taste **[W]** kann man in den Wireframe-Modus wechseln. Bei dem BSP-Baum kann man dabei gut sehen, dass scheinbar sinnlos willkürliche Schnitte durch die Geometrie laufen. Das kommt daher, dass die Polygone an entsprechend schräg verlaufenden Splitter-Ebenen geteilt wurden. Noch schöner wäre es, wenn man alle Polygone eines Leafs mit einer differenzierten Farbe rendern würde. Dann könnte man die einzelnen Leaves des Baums noch besser visualisieren. Schaltet man auf die Wireframe-Ansicht des Octrees um, dann wird man keine solchen willkürlichen Schnitte sehen. Der Octree fügt zu der ursprünglichen Geometrie ja nur Schnitte entlang den Weltachsen hinzu. Bei dem Octree werden aber auch die Bounding-Boxen der Leaves in der Wireframe-Ansicht gerendert. So kann man sehr gut die Unterteilungstiefe des Raums im Level erkennen. Dort, wo viel Geometrie vorliegt, ist der Octree wesentlich tiefer unterteilt als an Stellen mit wenig Geometrie.

13.6 Ein Blick zurück, zwei Schritt nach vorn

Gute Arbeit

Es gibt noch viel mehr über das Scene-Management zu erzählen, beispielsweise was den Scene-Graphs-Ansatz angeht. Ich hätte auch gern noch ein wenig mehr über Terrain erzählt und darüber, wie man die Quadrees dort effektiv zum Laufen bringt, oder über Patches mit Bounding-Boxen. Aber ich hoffe, in diesem Kapitel habt ihr trotzdem genügend Anregungen gefunden, um den Sinn und Zweck des Scene-Managements zu ergründen. Wie so oft im Leben gilt auch hier, dass man sich nicht stumpf für den Algorithmus ABC entscheiden sollte, weil das berühmte Spiel XYZ den ja auch verwendet. Wenn man das Prinzip der bedeutsamsten Techniken auf diesem Gebiet verstanden hat, dann kann man frei entscheiden, welche dieser bestehenden Techniken man für seine eigenen Zwecke adaptieren und entsprechend optimieren möchte. Und dazu seid ihr nun in der Lage.

Erweiterungspotenzial

In den folgenden Kapiteln dieses Buches werden wir als kleines Beispiel ein echtes Netzwerk-Spiel als Indoor-Ego-Shooter implementieren und dabei die hier entwickelte Klasse für den Octree zur Kollisionsabfrage einsetzen. Dabei ist unsere Methode für die Kollisionsabfrage mit der Level-Geometrie zwar ausreichend und funktional, aber sie könnte noch besser sein. Wir suchen hier aus dem Octree ja nur einen beliebigen Punkt der Kollision heraus, aber nicht den sinnvollsten. Sinnvoll wäre es, denjenigen Punkt zu finden, der dem kollidierenden Objekt am nächsten liegt. Dazu wäre es bes-

ser, die Kollision nicht mit den Polygonen an sich durchzuführen, sondern über Ebenen. So könnten wir eine sinnvolle Kollisionsebene zurückerhalten. Der Sinn besteht nun darin, dass wir aufgrund dieser Ebene – oder besser gesagt: ihres Normalenvektors – ganz einfach einen Vektor berechnen können, mit dem das kollidierende Objekt von der Level-Geometrie abprallt.

Wir könnten dies auch sogar ganz leicht umsetzen. Wenn wir ein Leaf in unserem Octree konstruiert und mit entsprechender Geometrie gefüllt haben, dann können wir aus dieser Geometrie für genau dieses Leaf einen BSP-Baum erstellen. Dieser ist dann nicht besonders groß, aber er ist für die Kollisionsabfrage in diesem Fall einfach besser geeignet, wenn man entsprechende Präzision braucht. Die Kollisionsabfrage funktioniert im BSP-Baum nämlich mit weniger Aufwand, als wenn wir jedes Mal die Polygone in einem Leaf des Octrees testen würden. Ein Octree, der für jedes Leaf einen BSP-Baum hat, wäre also ein interessanter Hybrid. Ich schlage vor, ihr versucht einfach mal, diese Erweiterung in unseren Octree einzubauen. Das kann sich lohnen.

*BSP-Tree im
Octree-Node*

Teil 4 Schwarze Magie der Spiele- programmierung

Kapitel 14:	Computer Aided Design (CAD)	693
Kapitel 15:	Pandoras Legacy	815
Kapitel 16:	Scripting und Definition von NPC-Verhalten	877

14 Computer Aided Design (CAD)

»Mitleid bekommst du umsonst. Neid musst du dir verdienen.«
(chinesisches Sprichwort)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Einsatz von Tools bei der Entwicklung von Computerspielen
- ➔ Struktur und Aufbau von CAD-Programmen
- ➔ Implementierung eines Editors für 3D-Geometrie und Level

14.1 Anwendungen für CAD-Tools

Entwickler, die mit konventionellen Hilfsmitteln ihre Arbeit verrichten müssen – sei dies nun die Entwicklung von Software, einer Maschine oder eines Bürokomplexes –, verdienen in der Tat unser Mitleid, das sie gemäß den weisen Chinesen auch umsonst bekommen. Doch wir wollen uns in diesem Kapitel den Neid lieber ehrlich verdienen. Das werden wir tun, indem wir einen kompletten Low-Polygon-Editor designen und implementieren. Mit diesem Tool wird es möglich sein, komplette Indoor-Level für das Spiel *Pandoras Legacy* zu konstruieren, das wir im nächsten Kapitel programmieren werden. Im folgenden Abschnitt schauen wir uns erst einmal an, wo vergleichbare Tools eingesetzt werden und welche prominenten Beispiele es für Level-Editoren noch so gibt.

Mitleid und Neid

Ingenieur, Architekt, Spiele-Entwickler

Auf den ersten Blick mögen diese drei Berufe wenig gemeinsam haben. Auf den zweiten Blick wird man dann aber doch wenigstens eine Gemeinsamkeit erkennen: Eines der Werkzeuge, mit denen alle drei arbeiten, ist die CAD-Software. Dabei steht CAD für *Computer Aided Design*, also die Unterstützung des Menschen beim Design verschiedenster Dinge durch den Computer. Der Architekt, der am Zeichenbrett auf DIN-A0-Papier Grundrisse und Außenansichten skizziert, gehört einer aussterbenden Spezies an. Bereits seit längerer Zeit verwenden Architekten CAD-Software, um Grundrisse und Pläne zu erstellen. Ein prominentes Beispiel dieser Kategorie ist das Pro-

*Drei Spezies, ein
Tool*

ogramm *ArchiCAD* der Firma *Graphisoft*. Im Bereich der Ingenieure hat sich die Konstruktion von Maschinen, Fahrzeugen usw. mittlerweile auch weg vom Reißbrett hin zum CAD verlagert. Von Heckspoilern über Flugzeugtragflächen bis hin zu einem Benzinmotor werden Maschinen nun am Computer entworfen und virtuell konstruiert. Eine hierfür sehr bekannte Software ist das Programm *AutoCAD* der Firma *Autodesk*. Und nun kommen wir auch schon zu den Spiele-Entwicklern, denn wenn schon das fertige Produkt vollständig auf dem Computer läuft, dann sollte das auf die Entwicklung ja auch größtenteils zutreffen. Tatsächlich finden wir auch hier einen Haufen verschiedener CAD-Software. Nicht weiter verwunderlich ist auch, dass die Firma *Autodesk* über die Division *Discreet* hier ebenfalls mit einem Produkt vertreten ist, nämlich mit dem bekannten *3ds max*.

*CAD ist CAD ist
CAD*

Das legt bereits die Vermutung nahe, dass sich die Grundtechnologien hinter den verschiedenen Ausprägungen von CAD-Software nicht sonderlich unterscheiden. Und das ist in der Tat so. Die ersten Editoren für Computerspiele, wie beispielsweise *wadedit* für *Doom* oder *build!* für *Duke Nukem3D*, waren in ihren Möglichkeiten noch sehr beschränkt und boten zum Konstruieren der Level bestenfalls eine zweidimensionale Draufsicht auf einen Level, aber die heutigen Tools sind ebenso komplex wie eine professionelle CAD-Software.

Level-Editing-Tools

*Von Text über 2D
zu CAD*

Die ersten Level-Editing-Tools waren eigentlich nur Textdatei-Editoren. Man konnte dort bestimmte Variablen und Attribute über eine Textdatei angeben. Auch den einfachen Aufbau eines Levels kann man durchaus auf verschiedene Weisen in eine Textdatei quetschen. Dies ist natürlich ein sehr unbefriedigendes Verfahren und sollte spätestens mit *MS-DOS* aus den virtuellen Welten verschwunden sein. Der nächste Evolutionsschritt waren dann die 2D-Editoren. Hier konnte man Level durch Setzen von Wänden und Items aus der Vogelperspektive basteln. Zu den bekanntesten Vertretern ihrer Art gehören hier sicherlich *WAD Edit* zum Editieren von *Doom*-Leveln sowie *build!* als Editor für *Duke Nukem 3D*-Level.

Den letzten Generationssprung bedeuteten dann CAD-ähnliche Tools wie beispielsweise der *WorldCraft*¹-Editor von *Valve*, der seit der Version 3.4 *Hammer*² heißt. Abbildung 14.1 ist ein Screenshot dieses Tools. Mit diesem Editor konnte man endlich alle drei Dimensionen eines Levels nahezu vollkommen frei entwickeln. Interessant sind sicherlich auch Ingame-Editoren, bei denen man den Level nicht in drei Seitenansichten und einer rein informativen 3D-Darstellung sieht, sondern bei denen man den Level einfach nur in der 3D-Ansicht sieht. Dort kann man dann Geometrien setzen, Texturen

- 1 Vgl. <http://www.planetquake.com/worldcraft/index2.shtm> für ein gutes Tutorial zum WorldCraft 1.6 Editor
- 2 Download unter: <http://www.fileplanet.com/files/80000/86826.shtml>

auftragen, Gegner positionieren usw. Das Spiel *Doom 3* soll beispielsweise einen solchen Ingame-Editor bieten. Ganz neu ist aber auch diese Idee nicht. Schon vor etlichen Jahren gab es das Tool *3D Construction Kit*, mit dem genau dies möglich war. Dort konnte man in der Ego-Perspektive einen Level erstellen und einfärben (Texturen gab es zu dieser Zeit in 3D-Spielen noch nicht) sowie einfache Scripts schreiben und den Objekten zuweisen.

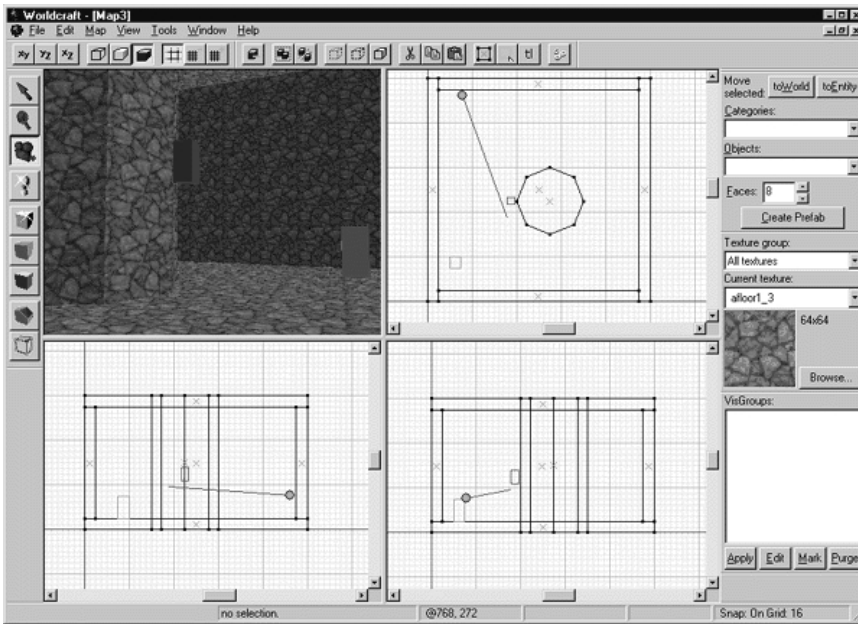


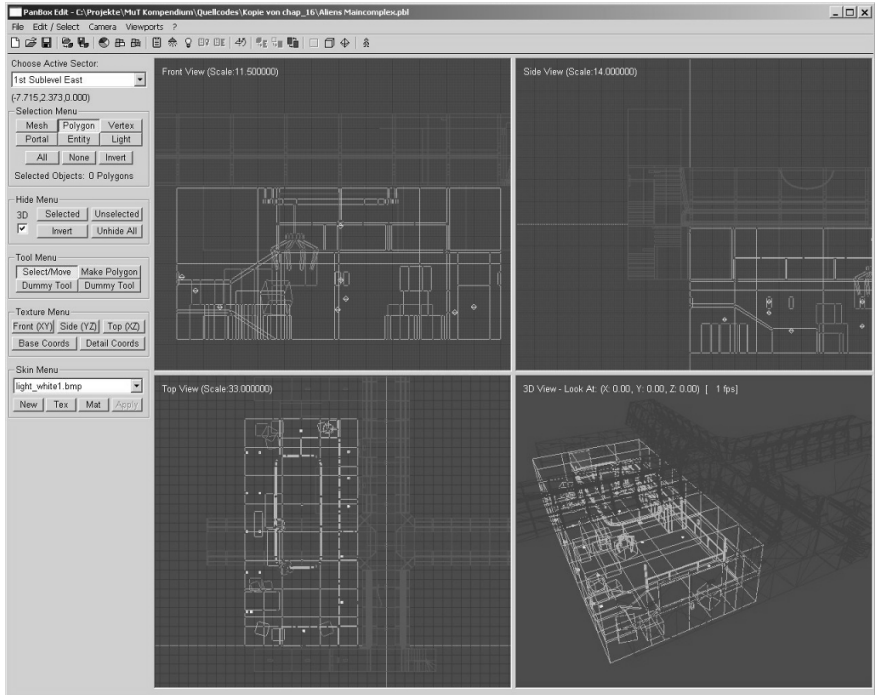
Abbildung 14.1: Screenshot des Valve Hammer-Editors. Das typische Vier-Fenster-Layout einer CAD-Anwendung ist hier gut zu erkennen. Drei orthogonale Ansichten zeigen den Level von der Seite, von vorne und von oben. Eine perspektivische Ansicht zeigt den Level in 3D.

14.2 Low-Polygon-Editor PanBox Edit

Hm, da lesen wir doch in diesem Kapitel ständig, dass wir hier einen kompletten Level-Editor programmieren wollen. Wir hören von professioneller CAD-Software und Low-Polygon-Editoren. Da ist doch etwas faul, das kann doch gar nicht in ein Buch oder sogar nur in ein Kapitel passen, oder? Gegenfrage: Warum nicht? Viele Dinge scheinen nach außen komplizierter, als sie tatsächlich sind. Dieses Kapitel wird zwar wieder recht umfangreich werden, aber dafür erhalten wir am Ende tatsächlich das gewünschte Tool – natürlich nicht in kommerzieller Qualität, aber Abstriche machen wir hier lediglich im Funktionsumfang des Editors, damit er überschaubar bleibt. Wir haben also keine Funktionen für die Polygon-Reduktion oder zur Animation in unserem Tool. Aber wir können damit vollwertig texturierte Level über ein komfortables Interface erstellen, inklusive aktivierbarer Türen und dynamischer Beleuchtung. Abbildung 14.2 zeigt einen Screenshot dessen, was wir am Ende dieses Kapitels implementiert haben werden.

Nichts ist unmöglich

Abbildung 14.2:
Screenshot von
PanBox Edit



Fähigkeiten des Tools

Erweiterungs- potenzial

Ein paar Unterschiede zu einer kommerziellen Software haben wir ja eben schon genannt. Damit wir uns aber mit viel Elan durch dieses Kapitel kämpfen können, will ich hier schon einmal eine Liste der Features angeben, die unser Tool am Ende auf alle Fälle bieten muss, damit wir sinnvoll mit ihm arbeiten können. Schließlich ist es nicht mein Ziel, euch eine kleine Applikation hinzuknallen, die nur für die Verwendung mit unserer Engine geeignet ist. Vielmehr möchte ich euch in diesem Kapitel zeigen, wie man ein Tool zur Bearbeitung von Low-Polygon-Modellen jeglicher Art implementieren kann. Da wir idealerweise ebenfalls einen Level-Editor brauchen, richten wir das Tool auch daraufhin aus. Trotzdem sollte das nicht darüber hinwegtäuschen, dass man mit diesem Editor auch 3D-Modelle für Fahrzeuge, Character usw. erstellen kann. Man muss lediglich zusätzlich zu der Speicher-Routine für das von uns definierte Format auch eine Speicher-Routine schreiben, die die Daten im *.x-Format in eine Datei schreibt; und schon hat man einen 1A-Editor für X-File-Modelle.

Anforderungen an das Tool

Ich denke, es ist klar, worauf ich hinaus will. *PanBox Edit* wird ein komplettes Tool sein, aber kein abgeschlossenes, denn es bietet euch ein beinahe unbegrenztes Erweiterungspotenzial. Aber gehen wir einmal darauf ein, was wir hier selbst implementieren werden. An Anforderungen gibt es eigentlich

nur recht wenige grobe Vorgaben, die man wie folgt auf einen Nenner bringen kann: »*PanBox Edit* soll es uns ermöglichen, texturierte Indoor-Level mit Lichtquellen und aktivierbaren Türen sowie Schatten werfenden Objekten zu konstruieren und zu bearbeiten und diese in einem geeigneten Format abzuspeichern.« Aus dieser recht simplen Anforderung leiten wir jetzt für ein besseres Verständnis unserer Aufgabe diverse Unterziele ab. Wir wollen mit diesem Tool auch die folgenden Dinge tun:

- ➔ Polygone und einfache geometrische Objekte erzeugen
- ➔ Objekte und Polygone bis auf die Ebene der Vertices editieren
- ➔ Objekte löschen und kopieren
- ➔ Objekte markieren und verstecken
- ➔ Geometrie zu einem so genannten Mesh gruppieren
- ➔ die Geometrie texturieren und die Texturkoordinaten editieren
- ➔ den Level laden und speichern
- ➔ ausgewählte Geometrie als PreFabs laden und speichern
- ➔ Startpunkte für Spieler und Gegner einfügen
- ➔ Punktlichtquellen positionieren und editieren
- ➔ Objekte markieren, die Schatten werfen sollen
- ➔ Portale für das Scene-Management definieren

Als PreFab bezeichnet man ein vorgefertigtes Objekt (engl. pre-fabricated). In unserem Kontext nennt man einzelne Teile eines 3D-Modells oder eines Levels so, die man separat speichert. Beispielsweise erzeugt man das Modell eines Schreibtischs oder eines ganzen Raums und speichert dieses als PreFab ab. Wenn man dann beim Konstruieren eines Levels wieder einen gleichen oder ähnlichen Schreibtisch oder einen Raum benötigt, spart man sich die Modellierungsarbeit und lädt einfach das PreFab in den Level.



Das ist im Großen und Ganzen schon alles, was unser Tool können muss und später auch können wird. Die Liste ist mit Sicherheit noch nicht vollständig, aber sie beschreibt schon ganz gut, was wir alles zu tun haben. In der Liste tauchen aber auch Begriffe auf, die der eine oder andere vielleicht noch nicht gehört hat. Einmal haben wir den Begriff des Scene-Managements und einmal den des Portals. Mit *Scene-Management* bezeichnen wir den Ansatz, über den wir später zur Laufzeit im Level entscheiden, welche Geometrie und welche Objekte in einem Frame gerendert werden müssen. Als Beispiele dafür seien hier die Stichworte Scene Graph, BSP-Tree, Quadtree, Octree und Portal-Engine genannt. Der Begriff *Portal* deutet bereits darauf hin, für welchen Ansatz wir uns hier entscheiden. Dies ist aber eine

*Portale und
Scene-
Management*

Thematik, die erst bei dem Entwurf des Game-Codes von Bedeutung ist, und daher begnügen wir uns zunächst damit, dass wir uns später darum kümmern. Bei der Implementierung der Portale für *PanBox Edit* werde ich noch kurz darauf eingehen. Diese Thematik wird in Kapitel 13, *Scene-Management*, und in Kapitel 15, *Pandoras Legacy*, ausführlich behandelt.

WinAPI-Rahmenanwendung

*Profi oder
Nachsitzen?*

Bevor wir die Ärmel hochkrempeln und den Compiler hochfahren, möchte ich noch kurz darauf eingehen, wie ich dieses Kapitel gestalten werde. Da dieses Buch für fortgeschrittene Programmierer gedacht ist, werde ich nicht den gesamten Quelltext der Anwendung beschreiben, weil das allein sicherlich das halbe Buch füllen würde. Insbesondere werde ich nicht detailliert auf die Implementierung der WinAPI-Rahmenanwendung eingehen. Ich werde nicht zeigen, wie man dort mit der WinAPI einen Button erstellt und einen Klick auf den Button erkennt oder wie man abfragt, ob ein Drücken der linken Maustaste erfolgt ist und ob sich der Cursor in dem Child-Window oben rechts befindet und wir dort folglich ein Objekt einfügen müssen.

WinAPI ist Pflicht!

Für alle, die bis jetzt noch keinen Plan haben, wie man das macht: Keine Panik. Für Selbstlerner und Schnelldenker weise ich auf den vollständigen Quellcode auf der CD-ROM hin. Allen anderen empfehle ich ein gutes Buch zur WinAPI wie beispielsweise den Petzold.³ Alles in allem hält sich der WinAPI-Code des Tools aber in Grenzen. Ich verwende auch keine MFC oder andere Hilfsbibliotheken, weil wir tatsächlich nicht viel mehr machen, als ein paar Buttons zu erzeugen und Mausclicks auf die Buttons und die Child-Windows abzufragen und entsprechend auf sie zu reagieren.

*Und jetzt hoch die
Ärmel*

Der restliche Code zu unserem Tool, der folglich der viel interessantere ist, weil er sich mit dem Handling von Vertices, Polygonen, Meshs usw. in dem Editor befasst, wird hier komplett erläutert.

14.3 Klassen-Design des Tools

Was steckt drin?

In diesem Abschnitt werden wir uns zunächst den allgemeinen Aufbau eines Levels ansehen, d.h. wie wir die Daten in unserem Editor verwalten wollen, denn dazu gibt es sicherlich so viele verschiedene Ansätze wie Elvis-Imitatoren. Man könnte ja einfach Listen von Vertices haben und über Indexlisten bestimmte Level-Abschnitte definieren.

Editor != Game

Bevor wir jetzt aber einen großen Vertex-Buffer initialisieren, sei noch darauf hingewiesen, dass ein Editor ein Editor ist und nicht das fertige Spiel. Im Game-Code kommt es nämlich darauf an, alles so performant wie mög-

³ Windows Programmierung. Das Entwicklerhandbuch zur WIN32- API, Microsoft Press Deutschland, 2000

lich zu gestalten, damit wir einen Level mit Lichtgeschwindigkeit rendern können. Im Editor kommt es darauf an, dass wir einen möglichst einfachen Zugriff auf alle Objekte bis hin zu einzelnen Vertices haben. Die kleinen Viecher werden wir nämlich ständig packen und editieren wollen. Wenn wir dazu über irgendwelche Indexlisten einen Vertex-Buffer verriegeln und die Daten herausziehen, editieren und wieder hineinschieben, dann klingt das nicht nur aufwändig und komplex, sondern ist es auch.

Folglich arbeiten wir hier mit einem ganz anderen Ansatz. Wir werden auf der untersten Ebene eine Klasse haben, in der wir die Vertices eines Polygons verwalten. So haben wir für jedes Polygon in unserem Level eine Instanz, die ihre eigenen Vertices hält und uns dort sehr leicht ranlässt.

Editor !

= Performance

Unsere Klasse ZFXPolygon in der Mathe-LIB dient ausschließlich dazu, mathematische Berechnungen und Kollisionsabfragen durchzuführen, und speichert nur Positions- und Normalenvektoren eines Polygons. Daher benötigen wir hier eine andere Klasse für Polygone, da wir mehr am Rendern und Verändern der Polygone interessiert sind. Dafür brauchen wir dann echte Vertices, also auch Texturkoordinaten usw.



Beim Rendern werden wir tatsächlich auch jedes einzelne Polygon an das Render-Device senden – eigentlich eine Kriegserklärung an jede Grafikkarte. Doch die OpenGL-API wird dies mit ihrem Treiber entsprechend batchen, um die Karten zu besänftigen; und was Direct3D angeht, so haben wir ja in unserer Render-DLL recht viel Aufwand betrieben, um dynamisch rendern zu können. Dort batchen wir eigenhändig Triangles mit gleichen States zusammen, um sie in großen Brocken zu rendern. Dreimal dürft ihr nun raten, warum wir uns seinerzeit die Mühe gemacht haben? :-)

Genereller Aufbau eines Levels

Natürlich steht auf der höchsten Ebene der Hierarchie der Level selbst, also entwerfen wir eine Klasse für einen Level. Dieser beinhaltet aber nachher recht wenig, denn wirbürden einen Großteil der Arbeit einem anderen Lastesel auf: dem Sektor. Ein Level untergliedert sich folglich in verschiedene Sektoren, wobei ein Sektor eine beliebige Untereinheit eines Levels ist. Das kann ein einzelner Raum sein, eine Menge von Räumen oder der halbe Level. Das ist für das Management der Objekte mehr oder weniger irrelevant. Nicht mehr ganz so irrelevant ist die Aufteilung eines Levels in Sektoren beim Rendern des Levels im Game-Code.

Der Level und sein Arbeitstier

Wir haben es beim Scene-Management schon angesprochen und werden bei *Pandoras Legacy* noch darauf zurückkommen, aber ein wenig kann ich hier schon vorgreifen: Früher war man bemüht, so wenig Pixel wie möglich am Bildschirm zu rendern. Ein BSP-Tree oder eine exakte Portal-Engine sind Scene-Management-Verfahren, die explizit darauf abzielen, aus einem Level

Sektoren für Performance

genau diejenigen Triangles oder gar Pixel zu bestimmen, die am Bildschirm zu sehen sind. Mit der heute zur Verfügung stehenden Hardware ist das ein wenig anders. Hier ist es von größerem Interesse, möglichst schnell möglichst viele Triangles zu bestimmen, die *eventuell* am Bildschirm zu sehen sind.

In unserem Ansatz bei *Pandoras Legacy* bestimmen wir immer nur die Sektoren, die am Bildschirm zu sehen sind, und rendern dann einfach nur diese Sektoren, ohne dort ein weiteres Culling der Level-Geometrie vorzunehmen. An dieser Stelle drücke ich mich schon mal vor den Steinen, die jetzt auf mich einprasseln werden. Viele Leute, die sich gerade durch den BSP-Algorithmus mit PVS gequält haben, werden zunächst nicht einsehen, warum man so viele potenziell verdeckte Polygone zur Grafikkarte senden sollte. Die Antwort ist einfach: Die Grafikkarte kann diese schneller transformieren, projizieren und rasterisieren, als wir ihre Verdeckung ermitteln könnten.

Verhältnismäßigkeit der Mittel

Natürlich ist es nicht wirklich richtig, dass eine aktuelle, durchschnittliche Grafikkarte schneller Triangles berechnet und verwerfen kann, wenn sie außerhalb des Bildschirms liegen, als wir deren Verdeckung berechnen könnten. Es gibt unzählige Ansätze, wie man eine solche Verdeckung effizient vorberechnen oder zumindest die Berechnung vorbereiten kann, beispielsweise durch einen BSP-Tree je Sektor oder eine Unterteilung eines Sektors durch einen Octree.

Man muss dabei aber immer die Verhältnismäßigkeit der Mittel bzw. des Aufwands zum Ergebnis sehen. Es ist auch zu berücksichtigen, wie aufwändig wir unsere Triangles gestalten. Verwenden wir sehr viele Effekte pro Pixel, beispielsweise durch aufwändige Pixel-Shader, Detail- und Bump-Maps, Beleuchtungen usw., dann wird ein einzelner Pixel wieder sehr teuer – gerechnet in Zeiteinheiten, die wir für das Rendern ausgeben müssen. Hier kann es sich wieder lohnen, den Overdraw (also das mehrfache Übermalen eines Pixels aufgrund von Verdeckungen) zu minimieren. Wir werden aber feststellen, dass wir mit relativ guten Effekten auch ohne zusätzliches Culling unter der Hierarchie-Ebene der Sektoren auf eine vergleichsweise sehr gute Frame-Rate kommen.

Level-Objekte in den Sektoren

Alle anderen Objekte, die in einem unserer Level auftreten können (wie z.B. Polygone, Lichtquellen, Portale oder Meshes), werden nun nicht durch die Level-Klasse verwaltet, sondern durch die Sektoren. Wenn wir einen Level konstruieren, so erzeugen wir zunächst lediglich Sektoren für den Level. Alle weiteren Objekte erzeugen wir dann für einen konkreten Sektor. Die folgende Auflistung zeigt, welche Objekte *PanBox Edit* später kennen wird:

➔ Polygone

- ➔ Meshs
- ➔ Lichtquellen
- ➔ Portale
- ➔ Entities
- ➔ Spawn-Points
- ➔ (Vertices)

Die *Polygone* an sich bzw. was sie repräsentieren dürfte klar sein. Ein *Mesh*, mathematisch korrekterweise als Polymesh bezeichnet, ist einfach eine Gruppierung mehrerer Polygone zu einem Objekt. An *Lichtquellen* wird das Tool neben dem obligatorischen ambienten Licht lediglich über Punktlichtquellen verfügen, wir verwenden also keine Spotlights oder gerichtetes Licht. Dafür kann man aber beliebig viele Lichtquellen setzen, und diese erzeugen ein Per-Pixel-Lighting. Über *Portale* haben wir schon einiges gehört und über *Entities* auch schon ein wenig. Die einzige Art Entity, die wir hier implementieren, ist eine Türfunktionalität für gleitende Türen. Die *Spawn-Points* sind diejenigen Punkte im Level, an denen der Spieler das Spiel startet bzw. an denen die Bots im Level ihre Jagd auf den Spieler beginnen. Die *Vertices* sind hier eingeklammert, weil es für sie keine eigene Klasse geben wird. Sie können als Untereinheiten eines Polygons editiert werden, sind hier aber dennoch als Objekt aufgeführt, weil man sie im Editor später separat bearbeiten kann.

Die Grundlage allen Seins: CLevelObject

Im vorigen Absatz sind alle Objekte aufgeführt, die wir in unserem Tool haben werden. Wir benötigen später auch die Möglichkeit, unsere Objekte zu identifizieren, wenn wir nicht genau wissen, was für ein Objekt sie wirklich sind. Warum wir diese Möglichkeit brauchen, dazu kommen wir gleich. Aber wir beginnen zunächst mit dem Festlegen von IDs für unsere Level-Objekte, auch liebevoll LOBs genannt.

IDs für Level-Objekte

```
#define LOB_NONE      0
#define LOB_ALL       2
#define LOB_SECTOR    4
#define LOB_MESH      8
#define LOB_POLYGON   16
#define LOB_PORTAL    32
#define LOB_ENTITY    64
#define LOB_LIGHT     128
#define LOB_SPAWNPOINT 256
```



Und nun zu dem Grund, warum wir ein Objekt in unserem Quelltext haben können, von dem wir nicht genau wissen, was für ein LOB es wirklich ist:

*Basisklasse
CLevelObject*

Wir haben viele Methoden, die für alle LOBs gleich sind. Daher bilden wir eine Basisklasse `CLevelObject`, die bereits viele dieser allgemeinen Aufgaben übernimmt und die auch durch rein virtuelle Methoden vorschreibt, welche Funktionalitäten die abgeleiteten Klassen, sprich die tatsächlichen Level-Objekte mindestens bieten müssen. Welche Vorteile eine solche Basisklasse hat, sehen wir später in diesem Kapitel, wenn wir beispielsweise Objekte in den Level einfügen oder aus dem Level löschen wollen. Aber hier ist erst einmal die Definition der Klasse:



```
class CLevelObject {
public:
    CLevelObject(void) { /* nothing */ }
    virtual ~CLevelObject(void) { /* nothing */ }

    // virtual Interface-Kram
    virtual void Init(void)=0;
    virtual void Release(void)=0;
    virtual void Save(FILE*)=0;
    virtual bool Load(FILE*)=0;
    virtual void Translate(const ZFXVector&)=0;
    virtual void CopyOf(CLevelObject*)=0;
    virtual bool Picked(const ZFXRay&, float*)=0;
    virtual bool Intersects(const ZFXAabb&,ZFXAXIS)=0;
    virtual void Render(ZFXVertexCacheManager*,
                        bool)=0;

    virtual void Reset(void) { Release(); Init(); }

    // alle kleinen Helferlein
    bool    IsOfType(DWORD dw){return(m_Type==dw);}
    DWORD   GetType(void)    {return m_Type;    }
    UINT    GetNumVerts(void) {return m_NumVerts;}
    UINT    GetNumIndis(void) {return m_NumIndis;}
    UINT    GetNumPolys(void) {return m_NumPolys;}
    ZFXAabb GetAabb(void)     {return m_Aabb;    }
    void    DeSelect(void)    {m_bSelected=false;}
    void    Select(void)      {m_bSelected=true; }
    bool    IsSelected(void)  {return m_bSelected;}
    void    UnHide(void)      {m_bHidden=false; }
    bool    IsHidden(void)    {return m_bHidden;}
    void    Hide(void)        {m_bHidden=true;  }
                                m_bSelected=false;}

    LPVOID* InitMem(LPVOID*, UINT, int);
    LPVOID* CheckMem(LPVOID*, UINT, int);

protected:
    UINT    m_NumVerts;        // Anzahl Vertices
    UINT    m_NumIndis;        // Anzahl Indices
}
```

```

UINT      m_NumPolys;      // Anzahl Polygone
bool      m_bSelected;     // ausgewählt?
bool      m_bHidden;       // versteckt?
DWORD     m_Type;          // LOB_MESH, ...
ZFXAabb   m_Aabb;          // Bounding Box
}; // class
typedef class CLevelObject *LPLEVELOBJECT;

```

Die meisten der hier bereits implementierten Methoden befassen sich mit dem Zugriff auf die Attribute der Klasse. Insbesondere kann man Level-Objekte selektieren (mit der Maus auswählen) und sie verstecken bzw. wieder anzeigen. Das Verstecken dient dazu, gewisse Objekte im Editor auszublenken, wenn sie einem die Sicht versperren und man sie für den aktuellen Editiervorgang nicht benötigt.

Attribute

Die rein virtuellen Methoden der Klasse `CLevelObject` sind fast alle selbsterklärend. Mit der Methode `CLevelObject::Picked` überprüfen wir, ob das Objekt von dem übergebenen Strahl getroffen wurde. Dies brauchen wir beim Picking der Objekte – daher wohl der Name. Als Picking bezeichnet man das Anklicken von 3D-Objekten mit der Maus. Wir benötigen das später, damit wir Objekte in der 3D-Ansicht des Editors mit der Maus selektieren können, um sie als Objekt für die Bearbeitung zu markieren. Die Methode `CLevelObject::Intersects` prüft, ob die Bounding-Box des Objekts mit der angegebenen Bounding-Box auf der angegebenen Achse überlappt. Dies brauchen wir in den zweidimensionalen Ansichten des Editors, wenn wir dort mit der Maus einen Selektionskasten aufziehen und die Objekte selektieren wollen, die sich innerhalb des Kastens befinden.

Methoden

Dann haben wir nur noch zwei Methoden, die weder rein virtuell noch oben bereits implementiert sind. Diese beiden beschäftigen sich wieder mit so einem »Igitt-Kram«, wie ihn hartgesottene C++-Programmierer nennen würden, und zwar mit der dynamischen Zuweisung von Speicher. `InitMem()` und `CheckMem()` dienen dazu, einem Zeiger eine bestimmte Menge Speicher zuzuweisen bzw. diese Menge bei Bedarf zu vergrößern. Es gibt wohl kaum ein dynamischeres System auf der Welt als unseren Level-Editor. Wir können im Voraus weder wissen noch abschätzen, wie viele Objekte eines Typs der Level-Designer benötigen wird, und die wollen alle irgendwo gespeichert werden. Folglich bleiben wir nicht auf die Dimensionen eines Arrays beschränkt, so wie beispielsweise der *build!*-Editor für Duke Nukem 3D, in dem man maximal 1024 Sektoren erstellen konnte. Damals waren die Sektoren außerdem noch sehr, sehr klein, weil eine exakte Portal-Engine verwendet wurde. Langer Rede kurzer Sinn: Hier sind die Implementierungen dieser Methoden:

*Igitt – dynamische
Speicher-
Allokation*



```

LPVOID* InitMem(LPVOID *ppMem, UINT Slots,
                int Stride) {
    if (Slots==0) return NULL;
    int n = ((int)(((float)Slots)/50)+1)*50;
    ppMem = (LPVOID*)malloc(Stride*n);
    return ppMem;
}

/*-----*/

LPVOID* CheckMem(LPVOID *ppMem, UINT Slots,
                int Stride) {
    if ( (Slots%50) == 0 )
        ppMem = (LPVOID*)realloc(ppMem,
                                   Stride*(Slots+50));
    return ppMem;
}

/*-----*/

```

Man beachte, dass wir hier jeweils Speicher in Brocken von 50 Slots bereitstellen. Wenn der Speicher für ein zusätzliches Objekt nicht mehr ausreicht, dann wird gleich Speicher für bis zu 50 weitere Objekte reserviert.

`new`, `malloc()` und Speichermanager und Co

Natürlich ist die hier gezeigte Verwendung der originären C-Funktionen wie `malloc()`, `realloc()` und `free()` für die vollkommen dynamische Speicherverwaltung aus Sicht eines Hardcore-C++-Programmierers ein grober Affront, so als würde man Poronkärstys ohne Preiselbeeren serviert bekommen. Solche Programmierer verwenden lieber die Operatoren `new` und `delete`, die dann implizit die C-Funktionen verwenden. Getreu dem Motto: Was ich nicht sehe, das macht mich auch nicht heiß.

*In einer kommerziellen Engine würde man aber diese Art der dynamischen Speicherallokation mitten im Code generell vermeiden. Dort kommen eigene Speichermanager (engl. *memory manager*) zum Einsatz. Diese reservieren beim Start für ihre Anwendung gleich einen ganzen Block von mehreren Megabyte an Speicher. Eine solche Reservierung ist wie eine Bestellung bei McDonalds: Sie braucht eine vergleichsweise lange Zeit und sollte daher zur Laufzeit vermieden werden. Möchte nun die Applikation doch zur Laufzeit dynamisch Speicherplatz belegen, dann reserviert sie sich eben nicht selbst dynamisch den Speicherplatz vom Betriebssystem. Vielmehr geht sie zum Memory-Manager und trägt dort ihr Anliegen vor. Dieser weist der Applikation dann einen entsprechend großen Block aus seinem Fundus zu, was vergleichsweise schneller geht.*

Neben der Geschwindigkeit ist natürlich auch die Sicherheit vor Speicherlecks ein weiteres Argument für einen Speichermanager. Wenn man

mit einem Speichermanager arbeitet, braucht man nicht überall über die Applikation verteilt Pointer mit dynamischem Speicher zu haben, sondern muss am Ende nur den Speicherblock des Speichermanagers freigeben – und die RAM-Welt ist wieder in Ordnung.

Auf unterster Ebene: CPolygon

Die erste Klasse, die wir von unserer Basisklasse `CLevelObject` ableiten werden, ist die Klasse `CPolygon`. Die Level, die wir hier erstellen, sind *polygonorientiert*; das bedeutet, wir bauen unsere Level aus einzelnen Polygonen oder Gruppen von Polygonen zusammen. Wir werden es später zwar auch zulassen, dass der Level-Designer einzelne Vertices editieren kann. Aber es wird nicht möglich sein, einzelne Vertices in den Level einzufügen oder aus ihm zu löschen. Das Einfügen macht an keiner Stelle Sinn, da Vertices nicht einzeln ohne Zugehörigkeit zu einem Polygon existieren können. Das Löschen könnte an einigen Stellen Sinn machen, wenn man einzelne Vertices aus einem Polygon entfernen möchte. Aber der Einfachheit halber, um den Quelltext zu unserem Tool in einem erklärbaren Umfang zu halten, verzichte ich hier darauf. Aber das wäre eine gute Hausaufgabe, wenn ihr den Editor später selbstständig erweitert und an eure Bedürfnisse anpasst.

*Polygone als
Atome unserer
Level*

Neben dem polygonorientierten Ansatz für die Konstruktion von Leveln gibt es viele Editoren, die einen brush-basierten Ansatz verwenden. Dort erstellt man nicht direkt Geometrie aus Polygonen, sondern fügt so genannte Brush-Objekte ein, die z.B. würfelförmig sind. Möchte man etwa eine Tür in einen Raum einfügen, so hat man einen Brush für den Raum und generiert dann für die Türöffnung einen so genannten Cut-Brush. Man hat also den geschlossenen Raum weiterhin als Brush und zusätzlich eine Information darüber, dass an einer bestimmten Stelle in diesem Brush ein Cut-Brush sitzt, der ein Loch in diesem Brush erzeugen soll.

Ein Brush ist also ganz allgemein eine Menge an Polygonen, und ein Level besteht aus einer Vielzahl solcher Mengen, die durch entsprechende mathematische Mengenoperationen miteinander verknüpft sind. Beispielsweise kann man aus mehreren Mengen eine Vereinigungsmenge oder eine Schnittmenge bilden. Diese Berechnungen von Mengen bezeichnet man in der Computergrafik allgemein als Constructive Solid Geometry – oder kurz CSG.

Klassen-Deklaration

Unsere Klasse für Polygone ist die umfangreichste Ableitung von `CLevelObject`. Das hängt damit zusammen, dass wir die gesamte Geometrie eines Levels auf Polygone herunterbrechen. Viele der Aufgaben, die unsere Klasse dabei erledigen muss, sind trivial, beispielsweise das Festlegen der SkinID, die für das Polygon verwendet werden soll. Andere Methoden, wie beispielsweise die Berechnung der Texturkoordinaten, sind ein klein wenig komplexer. Aber beginnen wir wie üblich mit der Klassen-Definition:

*There is no spoon
– but lots of
polygons*



```

class CPolygon : public CLevelObject {
public:
    CPolygon(void);
    virtual ~CPolygon(void);

    void    CopyOf(CLevelObject*);
    bool    CreatePortal(CPortal*);

    void    SetVertices(const VERTEX*, UINT);
    void    SetIndices(const WORD*, UINT);

    void    SetSkinID(UINT ID) { m_SkinID = ID; }
    UINT    GetSkinID(void)    { return m_SkinID; }
    CPolymesh* GetParent(void) { return m_pDad; }
    bool    IsPartOfMesh(void) {return m_bPartOfMesh;}

    void    GetTexOff(float *pU, float *pV) {
        if (pU) *pU = m_fTexOff[0];
        if (pV) *pV = m_fTexOff[1]; }
    void    GetTexRep(float *pU, float *pV) {
        if (pU) *pU = m_fTexRep[0];
        if (pV) *pV = m_fTexRep[1]; }

    void    Render(ZFXVertexCacheManager*,bool);

    void    Translate(const ZFXVector&);
    void    Rotate(ZFXAXIS, const ZFXVector&, float);
    void    Mirror(ZFXAXIS, const ZFXVector&, float);
    bool    Picked(const ZFXRay&, float*);
    bool    Intersects(const ZFXAabb&, ZFXAXIS);
    void    GetIntersectingVerts(const ZFXAabb&,
                                CSelectionBuffer*,ZFXAXIS);
    void    InsideOut(void);
    void    SetAsPartOfMesh(CPolymesh *p);
    void    Triangulate(void);
    void    CalcBoundingBox(void);
    void    CalcNormals(void);
    void    CalcTextureCoords(ZFXAXIS,const ZFXAabb*);
    void    TransTextureCoords(float ftU, float ftV,
                                float frU, float frV);
    void    GetTextureTrans(float*,float*,float*,
                                float*);

    void    Save(FILE*);
    bool    Load(FILE*);

private:
    VERTEX  *m_pVerts;           // Vertices
    WORD     *m_pIndis;          // Indices
    WORD     *m_pLineIndis;      // Polyline-Indices
    CPolymesh* m_pDad;           // Falls Mesh-Teil

```

```

UINT      m_SkinID;          // Skin-ID
bool      m_bPartOfMesh;     // Teil von Mesh?
float     m_fTexOff[2];      // Textur-Offset
float     m_fTexRep[2];      // Textur-Repeat

void      Init(void);
void      Release(void);
}; // class
typedef class CPolygon *LPPOLYGON;

```

Zwei Dinge sollten uns in dieser Klasse spanisch vorkommen: Zum einen haben wir hier Objekte des Typs `CSelectionBuffer` verwendet, und zum anderen kann ein `Polygon` anscheinend einen Vater haben, der vom Typ `CPolymesh` ist. Beginnen wir mit dem Vater: Ich hatte ja bereits gesagt, dass unser Editor auch Polygone gruppieren können soll. So eine Gruppe von Polygonen nennen wir ein *Polymesh*, oftmals aus Maulfaulheit auch einfach nur *Mesh* genannt. Wenn ein `Polygon` in unserem Editor Teil einer solchen Gruppe, also eines Meshs, ist, dann nennen wir dieses Mesh-Objekt seinen Vater. Der Selektions-Buffer ist durch seinen Namen eigentlich schon gut genug beschrieben. Immer wenn wir in unserem Editor etwas selektieren, gibt der Selektions-Buffer an, welche Objekte selektiert sind. Diesen Buffer implementieren wir als eigene Klasse, wie wir später noch sehen werden.

*Selektions-Buffer
und Polymeshs*

Die Accessor-Methoden unserer `CPolygon`-Klasse brauche ich nicht weiter zu erklären. Wir gehen direkt weiter zum Konstruktor und Destruktor der Klasse und schauen, was dort passiert.

*Konstruktor und
Destruktor*

```

CPolygon::CPolygon(void) { Init(); }
CPolygon::~CPolygon(void) { Release(); }

```

Wie ihr seht, tut sich hier nichts Spannendes, außer dass die entsprechenden Methoden aufgerufen werden, die dann die Arbeit erledigen. Also gehen wir direkt weiter zur Initialisierung eines Polygons. Hier vergeben wir lediglich für alle Attribute einen sinnvollen Default-Wert:

```

void CPolygon::Init(void) {
    m_NumVerts    = 0;
    m_NumIndis    = 0;
    m_NumPolys    = 1;
    m_SkinID      = 2;        // Default-Textur
    m_fTexOff[0]  = 0.0f;
    m_fTexOff[1]  = 0.0f;
    m_fTexRep[0]  = 1.0f;
    m_fTexRep[1]  = 1.0f;
    m_bSelected   = false;
    m_bHidden     = false;
    m_bPartOfMesh = false;
    m_pDad        = NULL;
}

```



*Initialisierung
eines Polygons*



```

m_pVerts      = NULL;
m_pIndis      = NULL;
m_pLineIndis  = NULL;
m_Type        = LOB_POLYGON;
memset(&m_Aabb, 0, sizeof(ZFXAabb));
} // Init

```

**Basisklasse nicht
vergessen!**

Man beachte, dass einige der Attribute aus der Basisklasse stammen. Wichtig ist hierbei insbesondere die Festlegung des LOB-Typs als Polygon. Interessant ist vielleicht auch der Wert für die SkinID. Normalerweise sollte der Level-Designer jedem Polygon eine Skin zuweisen. Der Editor kennt jedoch keine Polygone ohne eine Skin, und daher erhält jedes Polygon initial die ID-Nummer 2. Die ersten beiden IDs sind für besondere Skins reserviert, mit denen Lichtquellen und Portale im Level-Editor angezeigt werden. Folglich ist die Nummer 2 die erste echte Skin.

**Zurücksetzen
eines Polygons**

Wenn man ein Polygon freigeben oder einfach zurücksetzen möchte, dann ruft man die Methode `CPolygon::Release` auf, wie es auch der Destruktor der Klasse tut. In dieser Methode wird einfach der dynamisch allokierte Speicher freigegeben.



```

#define SAFE_DELETE(p)  {if(p!=0){delete(p);(p)=0;}}
#define SAFE_DELETE_A(p) {if(p!=0){delete[](p);(p)=0;}}
#define SAFE_FREE(p)    {if(p!=0){free(p);(p)=0;}}

void CPolygon::Release(void) {
    SAFE_DELETE_A(m_pVerts);
    SAFE_DELETE_A(m_pIndis);
    SAFE_DELETE_A(m_pLineIndis);
}

```

Erzeugen einer Instanz

**Vertices und
Indices festlegen**

Wenn man eine Instanz der Klasse `CPolygon` betanken möchte, dann muss man die Indices und die Vertices für die Instanz angeben. Diesen Job erledigen die Methoden `CPolygon::SetVertices` und `CPolygon::SetIndices`. Dabei führt das Setzen der Vertices dazu, dass die Bounding-Box für das Polygon neu berechnet wird, um sich den Abmessungen der neuen Menge Vertices anzupassen. Das Setzen der Indices führt dazu, dass der Normalenvektor des Polygons berechnet wird. Wir wissen ja nun, dass der Normalenvektor wichtig ist, um das Frontface und das Backface des Polygons auseinander zu halten. Sonst passiert in den Methoden nicht viel Dramatisches, außer dass wir entsprechende Mengen an Speicher allokierten und dann die Daten aus der Parameterliste in die Attribute des Polygons kopieren.



```

void CPolygon::SetVertices(const VERTEX *pVerts,
                          UINT NumVerts) {
    SAFE_DELETE_A(m_pVerts);
    SAFE_DELETE_A(m_pLineIndis);

    m_pVerts = new VERTEX[NumVerts];
    m_pLineIndis = new WORD[NumVerts*2];

    if ( !m_pVerts || !m_pLineIndis ) return;

    memcpy(m_pVerts, pVerts, sizeof(VERTEX)*NumVerts);
    m_NumVerts = NumVerts;

    for (UINT i=0; i<NumVerts; i++) {
        m_pLineIndis[(i*2)] = i;
        m_pLineIndis[(i*2)+1] = i+1;

        if (i==NumVerts-1) m_pLineIndis[(i*2)+1] = 0;
    }

    CalcBoundingBox();
} // SetVertices
/*-----*/

void CPolygon::SetIndices(const WORD *pIndis,
                         UINT NumIndis) {
    SAFE_DELETE_A(m_pIndis);
    m_pIndis = new WORD[NumIndis];
    if (!m_pIndis) return;
    memcpy(m_pIndis, pIndis, sizeof(WORD)*NumIndis);
    m_NumIndis = NumIndis;

    CalcNormals();
} // SetIndices

```

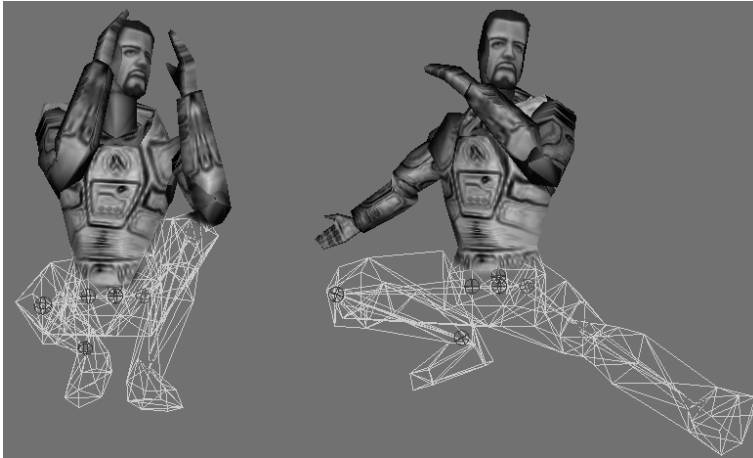
Auffällig ist hier vielleicht noch, dass wir im Attribut `m_pLineIndices` eine Anzahl an Indices beim Setzen der Vertices selbst generieren und speichern. Das liegt daran, dass wir das Polygon eventuell auch nur als Umriss rendern möchten und nicht als gefülltes Polygon mit Texturen. Der normale Drahtgitter-Rendermodus von Direct3D würde dazu führen, dass wir eben nicht den Umriss des Polygons sehen, sondern die Umrisse der einzelnen Dreiecke, in die das Polygon beim Rendern zerlegt (trianguliert) wird. Also müssen wir das Polygon im Linien-Rendermodus als Polylinie rendern. Das bedeutet, wir ziehen eine Linie vom ersten Vertex über die folgenden Vertices bis hin zum letzten Vertex und von dort aus wiederum zum ersten Vertex. Abbildung 14.3 verdeutlicht dieses Vorgehen. Wenn ein Level-Designer das ganz links stehende Polygon erzeugt, so wäre er gewiss sehr verduzt, wenn er in der orthogonalen Ansicht im Editor die triangulierte Variante sehen würde. Schließlich entspräche das nicht direkt dem, was er generiert hat.

Polylines

Außerdem sind in den orthogonalen Ansichten schon bei kleinen Leveln eine Unmenge an Linien auf dem Bildschirm zu sehen, so dass die zusätzlichen Innenkanten eines Polygons die Darstellung unnötig komplex werden lassen. Grund genug also, dass wir uns die Mühe einer sauberen Umrissdarstellung machen. Die Umsetzung beim Rendern werden wir im weiter unten folgenden Abschnitt über das Rendern von Instanzen der `CPolygon`-Klasse sehen.

Abbildung 14.3:

Ein Polygon, gefüllt mit Textur, als Umriss und als Triangulation gerendert (v.l.n.r.)



Laden und Speichern

Polygone in der Level-Datei speichern und aus ihr laden

Natürlich nützt uns ein Editor herzlich wenig, wenn wir die erstellten Level nicht speichern und dann wieder laden können. In der Klasse `CLevelObject` haben wir ja schon gesehen, dass die entsprechenden Methoden für alle Ableitungen implementiert werden müssen. Auf den Aufbau der Level-Datei kommen wir detailliert zu sprechen, wenn wir die `CLevel`-Klasse entwerfen. Hier sei nur so viel erwähnt: Jedes LOB muss sich selbst in eine angegebene, bereits offene Datei speichern bzw. aus dieser heraus laden können. Für die `CPolygon::Save`-Methode sieht das dann wie folgt aus:



```
void CPolygon::Save(FILE *pFile) {
    if (!pFile) return;

    fwrite(&m_NumVerts,    sizeof(UINT),    1, pFile);
    fwrite(&m_NumIndis,    sizeof(UINT),    1, pFile);
    fwrite(&m_SkinID,      sizeof(UINT),    1, pFile);
    fwrite(&m_Aabb,        sizeof(ZFXAabb),  1, pFile);
    fwrite(&m_bPartOfMesh, sizeof(bool),    1, pFile);
    fwrite(m_fTexOff,      sizeof(float),    2, pFile);
    fwrite(m_fTexRep,      sizeof(float),    2, pFile);

    fwrite(m_pVerts, sizeof(VERTEX), m_NumVerts, pFile);
    fwrite(m_pIndis, sizeof(WORD), m_NumIndis, pFile);
}
```

```

    fwrite(m_pLineIndis, sizeof(WORD), m_NumVerts*2, pFile);
    fflush(pFile);
} // Save
/*-----*/

bool CPolygon::Load(FILE *pFile) {
    bool bResult=true;

    if (!pFile) return false;

    fread(&m_NumVerts,    sizeof(UINT),    1, pFile);
    fread(&m_NumIndis,    sizeof(UINT),    1, pFile);
    fread(&m_SkinID,      sizeof(UINT),    1, pFile);
    fread(&m_Aabb,        sizeof(ZFXAabb), 1, pFile);
    fread(&m_bPartOfMesh, sizeof(bool),    1, pFile);
    fread(m_fTexOff,      sizeof(float),   2, pFile);
    fread(m_fTexRep,      sizeof(float),   2, pFile);

    if (m_NumVerts == 0) bResult = false;

    m_pVerts    = new VERTEX[m_NumVerts];
    m_pIndis    = new WORD[m_NumIndis];
    m_pLineIndis = new WORD[m_NumVerts*2];

    if ( !m_pVerts || !m_pIndis || !m_pLineIndis)
        return false;

    fread(m_pVerts, sizeof(VERTEX), m_NumVerts, pFile);
    fread(m_pIndis, sizeof(WORD),   m_NumIndis, pFile);
    fread(m_pLineIndis, sizeof(WORD), m_NumVerts*2, pFile);

    CalcNormals();
    return bResult;
} // Load

```

Diese Methoden zum Laden und Speichern werden bei allen LOBs nahezu identisch aussehen, auch wenn sich die Attribute der jeweiligen Klassen natürlich voneinander unterscheiden werden. Aber beim Speichern tut man nun mal nichts anderes, als alle notwendigen Daten für ein identisches Wiederherstellen des Objekts in eine Datei zu schreiben. Beim Laden liest man diese einfach wieder aus und reserviert noch entsprechenden Speicher für die zu ladenden Daten.

*Objekte kümmern
sich um sich
selbst!*

Nun können wir einer Instanz der Klasse CPolygon also unter Angabe eines gültigen Zeigers auf eine offene Binärdatei sagen, dass sie sich dorthin speichern bzw. von dort her laden soll.

Kopieren einer Instanz

Kein '='-Operator

Ebenso wie die Funktionen zum Laden und Speichern müssen wir für alle LOBs auch eine Möglichkeit haben, diese zu kopieren. Schließlich werden wir sehr viel mit diesen LOBs herumjonglieren und an ihnen herumeditieren. Wir verwenden aber explizit keine Überladung des Zuweisungsoperators, sondern schreiben uns eine eigene Methode dafür, weil ich solche Methodennamen einfach (an-)sprechender finde.



```
void CPolygon::CopyOf(CLevelObject *pLob) {
    if ( !pLob->IsOfType(m_Type) ) return;

    LPPOLYGON pPoly = (LPPOLYGON)pLob;

    Reset();
    SetVertices(pPoly->m_pVerts, pPoly->m_NumVerts);
    SetIndices(pPoly->m_pIndis, pPoly->m_NumIndis);
    m_SkinID = pPoly->m_SkinID;
    m_Aabb   = pPoly->m_Aabb;

    m_fTexOff[0] = pPoly->m_fTexOff[0];
    m_fTexOff[1] = pPoly->m_fTexOff[1];
    m_fTexRep[0] = pPoly->m_fTexRep[0];
    m_fTexRep[1] = pPoly->m_fTexRep[1];
} // CopyOf
```

Die benötigten Attribute werden hier aus der als Parameter angegebenen Instanz der CPolygon-Klasse in die aktuelle Instanz, also in das `this`-Objekt, kopiert. Eine kleine Sicherheitsabfrage sorgt dafür, dass es sich auch wirklich um eine Instanz derselben Klasse handelt. Der Parameter musste hier natürlich vom Typ `CLevelObject` sein, weil wir die Methode sonst nicht allgemein in der Basisklasse hätten definieren können. Gewisse Attribute, wie beispielsweise Zähler und Bounding-Box, werden durch das Aufrufen der Methoden zum Setzen der Vertices und Indices automatisch mit gesetzt.

Kollisionsabfrage

Was für
Kollisionen?

Eine Kollisionsabfrage für Polygone? In einem Level-Editor? Das sollte man vielleicht ein wenig relativieren. Es geht hier natürlich nicht um das Erkennen von Kollisionen der Objekte untereinander oder um Kollisionen von Level-Objekten miteinander. Es geht darum, dass der Level-Designer die LOBs mit der Maus auswählen kann. Hierbei treten verschiedene Fälle auf. In der perspektivischen (dreidimensionalen) Ansicht kann man in unserem Editor LOBs auswählen, indem man sie mit der Maus anklickt. Doch man kann die Objekte natürlich auch in den orthogonalen (zweidimensionalen) Ansichten auswählen. Dort muss man mit der Maus klicken und bei gedrückter Maustaste ein Selektionsrechteck aufziehen. Wir brauchen also

eine Methode, mit der wir feststellen können, ob eine Instanz der `CPolygon`-Klasse irgendwie innerhalb dieses Selektionsrechtecks liegt.

Das »Irgendwie« setzen wir so um, dass wir einfach prüfen, ob einer der Vertices des Objekts in diesem Selektionsrechteck liegt. Die folgende Methode übernimmt dazu eine AABB und eine Achse als Parameter. Das Selektionsrechteck ist zwar zweidimensional, aber wir verwenden dennoch eine dreidimensionale AABB und geben zusätzlich an, welche Dimension bei der Betrachtung keine Rolle spielen soll. Natürlich könnten wir auch gleich eine zweidimensionale `RECT`-Struktur verwenden, aber es ist z.B. auch vorstellbar, dass man eine begrenzte Selektionsreichweite angeben möchte und dann tatsächlich die Dreidimensionalität einer echten AABB ausnutzen kann. Wir verwenden hier aber eine Selektion bis in alle Unendlichkeit ohne Reichweitenbeschränkung. Ob ein Polygon nun 10 Meter oder 10 Millionen Kilometer von der Kamera entfernt ist, spielt für uns keine Rolle.

*Markieren von
Polygonen mit der
Maus*

```
bool CPolygon::Intersects(const ZFXAabb &aabb,
                          ZFXAXIS axis) {

    if (!m_Aabb.Intersects(aabb)) return false;

    for (UINT i=0; i<m_NumVerts; i++) {

        // vereinfache zu einem 2D-Problem
        if (axis == X_AXIS) {
            if ((m_pVerts[i].y < aabb.vcMax.y) &&
                (m_pVerts[i].y > aabb.vcMin.y) &&
                (m_pVerts[i].z < aabb.vcMax.z) &&
                (m_pVerts[i].z > aabb.vcMin.z))
                return true;
        }
        else if (axis == Y_AXIS) {
            if ((m_pVerts[i].x < aabb.vcMax.x) &&
                (m_pVerts[i].x > aabb.vcMin.x) &&
                (m_pVerts[i].z < aabb.vcMax.z) &&
                (m_pVerts[i].z > aabb.vcMin.z))
                return true;
        }
        else if (axis == Z_AXIS) {
            if ((m_pVerts[i].y < aabb.vcMax.y) &&
                (m_pVerts[i].y > aabb.vcMin.y) &&
                (m_pVerts[i].x < aabb.vcMax.x) &&
                (m_pVerts[i].x > aabb.vcMin.x))
                return true;
        }
    } // for
    return false;
} // Intersects
```



*Markieren
einzelner Vertices
mit der Maus*

Durch diese Methode kann man also herausfinden, ob ein Polygon im Selektionsrechteck liegt und damit markiert bzw. ausgewählt werden soll. Eingangs erwähnte ich aber, dass wir auch die Möglichkeit haben möchten, einzelne Vertices der Polygone zu editieren. Dazu schreiben wir eine fast analoge Methode, die jedoch prüft, ob einzelne Vertices in der AABB in zwei Dimensionen enthalten sind. Jetzt wird es etwas knifflig; hier kommt nämlich noch der Selektions-Buffer ins Spiel.

Selektions-Buffer

Normalerweise bieten unsere Level-Objekte eine Methode an, die `true` oder `false` zurückliefert, je nachdem, ob das Objekt in dem Selektionsrechteck liegt oder nicht. Auf einer höheren Ebene des Programms wird dieses Ergebnis ausgewertet, und die entsprechenden Objekte werden dann in den Selektions-Buffer gepumpt. Da wir aber Vertices nicht als Objekte einer Klasse umgesetzt haben, ist dies auf dieser Ebene aber nicht möglich. Wenn wir Vertices selektieren, so speichern wir statt eines Objekts einfach die Speicheradressen der selektierten Vertices im Selektions-Buffer. Diesen Buffer implementieren wir zwar erst später, aber wir setzen hier einfach einmal eine geeignete Methode voraus. :-)



```
void CPolygon::GetIntersectingVerts(const ZFXAabb &aabb,
                                   CSelectionBuffer *pSB,
                                   ZFXAXIS axis) {
    for (UINT i=0; i<m_NumVerts; i++) {
        // Vereinfachung auf 2D-Problem
        if (axis == X_AXIS) {
            if ((m_pVerts[i].y < aabb.vcMax.y) &&
                (m_pVerts[i].y > aabb.vcMin.y) &&
                (m_pVerts[i].z < aabb.vcMax.z) &&
                (m_pVerts[i].z > aabb.vcMin.z))
                pSB->AddVertex(&m_pVerts[i], this);
        }
        else if (axis == Y_AXIS) {
            if ((m_pVerts[i].x < aabb.vcMax.x) &&
                (m_pVerts[i].x > aabb.vcMin.x) &&
                (m_pVerts[i].z < aabb.vcMax.z) &&
                (m_pVerts[i].z > aabb.vcMin.z))
                pSB->AddVertex(&m_pVerts[i], this);
        }
        else if (axis == Z_AXIS) {
            if ((m_pVerts[i].y < aabb.vcMax.y) &&
                (m_pVerts[i].y > aabb.vcMin.y) &&
                (m_pVerts[i].x < aabb.vcMax.x) &&
                (m_pVerts[i].x > aabb.vcMin.x))
                pSB->AddVertex(&m_pVerts[i], this);
        }
    }
} // GetIntersectingVerts
```

Damit wären die Selektionsprozesse in den orthogonalen Ansichten abgedeckt. Bleibt noch der perspektivische Fall. Ein Mausklick im 3D-Raum entspricht natürlich keinem Punkt, sondern einem Strahl, da die Tiefe des Klicks ja nicht definiert werden kann. Man erzeugt also einen Strahl an der Stelle, die der Anwender mit der Maus angeklickt hat. Der Strahl geht von der Position der Kamera aus und hat eine unbegrenzte Reichweite. Jetzt prüfen wir nach, ob eine Instanz der `CPolygon`-Klasse von diesem Strahl *gepickt* (also erfasst) wurde. Ist das der Fall, dann melden wir das an den Aufrufer zurück, zusammen mit der Distanz bis zu der Stelle, an der der Strahl das Polygon trifft. Dies ist die Entfernung des Polygons an der getroffenen Stelle zu der Kamera.

*Anklicken des
Polygons mit der
Maus*

```
bool CPolygon::Picked(const ZFXRay &Ray, float *pFD) {
    ZFXVector vc0, vc1, vc2;
    WORD      I0, I1, I2;

    for (UINT i=0; i<m_NumIndis; i+=3) {

        I0 = m_pIndis[i];
        I1 = m_pIndis[i+1];
        I2 = m_pIndis[i+2];

        vc0.Set( m_pVerts[I0].x, m_pVerts[I0].y,
                 m_pVerts[I0].z );
        vc1.Set( m_pVerts[I1].x, m_pVerts[I1].y,
                 m_pVerts[I1].z );
        vc2.Set( m_pVerts[I2].x, m_pVerts[I2].y,
                 m_pVerts[I2].z );

        if ( ((ZFXRay)Ray).Intersects(vc0, vc1, vc2,
                                       true, pFD) )
            return true;
    }
    return false;
} // Picked
```



Ohne große Umschweife laufen wir durch die Triangulation des Polygons und prüfen jedes Triangle einzeln auf eine Kollision mit dem Strahl. Eine Kollision reicht uns jedoch. Wenn wir eine solche gefunden haben, dann können wir das sofort zurückmelden.

Rendern einer Instanz

Demit sind wir schon beim Rendern einer Instanz der `CPolygon`-Klasse angelangt. Wir erwarten als Parameter den Vertex-Manager der `ZFXEngine`. Zusätzlich geben wir noch durch ein Flag an, ob wir versteckte Level-Objekte rendern wollen oder nicht. Das Attribut `m_bHidden` der Basisklasse gibt ja an, ob ein Level-Objekt vom Level-Designer ausgeblendet wurde oder nicht. Es gibt aber Situationen, in denen wir eigentlich ausgeblendete

*Tischlein,
versteck dich*

Objekte trotzdem rendern möchten. In den orthogonalen Ansichten sind versteckte Objekte zwar grundsätzlich nicht sichtbar, aber in der perspektivischen Ansicht kann man die versteckten Objekte trotzdem rendern. Dazu dienen also diese Parameter.



```
void CPolygon::Render(ZFXVertexCacheManager *pVCM,
                    bool bHide) {
    if (m_bHidden && bHide) return;

    bool bAsLine = (pVCM->GetShadeMode() ==
                    RS_SHADE_LINES);
    if (bAsLine) {
        pVCM->Render(VID_UU, m_NumVerts, m_NumVerts*2,
                    m_pVerts, m_pLineIndis, m_SkinID);
    }
    else {
        pVCM->Render(VID_UU, m_NumVerts, m_NumIndis,
                    m_pVerts, m_pIndis, m_SkinID);
    }
} // Render
```

Polyline versus gefülltes Polygon

Das eigentliche Rendern über unser Interface ist nun keiner Erwähnung mehr wert. Interessant ist nur die Abfrage, ob unsere Render-DLL sich gerade im Modus für das Rendern von Linien befindet. In diesem Fall rendern wir unser Polygon über die Linien-Indices, um einen korrekten Umriss für das Polygon zu erzeugen (vgl. Abbildung 14.3). Anderenfalls rendern wir über die normale Indexliste.

Transformationen für Polygone

Verschieben eines Polygons

Jetzt sind wir mitten drin im Editieren von Polygonen. Und das ist nun wirklich einfacher als Pfannkuchenessen. Beginnen wir mit dem Verschieben eines Polygons. Wie nicht anders zu erwarten, kann man ein Polygon um einen bestimmten Vektor verschieben, indem man alle seiner Vertices um diesen Vektor verschiebt.



```
void CPolygon::Translate(const ZFXVector &vcT) {
    for (UINT i=0; i<m_NumVerts; i++) {
        m_pVerts[i].x += vcT.x;
        m_pVerts[i].y += vcT.y;
        m_pVerts[i].z += vcT.z;
    }
    m_Aabb.vcCenter += vcT;
    m_Aabb.vcMax    += vcT;
    m_Aabb.vcMin    += vcT;

    if (m_bPartOfMesh) m_pDad->CalcBoundingBox();
} // Translate
```


Natürlich sind wir auch schlau genug, die eigene Bounding-Box des Polygons mit zu verschieben, und auch an die liebe Verwandtschaft denken wir. Wenn das Polygon, das wir hier verschieben, Teil eines Polymesh-Objekts ist, dann müssen wir dessen Bounding-Box – nein, nicht verschieben, sondern neu berechnen. Schließlich verschieben wir nur einen Teil des Meshs und nicht das Mesh als Ganzes.

Ganz Fuchs

Das Rotieren eines Polygons ist nicht ganz so offensichtlich, wie es zunächst scheint. Aus unserer frühesten 3D-Jugend wissen wir ja noch, dass ein Objekt ungewollt auch verschoben wird, wenn wir es um einen Fixpunkt rotieren, der nicht sein Schwerpunkt ist. Da wir aber die Koordinaten unserer Polygone im Weltkoordinatensystem angeben, müssen wir ein Polygon zuerst mit seinem Zentrum an den Ursprung des Weltkoordinatensystems verschieben, es dort rotieren und dann wieder zurück verschieben. Die folgende Methode übernimmt als Parameter die Achse, auf der rotiert werden soll, den Vektor für die Verschiebung zum Fixpunkt hin und einen Wert für die Rotation in Radian. Natürlich denken wir auch hier wieder daran, dem Vater, so er denn existiert, eine neue Bounding-Box anzupassen.

*Rotieren eines
Polygons*

```
void CPolygon::Rotate(ZFXAXIS axis,
                    const ZFXVector &vcPos,
                    float a) {
    ZFXMatrix mat;
    ZFXVector vc;

    if (axis == X_AXIS) mat.RotaX(a);
    if (axis == Y_AXIS) mat.RotaY(a);
    if (axis == Z_AXIS) mat.RotaZ(a);

    for (UINT i=0; i<m_NumVerts; i++) {
        // Retranslation zum Ursprung
        vc.x = m_pVerts[i].x - vcPos.x;
        vc.y = m_pVerts[i].y - vcPos.y;
        vc.z = m_pVerts[i].z - vcPos.z;

        // rotieren
        vc = mat * vc;

        // zurück an Position verschieben
        m_pVerts[i].x = vc.x + vcPos.x;
        m_pVerts[i].y = vc.y + vcPos.y;
        m_pVerts[i].z = vc.z + vcPos.z;
    }

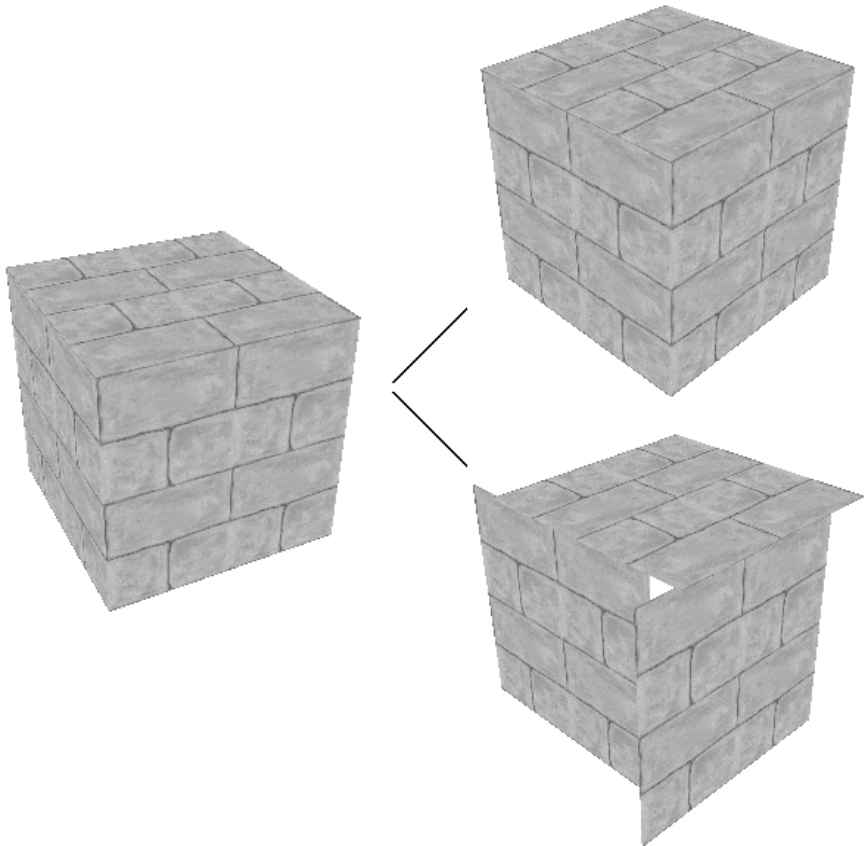
    CalcBoundingBox();
    if (m_bPartOfMesh) m_pDad->CalcBoundingBox();
} // Rotate
```



*Wohin
verschieben?*

Als Vektor für die Verschiebung könnte man prinzipiell den Vektor zum Zentrum der Bounding-Box des Polygons verwenden und diesen invertieren. Damit hätte man das Polygon direkt in den Ursprung verschoben. Der Grund, warum wir das nicht direkt in der Methode machen, ist ganz einfach: Wir können ein Objekt nur rotieren, wenn wir es selektiert haben. Wenn wir nun aber mehrere Polygone gleichzeitig selektieren und rotieren wollen, so hätte jedes einzelne Polygon einen anderen Fixpunkt, und wir würden die selektierten Objekte nicht wie gewünscht um denselben Punkt rotieren. Abbildung 14.4 verdeutlicht das auftretende Problem. Wir werden stattdessen den Vektor zum Mittelpunkt der Bounding-Box um alle Objekte im Selektions-Buffer invertieren und dafür verwenden.

Abbildung 14.4:
Rotation der einzelnen Polygone eines Würfels (links), einmal um denselben Fixpunkt (oben rechts) und einmal jeweils um das Zentrum eines Polygons (unten rechts)



*Spiegeln eines
Polygons*

Eine weitere Art der Transformation für Polygone ist das Spiegeln eines Polygons an einer der drei Achsen im Raum. Besonders bei der Konstruktion symmetrischer Objekte ist das später, wenn wir Level bauen, sehr hilfreich. Man erstellt beispielsweise eine Seite eines Gangs mit Verstrebnungen, Rohren usw., fügt diese dann als gespiegelte Kopie ein und hat die andere Seite des Gangs mit zwei Mausklicks fertig.



```
void CPolygon::Mirror(ZFXAXIS axis,
                    const ZFXVector &vcPos) {
    for (UINT i=0; i<m_NumVerts; i++) {
        m_pVerts[i].x -= vcPos.x;
        m_pVerts[i].y -= vcPos.y;
        m_pVerts[i].z -= vcPos.z;

        // spiegeln
        if (axis == X_AXIS) m_pVerts[i].x *= -1.0f;
        if (axis == Y_AXIS) m_pVerts[i].y *= -1.0f;
        if (axis == Z_AXIS) m_pVerts[i].z *= -1.0f;

        m_pVerts[i].x += vcPos.x;
        m_pVerts[i].y += vcPos.y;
        m_pVerts[i].z += vcPos.z;
    }

    CalcBoundingBox();
    if (m_bPartOfMesh) m_pDad->CalcBoundingBox();
} // Mirror
```

Das Spiegeln funktioniert so, dass man einfach die Koordinate auf der entsprechenden Achse invertiert bzw. mit -1 multipliziert. Das funktioniert aber nur so lange, wie sich das Objekt mit seinem Mittelpunkt im Ursprung befindet. Folglich müssen wir auch hier das Polygon zunächst zum Ursprung hin verschieben, dann die Koordinate invertieren und das Polygon dann wieder auf seine alte Position schieben.

Texturkoordinaten für Polygone

Neben den bereits direkt in der Klassendeklaration implementierten Methoden `CPolygon::GetTexOff` und `CPolygon::GetTexRep` benötigen wir noch drei andere Methoden für die Arbeit mit Texturkoordinaten. Grundsätzlich weisen wir die Texturkoordinaten nicht einzelnen Vertices zu, sondern einem Polygon als Ganzes. Dabei müssen wir neben den Grundtexturkoordinaten auch eine Verschiebung (engl. offset) und eine Wiederholung (engl. repeat) der Textur auf dem Polygon ermöglichen. Wir bauen in unseren Editor ausschließlich *planares Textur-Mapping* ein. Das bedeutet, dass wir Texturen auf eine Ebene projizieren. Andere Möglichkeiten sind zum Beispiel sphärisches oder zylindrisches Textur-Mapping, wobei man eine Textur auf eine Kugel bzw. einen Zylinder projiziert.

Planares Textur-Mapping

Die Projektion der Textur auf eine Ebene ist trivial. Man ignoriert dabei die Koordinate, auf deren Ebene projiziert werden soll, und berechnet dann rein zweidimensional auf den anderen beiden Achsen, wo ein Vertex innerhalb der Bounding-Box prozentual gesehen auf diesen beiden Achsen liegt. Ein Vertex, der beispielsweise auf der X-Achse direkt im Minimum der Bounding-Box auf dieser Achse liegt, ist um 0% vom Minimum verschoben

Projektion der Textur auf eine Ebene

und erhält den Wert 0.0f für die u-Texturkoordinate. Ein Vertex, der in der Mitte der Box auf dieser Achse liegt, ist um 50% verschoben und erhält entsprechend den Wert 0.5f für die u-Koordinate usw. Die Wiederholung und Verschiebung der Textur wird initial auf keine Verschiebung und einfache Wiederholung gesetzt.



```
void CPolygon::CalcTextureCoords(ZFXAXIS axis,
                                const ZFXAabb *pAabb) {
    VERTEX *pVert=NULL;
    ZFXAabb aabb;

    if (pAabb) aabb = *pAabb;
    else aabb = m_Aabb;

    m_fTexOff[0] = m_fTexOff[1] = 0.0f;
    m_fTexRep[0] = m_fTexRep[1] = 1.0f;

    ZFXVector vcSize = aabb.vcMax - aabb.vcMin;

    for (UINT i=0; i<m_NumVerts; i++) {
        pVert = &m_pVerts[i];

        // planares Textur-Mapping
        if (axis == X_AXIS) {
            pVert->tu = (pVert->z - aabb.vcMin.z)/vcSize.z;
            pVert->tv = (pVert->y - aabb.vcMax.y)/vcSize.y;
        }
        else if (axis == Y_AXIS) {
            pVert->tu = (pVert->x - aabb.vcMin.x)/vcSize.x;
            pVert->tv = (pVert->z - aabb.vcMax.z)/vcSize.z;
        }
        else if (axis == Z_AXIS) {
            pVert->tu = (pVert->x - aabb.vcMin.x)/vcSize.x;
            pVert->tv = (pVert->y - aabb.vcMax.y)/vcSize.y;
        }
    } // for
} // CalcTextureCoords
```

*Eine Textur auf
mehrere Polygone
ziehen*

Auch bei dieser Methode geben wir die Bounding-Box als Parameter an, obwohl das Polygon doch seine eigene AABB kennen sollte. Das ist zwar auch so, aber wer sagt denn, dass wir die Bounding-Box des Polygons als Projektionsbereich nehmen wollen? Wenn man mehrere Polygone selektiert und deren Gesamt-AABB verwendet, so kann man eine Textur mit Hilfe dieser Funktion nicht einmal auf ein einzelnes Polygon *mappen*, sondern einmal auf alle ausgewählten Polygone, so dass jedes Polygon nur das entsprechende Stück der Textur anzeigt. Dies kann beim Texturieren des Levels sehr hilfreich sein, wenn man eine Textur nahtlos über mehrere Polygone ziehen will, ohne sie auf jedem Polygon einzeln mit Wiederholung und Off-set entsprechend auszurichten.

Hat man die Textur einmal auf ein Polygon gemappt, so ist man i.d.R. mit der Texturierung noch nicht fertig; sondern hier beginnt erst die eigentliche Arbeit. Nun gilt es, die Wiederholung der Textur einzustellen (falls sie mehrfach auf dem Polygon wiederholt werden soll) und die Verschiebung der Textur anzupassen, falls nötig. Auch hier haben wir natürlich die Formel parat, nach der wir die Wiederholung und Verschiebung in die Texturkoordinaten einrechnen:

```
u = u_base * u_repeat + u_offset;
v = v_base * v_repeat + v_offset;
```

Die eben berechneten Basis-Texturkoordinaten aus dem planaren Mapping werden mit dem Faktor der Wiederholung multipliziert und dann zu dem Wert der Verschiebung addiert. Das Problem dabei ist nur folgendes: Aus den so entstehenden Texturkoordinaten kann man nicht mehr extrahieren, wie viel an Wiederholung und Verschiebung in den Wert eingeflossen ist. Da wir aber die Textur in unserem Editor über eine Dialogbox editieren werden und das auch mehrfach ermöglichen wollen, ohne die Textur jedes Mal neu mappen zu müssen, brauchen wir die Werte für die aktuelle Wiederholung und Verschiebung in separaten Variablen.

Wenn nun der Level-Designer ein Polygon texturieren will, ruft er den entsprechenden Dialog auf. Dieser zeigt mittels der beiden oben erwähnten in der Klassendeklaration implementierten Methoden die entsprechenden Werte für Wiederholung und Verschiebung an. So kann der Level-Designer ausgehend von der aktuellen Verschiebung und Wiederholung diese Werte ändern. Die nun folgende Methode setzt die entsprechenden Änderungen um.

```
void CPolygon::TransTextureCoords(float ftU, float ftV,
                                  float frU, float frV)
{
    if (frU == 0.0f) frU = 0.00001f;
    if (frV == 0.0f) frV = 0.00001f;

    for (UINT i=0; i<m_NumVerts; i++) {
        // Undo alte Werte
        if (m_fTexOff[0] != 0.0f)
            m_pVerts[i].tu -= m_fTexOff[0];
        if (m_fTexOff[1] != 0.0f)
            m_pVerts[i].tv -= m_fTexOff[1];

        if ((m_fTexRep[0] != 1.0f) &&
            (m_fTexRep[0] != 0.0f) ) {
            m_pVerts[i].tu /= m_fTexRep[0];
        }
        if ((m_fTexRep[1] != 1.0f) &&
            (m_fTexRep[1] != 0.0f) ) {
```

*Transformation
von Textur-
koordinaten*

*Zurück an den
Anfang*



```

        m_pVerts[i].tv /= m_fTexRep[1];
    }

    m_pVerts[i].tu = m_pVerts[i].tu*frU + ftU;
    m_pVerts[i].tv = m_pVerts[i].tv*frV + ftV;
}

m_fTexOff[0] = ftU;
m_fTexOff[1] = ftV;
m_fTexRep[0] = frU;
m_fTexRep[1] = frV;
} // TransTextureCoors

```

**Zerstörung der
Koordinaten
verhindern**

Wir müssen zuerst die aktuell gültige Wiederholung und Verschiebung aus den Texturkoordinaten herausrechnen, Dazu wenden wir die oben gezeigten Formeln invers an, und dann wenden wir die Formel mit den neuen Werten auf die so zurückerhaltenen Basis-Texturkoordinaten aus dem ursprünglichen Mapping an. Einen Catch 22 müssen wir allerdings beachten: Wenn der Wert für die Wiederholung 0 ist, dann würden wir die Texturkoordinate vernichten und könnten nicht einfach wieder ohne Sicherheitsabfrage beim nächsten Rutsch durch sie teilen. Ich habe mich hier dafür entschieden, den Wert einfach nicht 0 werden zu lassen.

**Abfragen der
Texturtrans-
formation**

Zu guter Letzt schreiben wir für den komfortablen Zugriff noch eine Methode, mit der man auf einen Rutsch die Verschiebung und Wiederholung der Textur auf dem Polygon abfragen kann:



```

void CPolygon::GetTextureTrans(float *pftU, float *pftV,
                               float *pfrU, float *pfrV)
{
    if (pftU) *pftU = m_fTexOff[0];
    if (pftV) *pftV = m_fTexOff[1];
    if (pfrU) *pfrU = m_fTexRep[0];
    if (pfrV) *pfrV = m_fTexRep[1];
} // GetTextureTrans

```

MISC-Methoden für Polygone

**Bounding-Box
eines Polygons
berechnen**

Das Berechnen einer Bounding-Box ist nicht sonderlich schwer, wenn es sich um eine AABB handelt. Und genau eine solche achsen-ausgerichtete Bounding-Box brauchen wir, wie wir gesehen haben, an einigen Stellen für unser Polygon. Die folgende Methode läuft einfach durch alle Vertices des Polygons und sucht dabei nach den auftretenden Minimal- und Maximalwerten auf den drei Koordinaten, aus denen dann die Bounding-Box konstruiert wird.

```

void CPolygon::CalcBoundingBox(void) {
    ZFXVector vcMax, vcMin;

    // Zufallsstartwert aus dem Polygon
    vcMax.x = vcMin.x = m_pVerts[0].x;
    vcMax.y = vcMin.y = m_pVerts[0].y;
    vcMax.z = vcMin.z = m_pVerts[0].z;

    for (UINT i=0; i<m_NumVerts; i++) {
        if ( m_pVerts[i].x > vcMax.x )
            vcMax.x = m_pVerts[i].x;
        else if ( m_pVerts[i].x < vcMin.x )
            vcMin.x = m_pVerts[i].x;
        if ( m_pVerts[i].y > vcMax.y )
            vcMax.y = m_pVerts[i].y;
        else if ( m_pVerts[i].y < vcMin.y )
            vcMin.y = m_pVerts[i].y;
        if ( m_pVerts[i].z > vcMax.z )
            vcMax.z = m_pVerts[i].z;
        else if ( m_pVerts[i].z < vcMin.z )
            vcMin.z = m_pVerts[i].z;
    } // for

    m_Aabb.vcMax    = vcMax;
    m_Aabb.vcMin    = vcMin;
    m_Aabb.vcCenter = (vcMax + vcMin) / 2.0f;
} // CalcBoundingBox

```



Als Nächstes müssen wir den Vertices eines Polygons noch einen Normalenvektor für die Beleuchtung zuweisen. Der Einfachheit halber unterstützt unser Editor keine Transformation einzelner Vertex-Normalen. Stattdessen berechnen wir stur den Normalenvektor für das Polygon und setzen diesen für jeden Vertex ein:

*Normalen-
vektoren
berechnen*

```

void CPolygon::CalcNormals(void) {
    ZFXVector vc0, vc1, vc2, vcN;

    vc0.x = m_pVerts[m_pIndis[0]].x;
    vc0.y = m_pVerts[m_pIndis[0]].y;
    vc0.z = m_pVerts[m_pIndis[0]].z;

    vc1.x = m_pVerts[m_pIndis[1]].x;
    vc1.y = m_pVerts[m_pIndis[1]].y;
    vc1.z = m_pVerts[m_pIndis[1]].z;

    vc2.x = m_pVerts[m_pIndis[2]].x;
    vc2.y = m_pVerts[m_pIndis[2]].y;
    vc2.z = m_pVerts[m_pIndis[2]].z;

    vcN.Cross( (vc1-vc0), (vc2-vc0) );
}

```



```

vcN.Normalize();

for (UINT i=0; i<m_NumVerts; i++) {
    m_pVerts[i].vcN[0] = vcN.x;
    m_pVerts[i].vcN[1] = vcN.y;
    m_pVerts[i].vcN[2] = vcN.z;
}
} // CalcNormals

```

*Frontface und
Backface
vertauschen*

Beim Einfügen eines Polygons in einen Level kann es durchaus vorkommen, dass man die Vertices nicht im richtigen Uhrzeigersinn anlegt. Dann ist die sichtbare Front-Seite des Polygons nicht diejenige, die wir eigentlich sehen wollen. In solch einem Fall muss man in der Lage sein, das Frontface und das Backface des Polygons zu vertauschen, damit die andere Seite des Polygons sichtbar wird. Nötig ist das unter Umständen auch, wenn man eine Gruppe von Polygonen spiegelt. Wir nehmen hier einfach die Indexliste des Polygons und drehen sie um. Diese neu umsortierte Liste der Indices setzen wir für das Polygon fest, und dadurch werden insbesondere auch die Normalenvektoren neu berechnet. Pfannkuchen.



```

void CPolygon::InsideOut(void) {
    WORD *pNewIndis = new WORD[m_NumIndis];

    for (UINT i=0; i<m_NumIndis; i++)
        pNewIndis[i] = m_pIndis[m_NumIndis-1-i];

    SetIndices(pNewIndis, m_NumIndis);

    SAFE_DELETE_A(pNewIndis);
} // InsideOut

```

*Ein Polygon als
Teil eines Meshs
definieren*

Soll ich die Katze schon mal aus dem Sack lassen? Eigentlich ist es ja kein großes Geheimnis: Natürlich verwendet die Klasse CPolymesh, in der wir Mesh-Objekte als Menge von Polygonen beschreiben wollen, auch nur Instanzen der Klasse CPolygon, um ihre Geometrie zu definieren. Wenn ein Polygon Teil des Meshs sein soll, dann muss das Polygon einen Pointer auf das entsprechende Mesh erhalten. Genau dies können wir über die folgende Methode festlegen:



```

void CPolygon::SetAsPartOfMesh(CPolymesh *p) {
    if (p) {
        m_pDad = p;
        m_bPartOfMesh = true;
    }
    else {
        m_pDad = NULL;
        m_bPartOfMesh = false;
    }
} // SetAsPartOfMesh

```


Und nun noch ein Wort zu den Portalen: In unserem Editor erzeugen wir Portale, indem wir zuerst ein Polygon an der gewünschten Stelle in den gewünschten Abmessungen erzeugen. Dann nehmen wir das Polygon und erzeugen ein Portal daraus. Dazu dient die folgende Funktion; bzw. sorgt sie dafür, dass die geometrischen Daten aus dem Polygon in die Instanz der CPortal-Klasse transferiert werden. Der Editor wird dann dafür sorgen, dass das Polygon gelöscht wird und nun als Portal weiterlebt. Wichtig zu erwähnen ist hier noch, dass wir nur Portale zulassen, die viereckig sind.

```
bool CPolygon::CreatePortal(CPortal *pPortal) {
    if (m_NumVerts != 4) return false;

    pPortal->SetGeometry(m_pVerts, m_pIndis, m_Aabb);

    return true;
} // CreatePortal
```

*Ein Portal aus
einem Polygon
erzeugen*



Damit haben wir unsere CPolygon-Klasse erledigt – oder sie uns? Für alle, die den von oben herabprasselnden Polygonen entgangen sind, geht es jetzt mit dem Elternteil der Polygone weiter – mit den Polymeshs. Wir haben nun nämlich alles gesehen, was wir benötigen, um in unserem Editor Polygone und auch einzelne Vertices von Polygonen zu editieren.

*Das war's für die
Polygone*

Komplexe Modelle: CPolymesh

So oder so ähnlich lautet ein ziemlich dummer Spruch bei der Bundeswehr, der wohl noch auf Zeiten zurückzuführen ist, in denen die oberen Ränge Angst hatten, wenn sich das Fußvolk zusammenrottete. Wir wollen eine solche Zusammenrottung jedoch mit allen Mitteln unterstützen und bieten daher Polygonen die Gelegenheit, sich zu kleinen Grüppchen zusammenzuschließen. Eine Gruppe von Polygonen nennt man ein *Polymesh*. Ob es sich dabei um eine offene oder geschlossene Geometrie handelt, ist uns zunächst egal. Wir lassen sogar die Möglichkeit offen, dass ein einzelnes Polygon ein Polymesh ausmacht. Das ist zwar scheinbar sinnfrei, aber beim Editieren eines Levels kann es schon einmal vorkommen, dass wir so an einem Polymesh herumbasteln, dass zu einem diskreten Zeitpunkt nur ein einziges Polygon zu einem Mesh gehört. Sobald aber gar kein Polygon mehr zu einem Mesh gehört, wird es auf dem Level gelöscht, da es nun absolut keinen Sinn mehr erfüllt.

*Ein Mann ist
nichts, zwei Mann
sind eine Party,
und drei Mann
sind eine
Verschwörung*

Ein Polymesh ist also mehr oder weniger nur eine Organisationseinheit für Polygone, was in anderen Editoren auch oft als Gruppe bezeichnet wird. Sehr praktisch ist das, wenn man beispielsweise einen Schreibtisch mit allem Drum, Dran und Drauf konstruiert hat und diesen nun verrücken möchte. Man kann sich dann die Mühe machen, alle Polygone einzeln zu selektieren und zu verschieben – und zwar immer wieder, wenn man den Tisch an einer anderen Stelle haben möchte. Oder man macht sich einmal die Mühe und

*Organisation ist
alles*

gruppiert die Polygone zu einem Mesh. Durch einen Klick in unserem Editor können wir dann das Mesh als Ganzes auswählen und bequem verschieben, rotieren usw.



Zudem erfüllen Polymeshs noch einen ganz anderen Sinn in unserem späteren Game-Code. Natürlich haben wir auch Echtzeit-Schatten in unserem Spiel. Nun ist es aber wenig sinnvoll und technisch nicht performant umsetzbar, dass jedes Polygon einen Schatten wirft. Neben den animierten Charakteren definieren wir daher im Game-Code von Pandoras Legacy, dass Polymeshs Schatten werfen. Wenn wir also unseren Level fertig konstruiert haben, fassen wir alle Polygone als Polymeshs zusammen, die einen gemeinsamen Schatten erzeugen sollen, beispielsweise ein Schreibtisch, eine große Maschine oder eine Kiste.

Klassendeklaration

Lang, aber kurz!

Sprach ich nicht eben noch davon, dass die Klasse CPolygon das umfangreichste LOB für unseren Editor sein wird? Das ist auch richtig so, und tatsächlich sieht die Klasse CPolymesh auch sehr umfangreich aus. Teilweise täuscht der Eindruck aber, denn etliche Aufgaben wird ein Polymesh-Objekt einfach an die entsprechenden Instanzen der CPolygon-Klasse delegieren, d.h., es wird die Funktionsaufrufe an alle Polygone, aus denen es besteht, weiterleiten. Viele Methoden dieser Klasse sind also mit einer for-Schleife über alle Polygone und einen zugehörigen Methodenaufwurf erledigt. Hier ist erst mal unsere neue Klasse:



```
class CPolymesh : public CLevelObject {
public:
    CPolymesh(void);
    virtual ~CPolymesh(void);

    void CopyOf(CLevelObject*);
    void Render(ZFXVertexCacheManager*, bool);
    void Translate(const ZFXVector&);
    void InsideOut(void);
    void SetSkinID(UINT);
    void CountNonRedundantSkins(UINT*,UINT*);
    CPolygon* PickPoly(const ZFXRay&, float, float*);
    bool Picked(const ZFXRay&, float*);
    bool Intersects(const ZFXAabb&, ZFXAXIS);
    void GetIntersectingPolys(const ZFXAabb&,
                             CSelectionBuffer*, ZFXAXIS);
    void GetIntersectingVerts(const ZFXAabb&,
                              CSelectionBuffer*, ZFXAXIS);
    void CreateFromPolygons(CPolygon*, UCHAR);
    HRESULT RemovePolygon(const LPPOLYGON);
    HRESULT AddPolygon(const CPolygon &);
```

```

void      Save(FILE*);
bool      Load(FILE*);
void      SavePurePolys(FILE*);

void      SelectAllPolys(CSelectionBuffer*);
void      SelectInvert(CSelectionBuffer*);

void      HideNoPolys(void);
void      HideInvertPolys(void);
void      HideSelectedPolys(void);
void      HideUnselectedPolys(void);

LPPOLYGON GetPolygon(UINT i){return m_ppPoly[i];}
void      CalcBoundingBox(void);
void      CalcNormals(void);
void      CalcTextureCoords(ZFXAXIS, ZFXAabb*);

void      TransTextureCoords(float ftU,
                             float ftV, float frU, float frV);
bool      GetTextureTrans(float *pftU, float
                          *pftV, float *pfrU, float *pfrV);
private:
  LPPOLYGON *m_ppPoly;
  bool      m_bSupprBoxCalc;

  void      Init(void);
  void      Release(void);
}; // class
typedef class CPolymesh *LPPOLYMESH;

```

Die Konstruktoren und Destruktoren unserer Level-Objekte werde ich im Folgenden nicht mehr zeigen, denn sie sind alle identisch und sehen so aus, wie wir sie bei der Klasse CPolygon gesehen haben. Sie tun nicht viel mehr, als die jeweiligen Member-Methoden Init() und Release() aufzurufen. Welchen Sinn hat diese Rufumleitung? Nun, auf diese Weise kann man ein Objekt der Klasse herunterfahren und wieder neu initialisieren, ohne den zugehörigen Speicher zu zerstören und neu einrichten zu müssen.

*Konstruktor und
Destruktor*

Bei der Initialisierung passiert hier nicht viel. Alle Attribute erhalten sinnvolle Initialwerte, und der Typ des Objekts wird als LOB_MESH festgelegt. Interessant ist aber noch die Bedeutung der Variable m_bSupprBoxCalc. Sie dient dazu, eine Neuberechnung der Bounding-Box zu unterdrücken. Wir werden später eine Methode entwickeln, die die Bounding-Box für das Mesh neu berechnet, und diese aufrufen, wenn wir ein Polygon zu dem Mesh hinzufügen. Wenn wir aber eine Vielzahl an Polygonen gleichzeitig zu einem Mesh hinzufügen, so können wir über diese Variable steuern, dass die Bounding-Box eben nicht nach jedem neuen Polygon neu berechnet wird, sondern dass sie erst nach dem Hinzufügen des letzten Polygons neu erstellt wird.

*Initialisierung
eines Polymeshs*



```
void CPolymesh::Init(void) {
    m_NumPolys      = 0;
    m_NumVerts      = 0;
    m_NumIndis      = 0;
    m_ppPoly        = NULL;
    m_bSelected      = false;
    m_bHidden        = false;
    m_bSupprBoxCalc = false;
    m_Type           = LOB_MESH;
    memset(&m_Aabb, 0, sizeof(ZFXAabb));
} // Init
```

*Zurücksetzen
eines Polymeshs*

Bei dem Zurücksetzen einer Instanz der Klasse `CPolymesh` müssen wir nur schön ordentlich aufräumen und sämtlichen Speicher freigeben, den wir für unser Objekt potenziell allokiert haben. Wir wollen ja schließlich nicht, dass uns irgendwann mal der böse Speicherteufel heimsucht.



```
void CPolymesh::Release(void) {
    if (m_ppPoly) {

        for (UINT i=0; i<m_NumPolys; i++)
            SAFE_DELETE(m_ppPoly[i]);

        SAFE_FREE(m_ppPoly);
    }
} // Release
```

Eine Instanz erzeugen

*Von nichts kommt
nichts*

Um eine Instanz der Klasse `CPolymesh` zu erzeugen, benötigt man als Input eine Menge an Polygonen, also Instanzen der Klasse `CPolygon`, die zu dem Mesh gehören sollen. Natürlich ist das nicht ganz richtig, denn man kann ein Objekt dieser Klasse auch ohne Daten erzeugen. Aber das ist wenig sinnvoll, darum wird man i.d.R. die folgende Methode kurz nach dem Erstellen der Instanz aufrufen, um eine Menge an Polygonen zu dem Mesh zuzuordnen:



```
void CPolymesh::CreateFromPolygons(const CPolygon
                                   *pPolys, UCHAR n) {
    Release();
    Init();

    m_bSupprBoxCalc = true;

    for (UINT i=0; i<n; i++) AddPolygon(pPolys[i]);

    CalcBoundingBox();
    m_bSupprBoxCalc = false;
} // CreateFromPolygons
```

Wie ihr seht, stellen wir zunächst sicher, dass das Polymesh bisher noch nichts enthält. Sollte das aber der Fall sein, dann verwerfen wir die bisherigen Daten und initialisieren das Objekt neu. Nun unterdrücken wir die Neuberechnung der Bounding-Box, die, wie man unschwer vermuten kann, in der Methode `CPolymesh::AddPolygon` verwendet wird, wenn das entsprechende Unterdrückungs-Flag nicht gesetzt ist. In einer Schleife gehen wir dann die als Parameter übergebene Liste an Polygonen durch und fügen eines nach dem anderen zu unserem Objekt hinzu. Abschließend berechnen wir die Bounding-Box neu, da sich die Abmessungen unseres Polymeshs ja nun mit Sicherheit geändert haben.

Eins nach dem anderen

Die Methode `CPolymesh::AddPolygon` verwenden wir aber nicht nur dazu, um eine Gruppe von Polygonen zu einem Mesh zu gruppieren, sondern wir können auch jederzeit zusätzliche Polygone zu einem Mesh hinzufügen. Das ist ein Mindestmaß an Flexibilität, was wir uns hier gönnen wollen, auch wenn wir nur ein vergleichsweise kleines Tool schreiben. Nun gut, das Hinzufügen eines Polygons ist ein Keks. Wir verwenden hier die Methode `CLevelObject::CheckMem` der Basisklasse, um die aktuell verfügbare Speichermenge im Attribut `CPolymesh::m_ppPoly` zu überprüfen. Diese Methode allokiert uns ja Speicher für jeweils 50 Objekte in der angegebenen Größe, falls im aktuell allokierten Speicher kein Platz mehr ist. Man beachte, dass wir nicht mit einem einfachen Pointer arbeiten, sondern mit einem Doppelpointer, um das Attribut später als Array von Pointern verwenden zu können.

Hinzufügen einzelner Polygone

```
HRESULT CPolymesh::AddPolygon(const CPolygon &Poly) {
    // Speicher allokieren, falls nötig
    m_ppPoly = (LPPOLYGON*) CheckMem( (LPVOID*)m_ppPoly,
                                       m_NumPolys, sizeof(LPPOLYGON) );

    // Neue Instanz erzeugen und Daten kopieren
    m_ppPoly[m_NumPolys] = new CPolygon();
    m_ppPoly[m_NumPolys]->CopyOf( (LPLEVELOBJECT)&Poly );
    m_ppPoly[m_NumPolys]->SetAsPartOfMesh(this);

    // Zähler anpassen
    m_NumVerts += m_ppPoly[m_NumPolys]->GetNumVerts();
    m_NumIndis += m_ppPoly[m_NumPolys]->GetNumIndis();
    m_NumPolys++;

    if (m_bSupprBoxCalc) return S_OK;

    CalcBoundingBox();
    return S_OK;
} // AddPolygon
```



Wichtig ist zudem auch, dass wir die Zähler für Vertices und Indices des Meshs entsprechend erhöhen. Diese brauchen wir später für Statistiken, die wir im Level-Editor anzeigen lassen wollen. Die Berechnung der Bounding-Box könnte übrigens weitaus schneller sein. Die hier verwendete Methode durchläuft alle Polygone und berechnet die Box komplett neu. Schneller wäre es, wenn man einfach nur die Vertices des neuen Polygons durchginge und die Box bei Bedarf auf der jeweiligen Achse entsprechend erweitern würde. Allerdings ist der Level-Editor nicht performance-kritisch, und diese Operation ist es schon gar nicht, zumal wir ja durch das Unterdrückungs-Flag ein unnötig wiederholtes Berechnen der Box verhindern.

*Entfernen
einzelner
Polygone*

Was wir dem Polymesh eingebracht haben, das müssen wir natürlich ebenso smooth wieder auslöffeln können. Oder anders formuliert: Wo wir Polygone hinzufügen können, müssen wir sie auch wieder entfernen können. Für unseren Editor soll es ja auch kein Problem sein, wenn man später einzelne Polygone aus einem existierenden Mesh herauslöschen will, ohne das ganze Mesh zu zerstören und die Polygone wieder als freie Polygone im Level zu haben. Dafür verwenden wir die folgende Methode:



```
HRESULT CPolymesh::RemovePolygon(const LPPOLYGON pPoly)
{
    UINT i=0;

    // suche den Pointer, der gelöscht werden soll
    for (i=0; i<m_NumPolys; i++)
        if (m_ppPoly[i] == pPoly) break;

    if (i >= m_NumPolys) return ZFX_FAIL;

    // Zähler anpassen
    m_NumVerts -= m_ppPoly[i]->GetNumVerts();
    m_NumIndis -= m_ppPoly[i]->GetNumIndis();

    // lösche das Objekt, letztes umkopieren, wenn nötig
    SAFE_DELETE(m_ppPoly[i]);
    if (i != (m_NumPolys-1))
        m_ppPoly[i] = m_ppPoly[m_NumPolys-1];

    m_NumPolys--;

    CalcBoundingBox();
    return ZFX_OK;
} // RemovePolygon
```

Umkopiererei

Diese Methode übernimmt also die Speicheradresse eines zu löschenden Polygons. In dem Array von Pointern auf die Polygone des Meshs suchen wir das entsprechende Polygon heraus. Das setzen wir so um, weil man später durch das Selektieren eines Polygon beispielsweise dessen Adresse im

Selektions-Buffer `CSelectionBuffer` speichert und das zu löschende Objekt dann über diese Adresse aufspürt. Nun passen wir noch schnell die Zähler der Instanz an und geben dann den Speicher für das Objekt frei. Falls es sich dabei nicht um das als letztes in der Liste stehende Polygon gehandelt hat, haben wir nun ein Loch in der Liste von Polygonen des Meshs. Dementsprechend kopieren wir einfach den Pointer auf das letzte Polygon der Liste an die entsprechende Stelle um und haben wieder eine saubere lineare Liste. Höre ich da jemanden etwa »Verkettete Liste« oder »STL« sagen? Ich hoffe nicht, sonst erzähle ich gleich etwas von Kanonen und Spatzen bzw. verweise auf einen eigenen Memory-Manager.

Laden und speichern

Ist das langweilig! Natürlich ist das Laden und Speichern analog zu den Methoden in der Klasse `CPolygon` umgesetzt. Eine Instanz unserer Klasse `CPolymesh` muss natürlich alle relevanten Attribute in eine Datei pumpen, damit wir das Mesh später wieder komplett identisch reproduzieren können. Doch halt! Eine Besonderheit gibt es hier doch: Wir haben hier eine Liste von Polygonen. Und wir haben eine Klasse für Polygone, die ein Polygon auch laden und speichern kann. Was läge da näher, als diese Methoden auch einzusetzen? Da oben auf der Fahne, die über dem Objektorientierungsbunker flat-tert, steht schließlich in großen Lettern: »Jedes Objekt kümmert sich um sich selbst!!!«. So viel zum Thema: *Leave no man behind. Hoo-ah!*

*Im Westen nichts
Neues*

```
void CPolymesh::Save(FILE *pFile) {
    if (!pFile) return;

    fwrite(&m_NumPolys, sizeof(UINT), 1, pFile);
    fwrite(&m_Aabb, sizeof(ZFXAabb), 1, pFile);
    fflush(pFile);

    // Polygone nach Textur sortieren
    qsort(m_ppPoly, m_NumPolys, sizeof(LPPOLYGON),
        SortBySkin);

    for (UINT i=0; i<m_NumPolys; i++)
        m_ppPoly[i]->Save(pFile);
    } // Save
/*-----*/
```



```
bool CPolymesh::Load(FILE *pFile) {
    if (!pFile) return false;

    fread(&m_NumPolys, sizeof(UINT), 1, pFile);
    fread(&m_Aabb, sizeof(ZFXAabb), 1, pFile);

    if (m_NumPolys == 0) return false;
```

```

m_ppPoly = (LPPOLYGON*)InitMem( (LPVOID*)m_ppPoly,
                                m_NumPolys,sizeof(LPPOLYGON));

for (UINT i=0; i<m_NumPolys; i++) {
    m_ppPoly[i] = new CPolygon();

    if ( !m_ppPoly[i]->Load(pFile) ) {
        // kritischer Fehler
        delete m_ppPoly[i];
        m_NumPolys = i;
        return false;
    }

    m_ppPoly[i]->SetAsPartOfMesh(this);
}
return true;
} // Load

```

Speicher allokieren

Das Speichern ist natürlich der einfachere Fall. Wir müssen lediglich zwei Attribute eines Polymeshs speichern; den Rest erledigt die Klasse `CPolygon` dann schon selbst. Es sei noch erwähnt, dass wir die Polygone nach der Textur sortieren, die sie verwenden. Das macht uns die Sache beim Laden im Game-Code später einfacher, weil wir von vorsortierten Daten ausgehen können. Das Laden gestaltet sich dann fast ebenso einfach. Hier müssen wir nur darauf achten, dass wir auch entsprechend Speicher für die zu ladenden Daten allokieren.

Reine Polygone extrahieren

Zum Speichern brauchen wir auch noch eine andere Methode, nämlich eine, die einfach alle einzelnen Polygone eines Mesh-Objekts in eine Datei schreibt, ohne Informationen über das Mesh selbst zu beachten. Dazu muss ich ein klein wenig ausholen: Unser Editor soll ja laut unseren Spezifikationen auch PreFabs mit abspeichern können. Wenn wir ein Mesh selektiert haben und es als Prefab abspeichern wollen, dann pumpen wir einfach seine Polygone in die Datei für das Prefab, denn die Gruppierung eines Meshs ist für ein Prefab irrelevant. Es ist ja selbst eine Art von Gruppierung. Unsere Prefabs sind also nichts anderes als eine Liste von Polygonen.



```

void CPolymesh::SavePurePolys(FILE *pFile) {
    UINT n=0, SkinID=0;
    char buffer[2048];

    if (!pFile) return;
    for (UINT i=0; i<m_NumPolys; i++) {
        // Name der Basis-Textur
        SkinID = m_ppPoly[i]->GetSkinID();

        SendMessage(CApplication::m_G.hGUI[CSK_LIST],
                    CB_GETLBTEXT, SkinID,
                    (LPARAM)(LPCSTR)buffer);
    }
}

```



```

// speichere String-Länge und String in das File
n = strlen(buffer);
fwrite(&n, sizeof(UINT), 1, pFile);
fwrite(buffer, n, 1, pFile);

m_ppPoly[i]->Save(pFile);
}
} // Save

```

Nanu, wie wir sehen, speichern wir doch nicht einfach nur die Polygone stur weg. Wir fragen auch noch den String aus einem Combo-Box-Steuerelement ab und speichern diesen zusammen mit seiner Länge auch in die Datei. Zudem ist dort auch noch eine Klasse namens `CApplication` erwähnt? Na schön, immer der Reihe nach. Unser Editor ist natürlich eine WinAPI-Anwendung, auch wenn wir bisher nur am Klassen-Unterbau für die Arbeit an der Geometrie werkeln. Diese Anwendung basiert im Wesentlichen auf der Klasse `CApplication`, die alle Steuerelemente bietet und die entsprechenden Methoden der Klassen aufruft, an denen wir hier gerade arbeiten. Ein statisches Attribut dieser Klasse ist eine Struktur namens `m_G`, in der alle wichtigen Variablen gespeichert sind, so auch die Handles auf die Steuerelemente der Anwendung. So kommen wir also an besagte Combo-Box. Doch was enthält diese nun? Diese Combo-Box enthält eine Auflistung der Texturen, sortiert nach der SkinID. Wir schauen hier also nach, welchen Namen die Textur hat, die das Polygon verwendet, das wir in die Datei speichern wollen. Den Namen dieser Textur speichern wir dann mit in die Datei für das Prefab ab.

*WinAPI und
Combo-Box?*

Wir können leider nicht einfach die ID der Textur für das Polygon speichern, denn in diesem Fall würde ein Prefab nur genau dann mit richtigen Texturen angezeigt werden, wenn wir es in denselben Level laden. Würden wir das Prefab in einen anderen Level laden, so hätten die Polygone bestenfalls eine falsche Textur. Im schlimmsten Fall wäre die entsprechende ID für eine Textur noch gar nicht vergeben, weil der Level weniger Texturen verwendet. Wir müssen für Prefabs also tatsächlich den Namen der Textur speichern. Bei einem ganzen Level ist das nicht nötig, weil dieser selbst eine Liste von Texturnamen speichert, wie wir später sehen werden. Diese werden bei jedem Ladevorgang in identischer Reihenfolge geladen, so dass für die Level-Geometrie auch wirklich das Speichern der SkinID ausreicht.

*Prefabs und
Texturen*

Eine Instanz kopieren

Dasselbe Spielchen wie für das Laden und Speichern eines Polymeshs setzt sich auch beim Kopieren fort. Das Polymesh an sich hat kein Attribut, außer der Liste von Polygonen, die tatsächlich für das Kopieren relevant sind. Von daher durchlaufen wir einfach die Liste von Polygonen und fügen die Polygone des zu kopierenden Objekts der Instanz hinzu, die die Kopie des angegebenen Objekts werden soll.

*Es lebe die
Objekt-
orientierung!*



```
void CPolymesh::CopyOf(CLevelObject *pLob) {
    if ( !pLob->IsOfType(m_Type) ) return;

    LPPOLYMESH pMesh = (LPPOLYMESH)pLob;

    Reset();

    m_bSupprBoxCalc = true;

    for (UINT i=0; i<pMesh->m_NumPolys; i++)
        AddPolygon( *pMesh->m_ppPoly[i] );

    CalcBoundingBox();
    m_bSupprBoxCalc = false;
} // CopyOf
```

Kollisionsabfrage

*Markieren von
Polymeshs mit der
Maus*

Wir bieten natürlich auch die Möglichkeit, Polymesh-Objekte mit der Maus zu selektieren. Dies erfolgt analog zu der Methode der Klasse CPolygon. Wir geben eine Bounding-Box des Bereichs an, den der Level-Designer mit der Maus in einer der orthogonalen Ansichten markiert hat. Zusätzlich geben wir noch die Achse an, die ignoriert werden soll. Das ist die Achse, an der ein Betrachter in der Ansicht entlangblickt. Hier prüfen wir zunächst auf eine Überlappung der Bounding-Box des Meshs mit der angegebenen Box (die auf der entsprechenden Achse von ∞ bis $-\infty$ läuft). Haben wir hier eine Überlappung, dann reichen wir den Test an die einzelnen Polygone weiter. Sobald wir auch tatsächlich ein Polygon des Meshs in der Box haben, melden wir unser Okay zurück, dass dieses Polygon unter dem Selektionsrechteck liegt.



```
bool CPolymesh::Intersects(const ZFXAabb &aabb,
                           ZFXAXIS axis) {
    if ( IsHidden() ) return false;

    // Schnitt mit der Bounding-Box?
    if (!m_Aabb.Intersects(aabb)) return false;

    // Prüfe auf Schnitt mit einem Polygon im Mesh
    for (UINT i=0; i<m_NumPolys; i++) {
        if (m_ppPoly[i]->IsHidden()) continue;
        if (m_ppPoly[i]->Intersects(aabb, axis))
            return true;
    }
    return false;
} // Intersects
```

Nun kann es aber sein, dass der Level-Designer nicht ein ganzes Mesh auswählen möchte, sondern einzelne Polygone des Meshs. Auch das lassen wir natürlich zu, nett wie wir nun einmal sind. Die Strategie dabei ist folgende: Der Level-Designer zieht wieder mit der Maus ein Selektionsrechteck in der orthogonalen Ansicht auf. Wir erweitern dieses Rechteck dann zu einer Bounding-Box, indem wir die fehlende Koordinate jeweils von $+\infty$ bis $-\infty$ laufen lassen. Wenn ein Polymesh nun eine Überlappung mit dieser Box hat, dann testen wir seine nicht versteckten Polygone mit deren entsprechender Methode analog. Ist ein Polygon des Meshs ebenfalls mit mindestens einem Vertex in der Box enthalten, dann fügen wir dieses Polygon bzw. einen Pointer auf dieses Polygon dem Selektions-Buffer hinzu. Damit gilt das entsprechende Polygon als markiert. Das Polymesh als Ganzes aber ist davon herzlich unberührt.

Markieren einzelner Polygone oder Vertices in einem Mesh mit der Maus

Doch wir gehen noch einen Schritt weiter. Wir können nicht nur einzelne Polygone separat aus einem Mesh heraus selektieren, sondern auch einzelne Vertices. Die Strategie ist dabei identisch mit der eben beschriebenen, nur dass wir eben nicht ganze Polygone auswählen, sondern einzelne Vertices des Polygons, die zu dem Selektions-Buffer hinzugefügt werden, falls sie in der Box liegen. Diese Funktionalität setzen wir mit den folgenden beiden Methoden unserer Klasse CPolymesh um:

```
void CPolymesh::GetIntersectingPolys(const ZFXAabb
                                     &aabb, CSelectionBuffer *pSB,
                                     ZFXAXIS axis) {
    if ( IsHidden() ) return;

    // Schnitt mit der Mesh-Bounding-Box?
    if (!m_Aabb.Intersects(aabb)) return;

    // auch Schnitt mit einem Polygon?
    for (UINT i=0; i<m_NumPolys; i++) {
        if (m_ppPoly[i]->IsHidden()) continue;
        if (m_ppPoly[i]->Intersects(aabb, axis))
            pSB->AddLob( m_ppPoly[i] );
    }
} // GetIntersectingPolys
/*-----*/

void CPolymesh::GetIntersectingVerts(const ZFXAabb
                                     &aabb, CSelectionBuffer *pSB,
                                     ZFXAXIS axis) {
    if ( IsHidden() ) return;

    // sammle Vertices in der Schnittmenge ein
    for (UINT i=0; i<m_NumPolys; i++) {
```



```

        m_ppPoly[i]->GetIntersectingVerts(aabb, pSB,
                                           axis);
    }
} // GetIntersectingVerts

```

*Anklicken des
Polymeshs mit der
Maus*

So weit zur Selektion von Meshs, Polygonen und Vertices in den Meshs über die orthogonalen (zweidimensionalen) Ansichten. Doch natürlich möchten wir auch ganze Meshs oder einzelne Polygone aus einem Mesh in der perspektivischen Ansicht auswählen können. Auch dazu verwenden wir wieder die Maus, jedoch können wir hier eine Auswahl durch einen einfachen Klick auf ein Objekt treffen. Dieser Klick wird dann zu einem Strahl im 3D-Raum umgerechnet. Ob ein Polymesh angeklickt worden ist, ist relativ einfach herauszufinden. Wir testen dazu auf eine Kollision des Strahls mit der Bounding-Box des Meshs. Haben wir eine Kollision aufgespürt, dann testen wir, ob auch wirklich ein Polygon des Meshs von dem Strahl getroffen wurde und nicht nur leerer Raum. Über einen Referenz-Parameter der Methode geben wir dann die Entfernung zum getroffenen Punkt im Mesh zurück. Schließlich kann es ja auch sein, dass der Strahl zwei hintereinander liegende Meshs trifft. Dann müssen wir natürlich das näher am Betrachter liegende auswählen.

Dasselbe Problem müssen wir lösen, wenn wir einzelne Polygone aus dem Mesh auswählen wollen. Eines ist ja klar: Wenn der Anwender auf einen Punkt an seinem Bildschirm klickt, so ist dort genau ein Polygon. Wir müssen also auch hier, falls mehrere Polygone getroffen werden, nur dasjenige Polygon im Mesh auswählen, das dem Betrachter am nächsten liegt.

Mit den folgenden zwei Methoden können wir in der 3D-Ansicht Meshs bzw. Polygone aus Meshs auswählen.



```

bool CPolymesh::Picked(const ZFXRay &Ray, float *pFD) {
    if ( IsHidden() ) return false;

    if ( !m_Aabb.Intersects(Ray, NULL) ) return false;

    else if ( PickPoly(Ray,9999.9f,pFD) != NULL )
        return true;

    return false;
} // Picked
/*-----*/

```

```

CPolygon* CPolymesh::PickPoly(const ZFXRay &Ray,
                              float fMin, float *pFD) {
    CPolygon *pPoly = NULL;

    if ( IsHidden() ) return NULL;

```

```

for (UINT i=0; i<m_NumPolys; i++) {

    if ( m_ppPoly[i]->IsHidden() ) continue;

    // delegieren an Picked()
    if ( m_ppPoly[i]->Picked(Ray, pfD) ) {

        // Auswahl des am nächsten liegenden
        if (*pfD < fMin) {
            fMin = *pfD;
            pPoly = m_ppPoly[i];
        }
    }
}
return pPoly;
} // PickPoly

```

Eine Instanz rendern

Es ist mir ja fast peinlich, euch diese Methode zu zeigen, aber wieder einmal kümmern sich die Objekte um sich selbst. Wenn wir ein Polymesh rendern wollen, dann rendern wir einfach alle seine Polygone. Die einzige Ausnahme sind versteckte Polygone, also solche, die der Benutzer nicht mit angezeigt haben möchte. Die überspringen wir einfach, und damit sind sie nicht sichtbar. Hier erfolgt dieselbe Steuerung des Versteckens über den Parameter der Methode wie beim Rendern der Polygone. Dort hatte ich ja bereits erwähnt, dass man versteckte Polygone manchmal eben doch in der 3D-Ansicht sehen möchte. In diesem Fall übergehen wir das Flag über den Versteck-Status eines Objekts einfach grob.

```

void CPolymesh::Render(ZFXVertexCacheManager *pVCMan,
                      bool bHide) {
    if ( IsHidden() ) return;

    for (UINT i=0; i<m_NumPolys; i++) {
        // keine versteckten Polygone rendern
        if ( bHide && m_ppPoly[i]->IsHidden() ) continue;
        m_ppPoly[i]->Render(pVCMan, bHide);
    }
} // Render

```



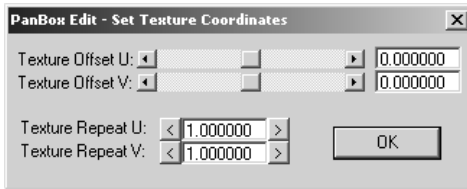
Texturkoordinaten für Polymeshs

Bei den Texturkoordinaten gibt es nichts Neues zu vermelden. Wenn wir ein planares Texturkoordinaten-Mapping durchführen möchten, delegieren wir den Job natürlich weiter an die Polygone, die dann die Drecksarbeit machen dürfen. Ebenso verfahren wir bei der Transformation von Texturkoordinaten. Ein bisschen Erklärungsbedarf gibt es aber vielleicht bei der Methode CPolymesh::GetTextureTrans. Diese benötigen wir, um die Texturtransforma-

*Und immer wieder
Mozarts Neunte*

tion des Meshs, also die Verschiebung und die Wiederholung in u - und in v -Richtung zu erhalten. Okay, wer hat da gerade gefragt, ob die verschiedenen Polygone des Meshs nicht auch verschiedene Werte für diese Variablen haben können? Sehr gut, setzen! Wenn wir also alle Polygone eines Meshs, sprich das gesamte Mesh, auswählen und seine Texturkoordinaten editieren wollen, so geht das nur, wenn alle Polygone dieselben Werte für Offset und Repeat haben. Anderenfalls könnten wir diese kaum über ein Menü unserer Applikation gleichzeitig editieren. Aber werfen wir dazu einen Blick auf Abbildung 14.5.

Abbildung 14.5:
Dialog zum Editieren von Textur-Verschiebung und -Wiederholung auf einem Level-Objekt



Tücken der Texturkoordinaten

In der Abbildung könnt ihr erkennen, dass die Verschiebung der Textur über zwei Schieberegler oder über eine direkte Eingabe für u und v eingestellt werden kann. Die Wiederholung kann man gleichfalls durch direkte Eingabe oder über entsprechende Buttons verändern. Das Problem dabei ist nun, dass die hier eingegebenen Werte für alle selektierten Polygone gelten. Wenn man beispielsweise ein gesamtes Polymesh auswählt und diesen Dialog aufruft, so kann er die aktuellen Einstellungen der Textur nur dann sinnvoll anzeigen, wenn sie für alle Polygone des Meshs identisch sind. Hat auch nur ein Polygon eine andere Verschiebung oder Wiederholung auf einer Achse, dann haben wir ein Problem. Ist dies der Fall, so werden alle Werte aller selektierten Polygone auf eine Wiederholung von 1 und eine Verschiebung von 0 gesetzt. Hier muss der Level-Designer ein wenig aufpassen. Natürlich könnten wir auch verbieten, dass dieser Dialog überhaupt verwendet werden kann, wenn man mehr als ein Polygon selektiert hat. Doch so erhalten wir uns die Flexibilität, eben doch mehrere Polygone gleichzeitig zu texturieren, falls alle dieselben Werte für Wiederholung und Verschiebung erhalten sollen.

Doch genug der Worte. Lassen wir den Code sprechen. Die Methode `CPolymesh::GetTextureTrans` gibt `true` als Rückgabewert zurück, wenn tatsächlich alle nicht versteckten Polygone im Mesh dieselben Werte bei den Texturtransformationen verwenden.



```
void CPolymesh::CalcTextureCoords(ZFXAXIS axis,
                                   const ZFXAabb *pAabb)
{
    if (!pAabb) pAabb = &m_Aabb;

    for (UINT i=0; i<m_NumPolys; i++) {
```

```

        m_ppPoly[i]->CalcTextureCoords(axis, pAabb);
    } // for
} // CalcTextureCoords
/*-----*/

void CPolymesh::TransTextureCoords(float ftU, float ftV,
                                   float frU, float frV)
{
    // Brute Force Verhinderung von Ofacher Wiederholung
    if ( (frU < 0.0001f) && (frU > -0.0001f))
        frU = 0.0001f;
    if ( (frV < 0.0001f) && (frV > -0.0001f))
        frV = 0.0001f;

    for (UINT i=0; i<m_NumPolys; i++)
        m_ppPoly[i]->TransTextureCoords(ftU,ftV,frU,frV);
} // TransTextureCoords
/*-----*/

bool CPolymesh::GetTextureTrans(float *pftU,
                                float *pftV, float *pfrU, float *pfrV)
{
    float ftU=0.0f, ftV=0.0f, frU=0.0f, frV=0.0f;

    for (UINT i=0; i<m_NumPolys; i++) {
        m_ppPoly[i]->GetTextureTrans(&ftU, &ftV,
                                     &frU, &frV);

        if (i>0) {
            if ( (*pftU != ftU) || (*pftV != ftV) ||
                (*pfrU != frU) || (*pfrV != frV) )
                return false;
        }
        *pftU = ftU; *pftV = ftV;
        *pfrU = frU; *pfrV = frV;
    }
    return true;
} // GetTextureTrans

```

Auswahl total und Versteckspielen

Wieder sind wir bei dem Auswählen von Objekten angekommen. Bisher kennen wir zwei Möglichkeiten, um bestimmte Objekte zu selektieren: Wir können in den orthogonalen (zweidimensionalen) Ansichten des Editors mit der Maus ein Selektionsrechteck aufziehen, und wir können Objekte in der perspektivischen (dreidimensionalen) Ansicht einfach anklicken. Doch es gibt noch zwei weitere Möglichkeiten: Zum einen können wir einfach alle Objekte eines bestimmten Typs im Level auswählen, beispielsweise alle Polygone. Zum anderen können wir eine bestehende Selektion invertieren.

Selektionen

Wir selektieren also alle Objekte, die bisher nicht ausgewählt sind, und deselektieren diejenigen, die gerade ausgewählt sind.

*Meshs sind etwas
Besonderes*

Das funktioniert normalerweise auf einer höheren Ebene in unserer Applikation, nämlich dort, wo wir die Listen von Polygonen, Polymeshs, Lichtern, Portalen usw. unseres Levels speichern. Doch die Klasse `CPolymesh` bildet eine einmalige Ausnahme in unserem Programm, wie ihr sicherlich schon hier und da bemerkt habt. Die Meshs verwenden ja selbst noch einen anderen LOB-Typ, nämlich das Polygon. Die Listen von Polygonen, die in einem Mesh stecken, sind also auf einer höheren Ebene dem Programm nicht bekannt. Folglich müssen wir auch für die beiden eben angesprochenen Selektionsverfahren eigene Methoden in die Klasse `CPolymesh` integrieren, um an die dortigen Polygone heranzukommen. Diese beiden Methoden sind die folgenden:



```
void CPolymesh::SelectAllPolys(CSelectionBuffer *pSB)
{
    for (UINT i=0; i<m_NumPolys; i++) {
        if ( !m_ppPoly[i]->IsHidden() )
            pSB->AddLob(m_ppPoly[i]);
    }
} // SelectAllPolys

/*-----*/

void CPolymesh::SelectInvert(CSelectionBuffer *pSB)
{
    for (UINT i=0; i<m_NumPolys; i++) {

        if ( m_ppPoly[i]->IsHidden() ) continue;

        // unselektierte Objekte auswählen
        if ( !m_ppPoly[i]->IsSelected() )
            pSB->AddLob(m_ppPoly[i]);
        // selektierte Objekte deselektieren
        else m_ppPoly[i]->DeSelect();
    }
} // SelectInvert
```

*Ab in den
Selektions-Buffer*

Auch hier verwenden wir wieder den Selektions-Buffer. Dieser ist normalerweise auf der eben angesprochenen höheren Ebene aktiv. Wenn wir beispielsweise ein einzelnes Polygon in einem Mesh auswählen, gibt das Mesh die Adresse des Polygons zurück, die dann auf der höheren Ebene in den Selektionsbuffer eingefügt wird. Wenn wir aber mehrere Polygone auswählen wollen, geben wir den Selektions-Buffer der Methode als Parameter mit, so dass die Methode die betroffenen Polygone gleich selbst in den Buffer schreiben kann. Hier ist auch schön zu sehen, dass als versteckt markierte Polygone vollkommen ignoriert werden.

Wenn hier immer von »der höheren Ebene« die Rede ist, so ist damit in der Regel die Klasse `CSector` gemeint, die die wichtigste Ordnungseinheit in unserem Editor und auch später im Spiel darstellen wird. In einem solchen Sektor werden jeweils die Listen von Polygonen, Meshs usw. gespeichert, die sich in dem Level befinden.



Da wir gerade beim Verstecken sind, machen wir dort auch weiter. Auch hier gibt es verschiedene Möglichkeiten, LOBs im Editor zu verstecken. Wenn ein LOB versteckt ist, bedeutet das nur, dass es für alle Operationen ignoriert wird. Es wird nicht gerendert, außer auf speziellen Wunsch in der perspektivischen Ansicht; es kann nicht direkt selektiert werden; es kann nicht gelöscht oder editiert werden. Auch das Verstecken von Objekten funktioniert auf einer höheren Ebene, doch wieder einmal gilt, dass die Polygone, die Teil eines Meshs sind, auf der höheren Ebene nicht bekannt sind. Darum muss die Klasse `CPolymesh` auch hier Methoden bieten, um das Versteckspielen umzusetzen. Im Folgenden realisieren wir vier Optionen, um den Versteckstatus der Polygone in einem Mesh zu steuern:

Versteckspiele

- ➔ verstecke die selektierten Polygone des Meshs
- ➔ verstecke die unselektierten Polygone des Meshs
- ➔ drehe den Versteckstatus der Polygone des Meshs um
- ➔ mache alle versteckten Polygone des Meshs sichtbar

Man beachte, dass es sich hierbei nicht um den Status des Meshs selbst handelt. Selbst wenn alle Polygone des Meshs versteckt sind, so ist das Meshs selbst eigentlich nicht versteckt, weil die Klasse `CPolymesh` ja ein eigenes Flag für den Versteckstatus hat. Diesen kleinen Haken müssen wir später beachten, damit wir beispielsweise keine Meshs selektieren können, die zwar nicht versteckt sind, aber dennoch eigentlich nicht sichtbar sind, weil alle ihre Polygone versteckt sind. Andersherum können wir aber auch das Mesh an sich selektieren und dann verstecken, ohne alle einzelnen Polygone verstecken zu müssen. Die hier gezeigten Funktionen dienen hauptsächlich dazu, einzelne Polygone in den Mesh-Objekten anzusteuern.

Auch alle Polygone des Meshs sind nicht das Mesh selbst

```
void CPolymesh::HideNoPolys(void) {
    for (UINT i=0; i<m_NumPolys; i++)
        m_ppPoly[i]->UnHide();
    } // HideNoPolys
/*-----*/
```



```
void CPolymesh::HideSelectedPolys(void) {
    for (UINT i=0; i<m_NumPolys; i++)
        if ( m_ppPoly[i]->IsSelected() )
            m_ppPoly[i]->Hide();
    }
```

```

    } // HideSelectedPolys
/*-----*/

void CPolymesh::HideUnselectedPolys(void) {
    if ( IsSelected() ) return;

    for (UINT i=0; i<m_NumPolys; i++)
        if ( !m_ppPoly[i]->IsSelected() )
            m_ppPoly[i]->Hide();
    } // HideUnselectedPolys
/*-----*/

void CPolymesh::HideInvertPolys(void) {
    for (UINT i=0; i<m_NumPolys; i++)
        if ( m_ppPoly[i]->IsHidden() )
            m_ppPoly[i]->UnHide();
        else
            m_ppPoly[i]->Hide();
    } // HideInvertPolys

```

*Alles eine Frage
der Flags*

Wie ihr hier schon sehen könnt, ist das Selektieren und Verstecken der LOBs *plain simple* implementiert. Wir steuern den jeweiligen Zustand einfach über die Flags der Basisklasse und werten sie bei bestimmten Aktionen (rendern, selektieren, ...) aus. Das ist keine Magie. Solange wir ein vernünftiges Interface haben, über das wir auf die LOBs zugreifen können, also in unserem Fall eine WinAPI-Oberfläche, ist das Editieren von LOBs ein Klacks.

MISC-Methoden für Polymeshs

Dreizeiler

Natürlich hat unsere Klasse CPolymesh noch die eine oder andere Hilfsmethode, die wir hier schnell besprechen. Die folgenden drei Methoden tun nichts weiter, als durch alle Polygone des Meshs zu laufen und die entsprechenden Aufrufe an jedes Polygon einmal zu delegieren:

```

➡ CPolymesh::InsideOut
➡ CPolymesh::SetSkinID
➡ CPolymesh::CalcNormals

```

*Bounding-Box
berechnen*

Die Berechnung der Bounding-Box eines Meshs ist natürlich ebenso simpel wie bei einem Polygon. Nur haben wir in einem Mesh nicht direkt eine Menge von Punkten, aus denen wir die Minimal- und Maximalwerte heraussuchen müssen. Aber wir haben einzelne Polygone, die wiederum eine korrekte Bounding-Box haben, die dann jeweils schon eigene Minimal- und Maximalwerte hat. Aus den einzelnen Bounding-Boxen der Polygone suchen wir also die absoluten Werte heraus und verwenden diese für die Bounding-Box des Meshs.

```

void CPolymesh::CalcBoundingBox(void) {
    ZFXVector vcMax, vcMin, vcTemp;
    ZFXAabb Aabb;

    // zufällige Startwerte
    Aabb = m_ppPoly[0]->GetAabb();
    vcMax = vcMin = Aabb.vcCenter;

    for (UINT i=0; i<m_NumPolys; i++) {
        Aabb = m_ppPoly[i]->GetAabb();

        vcTemp = Aabb.vcMax;
        if ( vcTemp.x > vcMax.x ) vcMax.x = vcTemp.x;
        else if ( vcTemp.x < vcMin.x ) vcMin.x = vcTemp.x;
        if ( vcTemp.y > vcMax.y ) vcMax.y = vcTemp.y;
        else if ( vcTemp.y < vcMin.y ) vcMin.y = vcTemp.y;
        if ( vcTemp.z > vcMax.z ) vcMax.z = vcTemp.z;
        else if ( vcTemp.z < vcMin.z ) vcMin.z = vcTemp.z;

        vcTemp = Aabb.vcMin;
        if ( vcTemp.x > vcMax.x ) vcMax.x = vcTemp.x;
        else if ( vcTemp.x < vcMin.x ) vcMin.x = vcTemp.x;
        if ( vcTemp.y > vcMax.y ) vcMax.y = vcTemp.y;
        else if ( vcTemp.y < vcMin.y ) vcMin.y = vcTemp.y;
        if ( vcTemp.z > vcMax.z ) vcMax.z = vcTemp.z;
        else if ( vcTemp.z < vcMin.z ) vcMin.z = vcTemp.z;
    }
    m_Aabb.vcCenter = (vcMax + vcMin) / 2.0f;
    m_Aabb.vcMax = vcMax;
    m_Aabb.vcMin = vcMin;
} // CalcBoundingBox

```



Eines hätten wir da noch zu klären: In der Deklaration der Klasse haben wir ja schon die folgende Methode gesehen:

*Adresse eines
bestimmten
Polygons*

```

LPPOLYGON CPolymesh::GetPolygon(UINT i)
{ return m_ppPoly[i]; }

```

Wir können uns also hier einen Pointer auf ein bestimmtes Polygon aus dem Mesh geben lassen. Da stellt sich doch die Frage, wie wir an die Nummer des Polygons kommen, das uns interessiert, und wozu wir das brauchen? Diese Methode wird im Editor nur an einer Stelle verwendet. Wir können ein Mesh ja auch fragmentieren; das bedeutet, wir zerstören die Instanz der Klasse CPolymesh, wollen aber die Polygone behalten und als ungruppierte, freie Polygone im Level weiterverwenden. Dazu müssen wir alle Polygone zuerst aus dem Mesh kopieren. Dann lassen wir einfach eine Schleife über alle Polygone im Mesh laufen und kopieren sie eines nach dem anderen über die entsprechenden Pointer als freie Polygone in den Level.

Aufbruch in eine neue Welt: CPortal

Sinn und Zweck von Portalen

Im Kapitel über das Scene-Management haben wir ja schon viel über die Theorie von Portalen und Portal-Engines gehört. Man verwendet sie, um Übergänge zwischen verschiedenen Teilen eines Levels zu realisieren und eine Art Graphenstruktur im Level aufzubauen. Im einfachsten Fall ist ein Raum in einem Indoor-Level ein so genannter **Sektor** (bei *Doom 3* auch »Area« genannt), und alle Öffnungen des Raum hin zu anderen Sektoren sind die Stellen, an denen die Portale dieses Sektors sind. Solche Öffnungen sind natürlich Türen, Fenster und Löcher in Wänden. Aber wer sagt denn, dass es hier immer nach den bisher bekannten physikalischen Gesetzen zugehen muss? Eine solche Öffnung kann auch ein Stargate oder ein Zauberspiegel sein – das ist ja grad das Schöne an Portalen. Sobald ein Objekt das Portal durchschreitet, wird es in den angrenzenden Sektor versetzt. Ob dieser im Level dann auch wirklich physikalisch hinter dem Portal liegt oder an einer ganz anderen Stelle im Level, ist vollkommen egal.

Funktion von Portalen in unserem Editor

Natürlich kann ein Portal nicht direkt zu einem anderen Sektor führen. Ein Portal führt immer nur zu einem anderen Portal, das dann wiederum zu einem anderen Sektor gehört. In unserem Editor werden die Portale aber keine Funktion erfüllen. Das bedeutet, sie werden nicht aktiv sein. Es geht hier nur darum, die Portale zu setzen, damit sie später im Game-Code für ein Spiel, das unsere Level verwendet, benutzt werden können. Wir werden auch eines nicht in den Editor implementieren, und das ist das manuelle Verbinden von Portalen.

Mehr über PanBox-Portale

Wie – keine Verbindung?

Da mag man sich fragen, welchen Sinn Portale erfüllen, wenn wir sie nicht verbinden? Moment, davon war nicht die Rede. Ich sagte nur: Wir werden sie im Editor nicht *manuell* verbinden. Warum sollten wir eine Arbeit machen, die man auch automatisieren kann? Wie wir eben noch mal gehört haben, führt ein Portal immer zu einem anderen Portal. Wenn man eine strikte Portal-Engine hat, dann müssten zwei Portale, die jeweils zueinander führen, eine identische Form und Größe haben. Wir sind jedoch mit unseren Definitionen etwas legerer.



Als kleiner Zwischeneinwurf sei erlaubt, dass ein Portal natürlich auch wieder zu einem anderen Portal im selben Sektor führen kann. Man geht also links aus einem Raum heraus und kommt rechts wieder herein. Das kann ein lustiges Spielchen für eine mythische Pforte sein, bei der man den richtigen Eingang finden muss, um weiterzukommen. Ein anderes lustiges Portal-spielchen ist, wenn das Portal B, über das man zum Portal A gelangt, eben nicht wieder zurück zum Portal A verweist, sondern auf das Portal C. So kann man eine knifflige Labyrinth-Aufgabe konstruieren.

Was wir tun werden, ist Folgendes: Wir erstellen in unserem Editor schlicht und einfach Portale als spezielle Polygone, und das war's. Intern verbindet der Editor automatisch die Portale, und zwar immer dann, wenn ein neues Portal im Level erzeugt wird. Das funktioniert ganz einfach so, dass zwei Portale, deren Bounding-Boxen sich irgendwie überlappen, miteinander verbunden werden. Damit verlieren wir viel der Flexibilität von Portalen, weil wir damit keine Portale verbinden können, die eben nicht physikalisch zusammen liegen. Dadurch sparen wir uns aber im Editor die Arbeit, zwei Portale von Hand miteinander zu verbinden. Ich denke mal, das Interface ist für den Anfang schon komplex genug.

Die beliebige Verbindung zwischen zwei Portalen habe ich im Editor schon vorbereitet, so dass Portale eindeutige IDs erhalten. Normalerweise benötigen unsere Polygone, Meshs, Lichter usw. keine eindeutigen IDs, denn sie müssen sich nicht innerhalb einer Level-Datei gegenseitig referenzieren. Ein Polymesh besteht zwar aus einer Menge von Polygonen, diese werden aber direkt zusammen mit dem Mesh linear abgespeichert. Bei Portalen ist das nicht möglich. Ein Portal A könnte ja auch auf ein Portal B verweisen, das in einem Sektor am anderen Ende des Levels liegt. Portale sind also die einzigen LOBs, die eine ID brauchen. Das ist im Editor bereits integriert, ebenso wie die Möglichkeit, ein Zielportal in einem Portal zu speichern. Ihr müsst also nur das Interface so erweitern, dass man zu einem selektierten Portal ein beliebiges anderes Portal angeben kann, das dann mit diesem Portal verbunden wird.

*Es ist aber alles
vorbereitet*

Natürlich kann man auch die Funktionalität zur automatisierten Verbindung sich berührender Portale, die wir nachher in der Klasse `CSector` sehen werden, mit anderen Verbindungsarten im Editor kombinieren. Man könnte einzelne Portale, die noch nicht verbunden sind, von Hand verbinden. Dabei sollte man auch eine Fallunterscheidung zwischen gegenseitigen Verbindungen und einseitigen Verbindungen treffen. Eine gegenseitige Verbindung ist gegeben, wenn man für Portal A als Zielportal das Portal B angibt und für das Portal B dann auch automatisch das Portal A als Ziel gesetzt wird. Eine einseitige Verbindung wäre es, wenn man dem Portal B in diesem Fall noch kein Ziel zuweist, sondern das Ziel später beliebig wählen kann, beispielsweise ein ganz anderes Portal C. Dazu ist es extrem hilfreich, wenn man beim Rendern der Portale im Editor auch als Beschriftung eines Portals seine ID und die ID des gesetzten Ziels angibt. Aber das sind Implementierungsdetails, die ich euch als Hausaufgabe überlasse.

Weitere Optionen

Klassendeklaration

Eines sollte ich noch vorweg schicken: Wie wir schon gelernt haben, drückt es sehr auf die Performance, wenn man Portale verwendet, die sehr viele Kanten haben, weil man dann beim Erzeugen eines Frustums mit einem Portal ein so genanntes *n-sided* Frustum erhält, das *n* Seitenebenen hat. Jede dieser Ebenen verlangt eine Culling-Berechnung. Logischerweise wächst der

*Portale werden
aus Polygonen
geboren*

Rechenaufwand umso mehr, je größer n ist, bis man den Punkt erreicht, an dem der Rechenaufwand für das Culling größer ist, als wenn man die Geometrie einfach ohne Culling stur herunterrendern würde. Deswegen führen wir hier die Restriktion ein, dass ein Portal in unseren Leveln immer genau vier Vertices hat – nicht einen mehr, aber auch keinen weniger. Ein solches Portal ist also auf alle Fälle ein Viereck und resultiert daher in einem »normalen« Frustum mit vier Seitenebenen wie das View-Frustum auch eines ist. Das Erzeugen eines Portals wird in unserem Editor auch nicht direkt möglich sein, denn warum sollten wir den Aufwand betreiben? Wir werden einfach ein ganz normales Polygon der Level-Geometrie erzeugen, und wenn es ein Rechteck ist, kann man es zu jedem beliebigen Zeitpunkt in ein Portal verwandeln. Die entsprechende Methode `CPolygon::CreatePortal` haben wir ja schon gesehen. Sie dient nur dazu, die Vertices und Indices des Polygons in ein angegebenes Portal zu kopieren. Die höhere Ebene, also die Klasse `CSector` hat die Aufgabe, ein Portal-Objekt zu erzeugen, diese Methode aufzurufen und dann das Polygon zu zerstören. So einfach machen wir das.

Aber nun endlich zurück zum Code. Hier ist die Deklaration der Klasse für unsere LOBs vom Typ Portal:



```
class CPortal : public CLevelObject {
public:
    CPortal(void);
    virtual ~CPortal(void);

    void CopyOf(CLevelObject*);
    void SetGeometry(const VERTEX*, const WORD*,
                    const ZFXAabb&);
    void Render(ZFXVertexCacheManager*, bool);
    bool Picked(const ZFXRay&, float*);
    bool Intersects(const ZFXAabb&, ZFXAXIS);

    // Verbindungs-Kram
    UINT GetID(void) {return m_ID; }
    UINT GetItsTargetID(void){return m_BID; }
    CSector* GetItsSector(void) {return m_pDad; }
    CPortal* GetItsTarget(void) {return m_pBrother; }
    void SetID(UINT ID) { m_ID = ID; }
    void SetItsSector(CSector *pS) { m_pDad=pS; }
    void SetItsTarget(CPortal *pP);
    void DisconnectFrom(UINT id)
        { if (m_BID==id) m_BID=0; }
    void Save(FILE*);
    bool Load(FILE*);

private:
    CSector* m_pDad; // Eigener Sektor
    CPortal* m_pBrother; // Zielportal
}
```

```

UINT      m_ID;           // Eigene ID
UINT      m_BID;          // Bruder-ID
VERTEX    *m_pVerts;      // Vertices
WORD      *m_pIndis;      // Indices
WORD      *m_pLineIndis;  // Line-Indices
bool      m_bVisible;     // Zielsektor sichtbar

void      Init(void);
void      Release(void);
}; // class
typedef class CPortal *LPportal;

```

Als Erstes fällt auf, dass wir hier nicht den Begriff »Zielportal« verwenden, sondern das Zielportal einfach als »Bruder« bezeichnen. In dieser Klasse ist sonst insgesamt nicht viel los. Das Portal kennt seinen Bruder, falls es einen hat, es weiß, zu welchem Sektor es gehört, und es hat Geometrie, mit der wir es im Editor darstellen können. Ansonsten kommt noch der übliche Kram aus der Basisklasse hinzu, also die Implementierung der dort abstrakten Methoden und natürlich die Methoden für die Selektion eines Portals.

*Auch du, mein
Bruder Brutus*

Es sei nochmals darauf verwiesen, dass wir die Portale nur im Editor anzeigen, weil man sie dort generieren und setzen muss. Im endgültigen Spiel werden die Portale natürlich nicht angezeigt.



Eine Instanz erzeugen

Auch hier betreiben wir dasselbe Spielchen wie bei allen anderen LOBs: Der Konstruktor und der Destruktor dieser Klasse rufen einfach die Methoden CPortal::Init und CPortal::Release auf. Das haben wir ja bei den anderen LOBs schon gesehen. In diesen Methoden passiert ebenso nichts Dramatisches außer dem Initialisieren der Attribute auf sinnvolle Startwerte und dem Festlegen des LOB-Typs bzw. das Freigeben des dynamisch belegten Speichers.

*Konstruktor und
Destruktor und
das alte LOB-Spiel*

```

void CPortal::Init(void) {
    m_NumVerts    = 0;
    m_NumIndis    = 0;
    m_NumPolys    = 1;
    m_ID          = 0;
    m_BID         = 0;
    m_bSelected   = false;
    m_bHidden     = false;
    m_bVisible    = true;
    m_pVerts      = NULL;
    m_pIndis      = NULL;
    m_pLineIndis  = NULL;
    m_pDad        = NULL;
    m_pBrother    = NULL;
    m_Type        = LOB_PORTAL;
}

```



```

        memset(&m_Aabb, 0, sizeof(ZFXAabb));
    } // Init
/*-----*/

void CPortal::Release(void) {
    SAFE_DELETE_A(m_pVerts);
    SAFE_DELETE_A(m_pIndis);
    SAFE_DELETE_A(m_pLineIndis);
} // Release

```

Festlegen der Geometrie

Das Festlegen der Geometrie ist die einzige Möglichkeit, eine Instanz der Klasse CPortal mit entsprechenden Werten zu betanken. Das machen wir über das Kopieren der Daten aus einem Polygon. Es sei hier noch erwähnt, dass wir in unserem Editor ein Portal, wenn wir ein Polygon erst einmal in ein Portal verwandelt haben, nicht mehr verschieben oder rotieren oder gar seine Geometrie ändern können. Schließlich würde es später wenig Sinn machen, die Portale noch zu verändern, wenn man im Editor schon Verbindungen zwischen den Portalen festgelegt hat. Ein Portal kann man in unserem Editor also nur selektieren, um es zu verstecken oder um es zu löschen. Das sind die einzigen beiden Editier-Optionen.



```

void CPortal::SetGeometry(const VERTEX *pVerts,
                          const WORD *pIndis,
                          const ZFXAabb &aabb) {

    Reset();

    m_NumVerts = 4;
    m_NumIndis = 6;

    m_pVerts = new VERTEX[m_NumVerts];
    m_pIndis = new WORD[m_NumIndis];
    m_pLineIndis = new WORD[m_NumVerts*2];

    if (!m_pVerts || !m_pIndis || !m_pLineIndis) return;

    memcpy(m_pVerts, pVerts, sizeof(VERTEX)*4);
    memcpy(m_pIndis, pIndis, sizeof(WORD)*6);

    for (UINT i=0; i<m_NumVerts; i++) {
        m_pLineIndis[(i*2)] = i;
        m_pLineIndis[(i*2)+1] = i+1;
        if (i==m_NumVerts-1) m_pLineIndis[(i*2)+1] = 0;
    }
    memcpy(&m_Aabb, &aabb, sizeof(ZFXAabb));
} // SetGeometry

```


Laden und speichern

Das Laden und Speichern erfolgt hier so wie bei allen anderen LOBs. Wir schreiben einfach linear alle relevanten Daten einer Instanz der Klasse CPortal in eine offene Datei, zuerst die simplen Attribute und danach die dynamisch allokierten Listen. Beim Laden machen wir dasselbe Spielchen rückwärts, wir müssen dort nur auch den entsprechenden Speicher über die Methoden der Basisklasse allokiieren.

Was soll man da noch sagen?

```
void CPortal::Save(FILE *pFile) {
    fwrite(&m_ID,      sizeof(UINT),    1, pFile);
    fwrite(&m_BID,      sizeof(UINT),    1, pFile);
    fwrite(&m_NumVerts, sizeof(UINT),    1, pFile);
    fwrite(&m_NumIndis, sizeof(UINT),    1, pFile);
    fwrite(&m_bVisible, sizeof(bool),    1, pFile);
    fwrite(&m_Aabb,      sizeof(ZFXAabb), 1, pFile);

    fwrite(m_pVerts, sizeof(VERTEX), m_NumVerts, pFile);
    fwrite(m_pIndis, sizeof(WORD), m_NumIndis, pFile);
    fwrite(m_pLineIndis, sizeof(WORD), m_NumVerts*2, pFile);
} // Save

/*-----*/
```

```
bool CPortal::Load(FILE *pFile) {
    fread(&m_ID,      sizeof(UINT),    1, pFile);
    fread(&m_BID,      sizeof(UINT),    1, pFile);
    fread(&m_NumVerts, sizeof(UINT),    1, pFile);
    fread(&m_NumIndis, sizeof(UINT),    1, pFile);
    fread(&m_bVisible, sizeof(bool),    1, pFile);
    fread(&m_Aabb,      sizeof(ZFXAabb), 1, pFile);

    if (m_NumVerts == 0) return false;

    m_pVerts = new VERTEX[m_NumVerts];
    m_pIndis = new WORD[m_NumIndis];
    m_pLineIndis = new WORD[m_NumVerts*2];
    if ( !m_pVerts || !m_pIndis || !m_pLineIndis)
        return false;

    fread(m_pVerts, sizeof(VERTEX), m_NumVerts, pFile);
    fread(m_pIndis, sizeof(WORD), m_NumIndis, pFile);
    fread(m_pLineIndis, sizeof(WORD), m_NumVerts*2, pFile);
    return true;
} // Load
```



Eine Instanz kopieren

Auch das Kopieren einer Instanz dieser Klasse bringt uns keine neuen Erkenntnisse. Auch hier werden die wichtigsten Attribute kopiert, und fertig ist die Wurst.



```
void CPortal::CopyOf(CLevelObject *pLob) {
    if ( !pLob->IsOfType(m_Type) ) return;
    LPPORTAL pPort = (LPPORTAL)pLob;
    Reset();

    SetGeometry(pPort->m_pVerts, pPort->m_pIndis,
                pPort->m_Aabb);

    m_pDad      = pPort->m_pDad;
    m_pBrother  = pPort->m_pBrother;
    m_ID        = pPort->m_ID;
    m_BID       = pPort->m_BID;
    m_bVisible  = pPort->m_bVisible;
} // CopyOf
```

Kollisionsabfrage und Rendern einer Instanz

*Ein Portal ist ein
Polygon ist ein
Polygon ist ein
Polygon*

Ich könnte jetzt die ganzen Methoden für das Selektieren oder das Picken eines Portals in den orthogonalen und der perspektivischen Ansicht abdrucken, aber damit würde ich wertvolle Seiten mit redundantem Code verschwenden. Wir wissen ja, dass wir ein Portal aus einem Polygon erstellen. Und ein Portal ist auch nichts anderes als ein Spezialfall eines Polygons. Die Kollisionsabfragen mit einem Portal entsprechen also denen der Klasse CPolygon.

*Verbesserungs-
vorschlag*

Dasselbe gilt für das Rendern eines Portals, das ebenso funktioniert wie das Rendern eines Polygons. Im Nachhinein betrachtet, hätte es also Sinn gemacht, die Klasse CPortal nicht direkt von CLevelObject abzuleiten, sondern direkt von CPolygon und die Methoden zum Rendern und für die Kollisionsabfragen nicht mehr zu überladen. Aber hinterher ist man ja immer schlauer.

MISC-Methoden für Portale

C'mon brother

Eine Sache müssen wir hier noch sauber vorbereiten, auch wenn wir Portale in dieser Version des Editors nicht miteinander verbinden werden. Aber wenn man diese Funktion später integrieren möchte, kann es ja nicht schaden, das hier schon einmal korrekt vorzubereiten. Wenn man also eine Instanz der Klasse CPortal als Bruder einer anderen Instanz dieser Klasse angeben möchte, dann muss man auf einen Fallstrick Acht geben: Da wir immer mit einem DAU als Anwender rechnen und stets nach der Devise *better safe than sorry* handeln wollen, verhindern wir, dass man ein Portal als seinen eigenen Bruder angeben kann. Das ist so ziemlich das Einzige, was

man mit einem Portal nicht machen sollte, weil das je nach Implementierung der Portal-Engine zu unendlichen Schleifen führen kann.

```
void CPortal::SetItsTarget(CPortal *pP) {
    // unendliche Loops verhindern
    if (pP->GetID() == m_ID) return;

    if (pP) {
        m_pBrother = pP;
        m_BID = m_pBrother->GetID();
    }
    else {
        m_pBrother = NULL;
        m_BID = 0;
    }
} // SetItsTarget
```



Es kann auch nötig sein, die Verbindung eines Portals aufheben zu können, wenn man beispielsweise ein Portal doch mit einem anderen verbinden möchte oder wenn der Anwender den Bruder einfach löscht. Die obige Methode akzeptiert hier auch einen NULL-Pointer für diesen Fall.

*Verbindung
aufheben*

Es werde Licht: CLight

Kommen wir nun zu der Klasse `CLight`, über die wir die Beleuchtung eines Levels regeln. Wir verwenden hier der Einfachheit halber ausschließlich Omni-Lights, also Punktlichtquellen. Dabei hat die Klasse `CLight` erst mal nichts damit zu tun, wie wir die Beleuchtung beim Rendern regeln. Das Einzige, was wir dann von den Instanzen brauchen, ist die Position der Lichtquelle, ihr Radius und die Farbe des Lichts. Dementsprechend ist unsere Klasse für Lichter weitaus unspektakulärer, als ihr euch das eventuell vorgestellt habt, und wir können sie entsprechend zügig abhandeln. Die Deklaration der Klasse ist weitaus länger als die Implementierung der Methoden.

*Lichter in
unserem Editor*

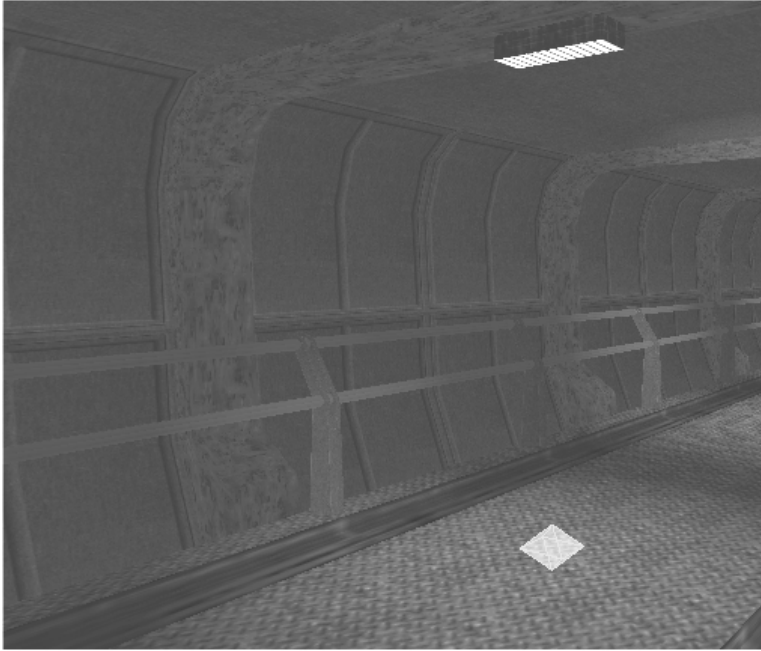
Weil wir hier nur eine Art von Lichtquellen verwenden, haben wir aber ein kleines Problem. Ein Omni-Light strahlt in alle Richtungen gleichmäßig viel Licht aus. Solche Lichtquellen sind aber in der Realität sehr selten. Oder besser gesagt: Das ist in der Realität fast die einzige Art von Lichtquelle, die es wirklich gibt (von Laserpointern mal abgesehen) denn jegliche Lichtquellen sind meistens so abgeschirmt, dass sie ihr Licht in Form eines Spotlights aussenden, also nur in einem bestimmten Kegel. Beispielsweise sorgt ein Lampenschirm oder die lichtundurchlässige Fassung einer Glühlampe dafür, dass das Licht nicht gleichmäßig in alle Richtungen strahlen kann. Spotlights sind also in unserer realen Welt weitaus häufiger anzutreffen. Doch Omni-Lights sind generell leichter zu berechnen, und daher verwenden wir sie. Das ist auch nicht so schlimm, wie es sich anhört, denn man kann ein Spotlight wunderbar mit einem Omni-Light simulieren. Das zeigt Abbil-

*Lichtquellen !
= Schattenwerfer*

dung 14.5. Wir modellieren einfach die Lampe dort, wo sie sein soll, beispielsweise direkt an der Decke. Die im Spiel später unsichtbare Lichtquelle, also die Position von der aus wir das Licht berechnen, setzen wir aber an die Stelle, wo der Lichtkegel auf eine Wand oder auf den Boden auftreffen soll. So wirkt die Lampe wie ein echtes Spotlight.

Abbildung 14.6:

Faken eines Spotlights durch ein Omni-Light. Die eigentliche Lampe hängt oben an der Decke, doch die Lichtquelle ist am Boden, wo der Lichtkegel auftreffen soll.



Shadow Caster

Das Problem dabei ist aber, dass wir – wenn wir von der Position der Lichtquelle ausgehen würden – natürlich die Schatten falsch berechnen würden. Daher werden wir bei den Objekten unserer Klasse `CLight` über ein Flag unterscheiden, ob sie wirklich eine Lichtquelle oder ein *Shadow Caster* sind. Die Lichtquellen werden dann auch wirklich zur Lichtberechnung verwendet und können, wie oben gezeigt, auch zum Faken von Spotlights verwendet werden. Die als Shadow Caster gekennzeichneten Instanzen werden hingegen als diejenigen Punkte verwendet, von denen das Licht rein physikalisch korrekt betrachtet ausgehen müsste. In Abbildung 14.6 wäre das also ein Punkt direkt an der Lampe an der Decke. Von dort aus wird kein Licht abgestrahlt, sondern von dieser Position aus werden dann Echtzeit-Schatten für die entsprechenden Objekte berechnet. Mit dieser simplen Lösung können wir viele Probleme auf einmal beseitigen.

Klassendeklaration

Geometrische Repräsentation der Lichter

Nachdem die Berechnung der Lichter also geklärt ist, wenden wir uns der Klassendeklaration der Lichter zu. Ähnlich wie bei den Portalen gilt, dass diese Objekte selbst später im Spiel nicht angezeigt werden. Aber im Editor

müssen wir sie schon sehen, damit wir sie auswählen und editieren können. Wie bereits in Abbildung 14.6 gezeigt, verwenden wir dazu einen Quader, der auf einer Spitze steht. Ich habe dem Editor auch die Klasse `CGeometry` spendiert, die über statische Methoden eine Menge an Vertices und Indices erzeugt, die eine geometrische Form ergeben. Dort sind zwei verschiedene Formen implementiert, nämlich besagter Quader und ein Würfel. Diese werden wahlweise als Liste von Polygonen oder als Mesh zurückgegeben.

Entsprechend hat unsere Klasse `CLight` ein Attribut der Klasse `CPolymesh`, in dem wir die Geometrie für das Anzeigen der Lichtquelle im Editor speichern. Die anderen Methoden sollten nun selbsterklärend sein.

```
class CLight : public CLevelObject {
public:
    CLight(void);
    virtual ~CLight(void);

    void        CopyOf(CLevelObject*);

    void        Render(ZFXVertexCacheManager*,bool);
    void        Translate(const ZFXVector&);

    bool        Picked(const ZFXRay &Ray, float *pFD)
                { return m_Mesh.Picked(Ray, pFD); }

    bool        Intersects(const ZFXAabb &b, ZFXAXIS a)
                { return m_Mesh.Intersects(b, a); }

    ZFXVector   GetPosition(void) { return m_vcPos; }
    void        SetPosition(const ZFXVector&);

    void        SetColor(const ZFXCOLOR &clr)
                { memcpy(&m_Clr,&clr,sizeof(ZFXCOLOR));}

    void        SetColor(float fR, float fG, float fB)
                { m_Clr.fR = fR; m_Clr.fB = fB;
                  m_Clr.fG = fG; m_Clr.fA = 1.0f; }

    ZFXCOLOR    GetColor(void) { return m_Clr; }

    void        SetRadius(float fR) { m_fR = fR; }
    float       GetRadius(void)     { return m_fR; }
    void        SetFlickering(bool b) {m_bFlicker = b;}
    bool        IsFlickering(void) {return m_bFlicker;}
    void        SetShadowCasting(bool b);
    bool        IsShadowCasting(void){return m_bShadow;}
    void        Save(FILE*);
    bool        Load(FILE*);
```



```

private:
    CPolymesh m_Mesh;
    ZFXVector m_vcPos;
    ZFXCOLOR m_Clr;
    float m_fR;
    bool m_bFlicker;
    bool m_bShadow;

    void Init(void);
    void Release(void);

}; // class
typedef class CLight *LPLIGHT;

```

Init() Da wir hier keinen dynamischen Speicher verwenden, kann die Methode `CLight::Release` leer bleiben. Bei der Initialisierung ist vielleicht das Erzeugen der Geometrie noch bemerkenswert: Im zweiten Parameter der Methode `CGeometry::CreateLight` hätte man auch alternativ oder zusätzlich zum ersten Parameter eine Liste von Polygonen angeben können. Der dritte Parameter ist die Position, an der die Geometrie erzeugt werden soll, und die folgenden drei Parameter geben die gewünschte Größe auf der jeweiligen Achse an.



```

void CLight::Init(void) {
    CGeometry geo;
    m_NumVerts = 0;
    m_NumIndis = 0;
    m_NumPolys = 1;
    m_vcPos = ZFXVector(0,0,0);
    m_bSelected = false;
    m_bHidden = false;
    m_bFlicker = false;
    m_bShadow = false;
    m_Type = LOB_LIGHT;
    memset(&m_Aabb, 0, sizeof(ZFXAabb));

    geo.CreateLight(&m_Mesh, NULL,
                   ZFXVector(0,0,0),
                   0.2f, 0.2f, 0.2f);

    m_Mesh.SetSkinID(1);
} // Init

```

Flackernde Lichter

Es gibt hier auch das Flag `m_bFlicker`, über das man einstellen kann, ob ein Licht flackern soll oder nicht. Selbst bei *Doom 1* gab es schon flackernde Lichter, die für eine gruselige Atmosphäre sorgten. Hierbei handelt es sich wirklich nur um ein Flag, das zu interpretieren wir der Applikation überlassen, die unsere Level-Daten lädt und interaktiv anzeigt.

Eine Instanz laden, speichern und kopieren

Das Kopieren ist ebenso wie das Laden und Speichern einer Instanz der Klasse `CLight` ein alter Hut. Wir pumpen immer noch die wichtigen Attribute in eine Datei bzw. kopieren diese. Ebenso wie eine Instanz `CPolymesh` die Methoden der Klasse `CPolygon` benutzt, verwendet `CLight` die Methoden seines Attributs `CPolymesh`, damit dieses sich um sich selbst kümmert.

*Immer noch ein
alter Hut*

```
void CLight::CopyOf(CLevelObject *pLob) {
    if ( !pLob->IsOfType(m_Type) ) return;
    LPLIGHT pLight = (LPLIGHT)pLob;
    Reset();
    m_Aabb      = pLob->GetAabb();
    m_vcPos     = pLight->m_vcPos;
    m_Clr       = pLight->m_Clr;
    m_fR        = pLight->m_fR;
    m_bFlicker  = pLight->m_bFlicker;
    m_bShadow   = pLight->m_bShadow;
    m_Mesh.CopyOf( &pLight->m_Mesh );
} // CopyOf

/*-----*/

void CLight::Save(FILE *pFile) {
    fwrite(&m_vcPos,    sizeof(ZFXVector), 1, pFile);
    fwrite(&m_Aabb,     sizeof(ZFXAabb),   1, pFile);
    fwrite(&m_Clr,      sizeof(ZFXCOLOR),  1, pFile);
    fwrite(&m_fR,       sizeof(float),     1, pFile);
    fwrite(&m_bFlicker, sizeof(bool),      1, pFile);
    fwrite(&m_bShadow,  sizeof(bool),      1, pFile);
    m_Mesh.Save(pFile);
} // Save

/*-----*/

bool CLight::Load(FILE *pFile) {
    fread(&m_vcPos,    sizeof(ZFXVector), 1, pFile);
    fread(&m_Aabb,     sizeof(ZFXAabb),   1, pFile);
    fread(&m_Clr,      sizeof(ZFXCOLOR),  1, pFile);
    fread(&m_fR,       sizeof(float),     1, pFile);
    fread(&m_bFlicker, sizeof(bool),      1, pFile);
    fread(&m_bShadow,  sizeof(bool),      1, pFile);
    return m_Mesh.Load(pFile);
} // Load
```



Kollisionsabfrage und Rendern einer Instanz

Es gibt hier nichts zu der Kollisionsabfrage für die Selektion eines Lichtobjekts und über das Rendern zu sagen. Beide Methoden werden einfach an das Attribut der Klasse `CPolymesh` weitergeleitet.

Interaktive Objekte: CEntity

Animation über Entities

Eine *Entity* (dt. Einheit, Rechtspersönlichkeit) ist in unserem Kontext ein geometrisches Gebilde wie ein Mesh, das aber nicht statisch ist, sondern dem Animationen und Bewegungen zugewiesen werden können, und die der Spieler in der interaktiven Darstellung unseres Levels auch selbst aktivieren kann. Solche Entities kann man beliebig komplex gestalten, von einfachen Schaltern, die der Spieler umlegt, über Fahrstühle und sich bewegende Plattformen bis hin zu ganzen Zügen, die kontinuierlich oder auf Knopfdruck eine Reihe von Stationen abfahren. Das Spiel *Duke Nukem 3D* hatte damals schon ein sehr ausgeklügeltes System, um solche Züge über beliebig viele Wegpunkte im Level fahren zu lassen.

Türen

Ganz so weit gehen wir hier nicht. Wir werden nur eine Art von Entity implementieren, und zwar Türen. Diesen kann man eine der drei Weltachsen als Bewegungsachse zuordnen, zusammen mit der Länge der Strecke, die sie sich bewegen sollen, und einer Zeit in Sekunden, die sie nach dem Zurücklegen der Strecken pausieren sollen, bevor sie wieder dieselbe Strecke zurückgleiten. Angestoßen wird diese einfache Animationssequenz später im Spiel dadurch, dass der Spieler diese Türen »aktiviert«, also beispielsweise eine bestimmte Taste in der Nähe der Tür drückt. Man könnte die Animation aber natürlich auch automatisch anfahren, wenn der Spieler direkt vor der Tür steht.

Komplexere Objekte ergänzen

Ihr werdet gleich sehen, wie man so eine einfache Animationssequenz für eine Entity definieren kann, und danach sollte es euch nicht weiter schwerfallen, auch Entities in den Editor zu integrieren, die komplexere Bewegungspfade zulassen. Dazu könnte man, wie bereits angedeutet, eine Klasse für Wegpunkte erzeugen. Einer Entity kann man dann eine Liste von Wegpunkten zuweisen, die sie abzufahren hat.

Klassendeklaration

Definitionen

Bevor wir uns an die Klasse für die Entities wagen, müssen wir noch ein paar Dinge definieren. Eine Instanz dieser Klasse wird eine ganz allgemeine Entity ohne spezielle Fähigkeiten sein. Je nachdem, wie weit man seine Entities noch erweitern möchte, kann es hier Sinn machen, verschiedene Arten von Entities als eigene Ableitungen dieser Klasse zu realisieren. Wir werden das hier aber nicht tun, sondern verschiedene Entity-Typen definieren. Für jeden eigenen Typ würde man dann eine Struktur einführen, die die spezifischen Daten enthält. Das sieht für unsere einzige Entity-Art, also eine Tür, wie folgt aus:


```
typedef enum ZFXENTITY_TYPE {
    ENT_UNKNOWN,
    ENT_DOOR
} ZFXENTITY;

typedef struct ENTITYDOOR_TYPE {
    float    fDist;
    float    fPause;
    ZFXAXIS Axis;
    UINT     Portal[2];
} ENTITYDOOR;
```



Kommen wir nun zu unserer Klasse für Entities. Ein Entity-Objekt hat natürlich auch immer Geometrie, die im Level angezeigt werden muss. Wir setzen das hier so um, dass wir ein Attribut vom Typ `CPolymesh` in die Klasse einfügen, das die Geometrie beinhaltet. Dazu hat diese Klasse nur noch eines zu bieten, was wir bisher noch nicht gesehen haben, und zwar einen `void-Pointer`, in dem wir jeweils die entity-spezifische Struktur, hier also immer `ENTITYDOOR`, abspeichern.

CEntity

```
class CEntity : public CLevelObject {
public:
    CEntity(void);
    virtual ~CEntity(void);

    void        CopyOf(CLevelObject*);
    void        Render(ZFXVertexCacheManager*,bool);
    void        Translate(const ZFXVector&);
    bool        Picked(const ZFXRay&, float*);
    bool        Intersects(const ZFXAabb&, ZFXAXIS);
    void        Save(FILE*);
    bool        Load(FILE*);

    ZFXENTITY    GetSubType(void) { return m_SubType; }
    void        SetSubType(ZFXENTITY);
    void        SetGeometry(const CPolymesh *pMesh);
    void        SetData(const ENTITYDOOR&);
    void        SetData(const ENTITYPLAYER&);
    void*       GetData(void) { return m_pData; }

    void        RemovePortalConnections(UINT);
    bool        ConnectTo( CPortal* );

private:
    ZFXENTITY    m_SubType;    // Tür, ...
    CPolymesh    m_Mesh;      // Polymesh-Objekt
    void        *m_pData;     // Info Struktur
    UINT        m_nSize;      // Größe von pData
```



```

        void      Init(void);
        void      Release(void);
    }; // class
typedef class CEntity *LPENTITY;

```

Eine Instanz erzeugen

*Erst war das
Mesh, dann die
Entity*

Warum sollten wir uns alles komplizierter machen, als es nötig ist? Portale haben wir ja ganz einfach dadurch erzeugt, dass wir ein Polygon über die bereits im Editor vorhandene Funktionalität zum Erstellen von Polygonen konstruiert haben. Dieses Polygon haben wir dann selektiert und konnten es per Knopfdruck in ein Portal verwandeln. Für die Entities machen wir das genauso genial einfach. Allerdings gehen wir hier nicht von einem restringsierten Polygon aus, sondern von einem beliebigen Mesh. Der Level-Designer modelliert also ganz normal eine Tür aus einer beliebigen Menge an Polygonen. Diese gruppiert er dann zu einem Mesh, selektiert das Mesh und drückt auf einen Knopf, der das Mesh in eine Entity vom Typ *Door* verwandelt.

Polygone können wir bereits aus Vertex- und Indexlisten erzeugen und zu einem Mesh gruppieren, das wir selektieren. Es ist kein Problem, dieses Mesh nun per Mausklick im Interface des Editors zu löschen und vorher ein CEntity-Objekt mit einer Kopie des Meshs zu erzeugen.

Die üblichen Funktionalitäten

*Delegation an das
Mesh-Objekt*

Auch an dieser Stelle gibt es zu den üblichen Funktionalitäten (wie zum Picking, zur Selektion, zum Rendern usw.) nichts zu sagen, denn dafür verwenden die Methoden dieser Klasse natürlich ihre Instanz der Klasse CPolymesh. Also brauchen wir uns hier nicht weiter darum zu kümmern. Mit solchen Aspekten im Hinterkopf haben wir ja extra die Klasse CPolymesh eingeführt. Bei komplexeren Entities könnte man sich auch überlegen, stattdessen eine Liste von Entities in der Klasse CEntity zu verwenden, so dass man z.B. die einzelnen Wagons einer Entity der Art *Subway Train* als einzelne Meshs betrachten könnte.

MISC-Methoden für Entities

*Entity-Typ
festlegen*

Auch wenn wir hier nur einen Typ von Entity zulassen, nämlich den Typ *Tür*, so werden wir ihn trotzdem nicht durch Hardcoding in den Quelltext integrieren. Die Methode CEntity::SetSubType dient dazu, den Typ des Entity-Objekts festzulegen. Bisher reagiert diese Methode zwar nur auf Türen, aber man kann die Methode beliebig erweitern. Abbildung 14.7 zeigt einen Screenshot des Dialogs, den man erhält, wenn man ein Mesh im Editor als Entity definiert. Wie ihr seht, steht bisher nur eine gleitende Tür zur Auswahl. Aber andere Optionen sind im Editor bereits vorgesehen.

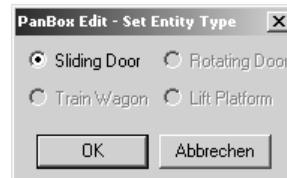


Abbildung 14.7:
Auswahl des Entity-Typs

Und hier ist die versprochene Methode, die intern auch durch den in Abbildung 14.7 gezeigten Dialog verwendet wird, um den dort vom Level-Designer ausgewählten Typ festzulegen.

```
void CEntity::SetSubType(ZFXENTITY SubType) {
    if (SubType == m_SubType) return;

    m_SubType = SubType;

    SAFE_FREE(m_pData);

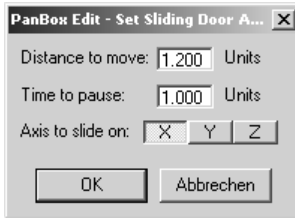
    switch (m_SubType) {
        case ENT_DOOR:
            m_nSize = sizeof(ENTITYDOOR);
            m_pData = malloc(m_nSize);
            if (!m_pData) break;
            memset(m_pData, 0, m_nSize);
            break;
        default: break;
    } // switch
} // SetSubType
```



Ebenfalls über einen Dialog kann man dann die Eigenschaften einer Tür festlegen. Dazu muss man das entsprechende Entity-Objekt selektieren und einen Knopf im User-Interface des Editors anklicken. Dann erscheint der in Abbildung 14.8 gezeigte Dialog. Dort kann man die Distanz eingeben, über die das Objekt in der interaktiven Darstellung bei einer Aktivierung bewegt werden soll, ebenso wie die Zeit in Sekunden, die die Tür im offenen Zustand verharren soll, bevor sie sich automatisch wieder schließt. Als Drittes kann man noch eine der drei Weltachsen auswählen, auf denen sich die Tür bewegen soll. In der Regel ist das die horizontale X- oder Z-Achse, aber natürlich für *spacige* Türen auch die Y-Achse. Denkbar ist auch hier eine Erweiterung, und zwar auf eine beliebige Achse. Dazu müsste man im Dialog auch einen Normalenvektor definieren können, der die Bewegungsrichtung angibt. Noch besser wäre natürlich eine grafische Umsetzung in der man mit der Maus einen entsprechenden Vektor ziehen kann. Es sei noch erwähnt, dass die Richtung der Bewegung auf der Achse über das Vorzeichen bei der Distanz gesteuert werden kann. Ein positiver Wert bewirkt eine Bewegung entlang der positiven Richtung der Achse um umgekehrt.

*Eigenschaften
einer Tür*

Abbildung 14.8:
Eigenschaften einer
Tür-Entity einstellen



Entities mit Portalen verbinden

*There's a new
Door in Town*

Nun hält noch eine schöne Eigenschaft Einzug in unseren Editor. Portale kann man natürlich auch aktivieren bzw. deaktivieren. Man erinnere sich nur daran, wie heftig die Jungs vom SG1-Team immer an diesem *Stargate* drehen müssen, damit sich dort etwas tut. Nun überlegen wir mal weiter: Über Portale bestimmen wir, welche Sektoren wir rendern müssen. Je weniger wir rendern müssen, umso besser ist das für die Performance. Hinzu kommt, dass Portale normalerweise in entsprechenden Öffnungen eines Sektors sitzen, die zumeist Übergänge in Form von Türen sein werden. Aha, jetzt kommen wir der Sache schon näher, nicht wahr? Wenn also in so einem Übergang bei dem Portal eine Tür ist und diese Tür auch noch geschlossen ist, dann ist auch das Portal verdeckt. Logischerweise brauchen wir den dahinter liegenden Sektor nicht zu rendern, selbst wenn das Portal im View-Frustum enthalten ist.

*Geschlossene
Türen schließen
Portale*

Unsere Absicht dürfte nun klar sein. Wenn eine Entity eine Tür ist und diese Tür sich mit einem Portal überlappt, dann brauchen wir das Portal nur in Betracht zu ziehen, wenn die Tür nicht geschlossen ist. Anderenfalls könnten wir so tun, als gäbe es an dieser Stelle gar kein Portal. Ebenso wie bei den Portalen werden wir auch diese Funktionalität automatisiert durchführen, und Türen auf Portale verweisen lassen, die in ihrem Türrahmen sitzen. Jedes Mal, wenn man im Editor ein Mesh als Tür-Entity definiert, prüft der Editor selbstständig alle Portale im Level und speichert in der Instanz der CEntity-Klasse diejenigen Portale ab, die sich mit der Tür überlappen. Das können maximal zwei sein, auch wenn der Level-Designer mehr Portale auf dieselbe Stelle setzen kann, was aber keinen Sinn machen würde. Die folgenden beiden Methoden dienen dem Editor dazu, diese Funktionalität zu realisieren:



```
void CEntity::RemovePortalConnections(UINT id) {
    if (m_SubType != ENT_DOOR) return;

    ENTITYDOOR *pData=NULL;

    pData = (ENTITYDOOR*)m_pData;
    if (pData->Portal[0] == id) pData->Portal[0]=0;
    if (pData->Portal[1] == id) pData->Portal[1]=0;
} // RemovePortalConnections
```

```

/*-----*/

bool CEntity::ConnectTo(CPortal *pPortal) {
    if (m_SubType != ENT_DOOR) return false;

    ENTITYDOOR *pData=NULL;

    pData = (ENTITYDOOR*)m_pData;
    if (pData->Portal[0] != 0) {
        if (pData->Portal[1] == 0) {
            pData->Portal[1] = pPortal->GetID();
            return true;
        }
        else return false;
    }
    else {
        pData->Portal[0] = pPortal->GetID();
        return true;
    }
} // RemovePortalConnections

```

Auch diese automatisierte Steuerung bei dem Einfügen eines Portals oder einer Tür-Entity in den Level wird durch den Editor über die Klasse `CSector` umgesetzt. Doch dazu kommen wir im folgenden Abschnitt. Hier wurde nur gezeigt, wie man auf den unteren Ebenen der LOBs die Portale über ihre im Level eindeutige ID miteinander oder mit einer Tür-Entity verknüpfen kann.

Die höhere Ebene

Ich mach den Fisch: CSpawnPoint

Ganz fix klären wir hier noch Mini-LOB ab: Unserem Level fehlt bisher die Möglichkeit, Punkte zu definieren, an denen Characters, also Spieler oder computergesteuerte Bots, ihr Spiel im Level beginnen. Solche Punkte nennt man für gewöhnlich *Spawn Points* (dt. Laichplätze), und man sagt, dass Objekte dort *respawnen*, wenn sie im Level auftauchen. Die folgende Klasse zeige ich nur mit ihrer Deklaration, da sie so kurz ist, wie sie nur sein kann. Ein Spawn Point braucht nur ein einziges Attribut, und das ist die Position im 3D-Raum, an der der Punkt liegt. Um das Objekt auch rendern zu können, fügen wir ihm noch eine Instanz der Klasse `CPolymesh` hinzu. Dort verwenden wir dasselbe Objekt, mit dem wir auch Lichtquellen im Level visualisiert haben. Natürlich rendern wir es in einer anderen Farbe, um es von Instanzen der Klasse `CLight` unterscheiden zu können. Die Klassendeklaration sieht wie folgt aus:

*Laichplätze für
Characters*



```

class CSpawnPoint : public CLevelObject {
public:
    CSpawnPoint(void);
    virtual ~CSpawnPoint(void);

    void        CopyOf(CLevelObject*);
    void        Render(ZFXVertexCacheManager*,bool);
    void        Translate(const ZFXVector&);

    bool        Picked(const ZFXRay &Ray, float *pfD)
                { return m_Mesh.Picked(Ray, pfD); }
    bool        Intersects(const ZFXAabb &b, ZFXAXIS a)
                { return m_Mesh.Intersects(b, a); }

    ZFXVector    GetPosition(void) { return m_vcPos; }
    void        SetPosition(const ZFXVector&);
    void        Save(FILE*);
    bool        Load(FILE*);

private:
    CPolymesh m_Mesh;
    ZFXVector m_vcPos;

    void        Init(void);
    void        Release(void);
}; // class
typedef class CSpawnPoint *LPSPAWNPOINT;

```

*Alles bleibt am
Mesh hängen*

Initial hat ein Spawn Point die Position (0,0,0) im Level, also am Weltursprung. Über die Methode `CSpawnPoint::Translate` kann man diesen Punkt verschieben bzw. ihn über die entsprechenden Methoden abfragen und direkt neu setzen. Alle anderen Methoden dieser Klasse wie beispielsweise das Rendern und die Kollisionsabfragen delegieren ihre Arbeit an die Klasse `CPolymesh`.

Lokales Management: CSector

*Die grausame
Wahrheit*

Hatte ich weiter oben nicht gesagt, dass die Klasse `CPolygon` die aufwändigste Ableitung von `CLevelObject` sei? Okay, das war gelogen. Wenigstens zum Teil. Die Implementierung der Klasse `CSector` ist zwar nicht sonderlich schwierig zu verstehen, aber sie ist sehr umfangreich. So langsam fällt aber jedes kleine Teilchen an seinen Platz, und wir sollten schon bald wieder Land sehen können. Schauen wir mal, was wir bisher haben. Wir haben Polygone, Polymeshes, Portale, Lichtquellen und Entities. Alle diese Objekte sind natürlich Level-Objekte, die also einem Level zugeordnet werden können. Da wir es hier aber mit einer Portal-Engine zu tun haben und unsere Level in Sektoren unterteilen, die durch Portale miteinander verbunden sind, nehmen wir eine andere Art von Zuteilung vor. Die oben aufgezählten

Objekte sind zwar Level-Objekte, werden aber durch die Sektoren verwaltet, die selbst Level-Objekte sind. Eine Instanz der Klasse `CLevel` hat also eigentlich gar keine Ahnung von Polygonen, Meshs usw. Sie verwaltet nur eine Liste von `CSector`-Instanzen. In diesem Abschnitt werden wir uns also darum kümmern müssen, all die kleinen Level-Objekte zu verwalten, es dem Use-Interface `CApplication` zu ermöglichen, Polygone, Meshs, Portale, Lichter, Spawn Points und Entities in einen Sektor einzufügen, und so weiter. Die Klasse `CSector` ist also unser Mann für das lokale Management und das Arbeitstier unseres Editors.

Aber wenn wir diesen Abschnitt bewältigt haben, dann haben wir das Schlimmste quasi überstanden und unseren Editor so gut wie fertig.

Klassendeklaration

An dieser schaurigen Einleitung könnt ihr schon sehen, dass die Klasse für Sektoren sehr umfangreich ist. Daher halte ich es für besser, sie nicht ganz zu zeigen, sondern einzelne Dinge aus der Klasse herauszugreifen und separat zu erläutern. Am einfachsten ist dabei der Statistik-Kram, der keiner weiteren Erläuterung bedarf. Jedes LOB hat Zähler für die Anzahl an Vertices und Indices bzw. in einigen Fällen auch für die Anzahl an Polygonen (die Meshs und Sektoren) und für die Anzahl an Meshs, Portalen usw. (die Sektoren). Im Editor kann man sich für einen Sektor und für einen Level jeweils eine entsprechende Statistik anzeigen lassen. Ein Sektor bietet für diesen Zweck die folgenden Methoden:

```
UINT CSector::GetNumVertsM(void); // Verts im Meshs
UINT CSector::GetNumIndisM(void); // Indis im Meshs
UINT CSector::GetNumMeshs(void);
UINT CSector::GetNumPorts(void);
UINT CSector::GetNumLights(void);
UINT CSector::GetNumEntities(void);
UINT CSector::GetNumPolysTotal(void);
```

Stück für Stück

*Statistiken
erstellen*

Als Nächstes haben wir die beiden Methoden `CSector::Picked` und `CSector::Intersects`. Diese erfüllen für Sektoren keinen Sinn, da wir ganze Sektoren nicht auswählen oder picken können. Wir müssen aber beide Methoden implementieren, da wir von der Klasse `CLevelObject` ableiten und diese Methoden dort abstrakt deklariert sind. Wir geben dort also einfach `false` zurück, falls jemand sie doch einmal aufrufen sollte. Nun stellt sich die Frage, wie wir einen Sektor auswählen, um ihm andere Level-Objekte zuzuordnen. Und natürlich müssen wir uns auch fragen, wie wir LOBs in einem Sektor selektieren können, denn anders kommen wir ja an die LOBs nicht heran. Nun, die erste Frage kann man einfach beantworten: Das User-Interface des Editors bietet eine Combo-Box an, in der alle Sektoren aufgelistet sind. Der dort ausgewählte Sektor ist aktiv, und alle über das User-Interface erzeugten LOBs wie beispielsweise Polygone oder Lichter werden diesem

*Sektoren können
nicht per Maus
selektiert werden*

Sektor zugeordnet. Die zweite Frage ist auch schnell geklärt, denn für die Selektion und das Picking von LOBs gibt es entsprechende Methoden des Sektors, die dann die LOBs durchgehen und die Operationen dort durchführen.

*Attribute der
Klasse*

Um die Übersichtlichkeit der Klassendeklaration von `CSector` zu gewährleisten, ziehe ich die Attribute aus der Klasse heraus und zeige sie hier. Bei den Portalen hatte ich schon erwähnt, dass Portale die einzigen Objekte sind, die innerhalb eines Levels eindeutige IDs benötigen. Bei den Sektoren lösen wir das ein wenig anders. Wir brauchen zwar keine eindeutigen IDs für das interne Management der einzelnen Instanzen. Aber wir brauchen eine Möglichkeit, wie der Level-Designer die einzelnen Instanzen unterscheiden und entsprechend im Editor anwählen kann. Numerische IDs wie 0, 1, 2 usw. sind wohl wenig aussagekräftig. Jeder Sektor erhält bei uns also einen eindeutigen Namen, den der Level-Designer frei vergeben kann. Natürlich prüft der Editor hier die Integrität der Daten und lässt einen Namen nur zu, falls dieser noch nicht im Level für einen Sektor existiert. Neben dem Namen hat jeder Sektor auch einen eigenen Wert für das dort vorherrschende ambiente Licht. Das kann sich also von Sektor zu Sektor unterscheiden und muss nicht im gesamten Level gleich sein. Die weiteren Attribute der Klasse sind sechs Listen für sechs verschiedene Pointer-Typen und entsprechende Zähler dazu. Man beachte, dass wir einen gesonderten Zähler für Polygone haben, die in Mesh-Objekten stecken, und dass der Zähler für Polygone schon ein Attribut der Basisklasse `CLevelObject` ist.

Hier ist die Liste der Attribute der Klasse `CSector`:

```
private:
    char        m_chName[256]; // Sektorname
    ZFXCOLOR    m_Ambient;     // ambientes Licht

    LPPOLYGON   *m_ppPoly;     // Sektorpolygone
    LPPOLYMESH  *m_ppMesh;     // Sektor-Meshs
    LPPORTAL    *m_ppPortal;   // Sektorportale
    LPLIGHT     *m_ppLight;    // Sektorlichter
    LPENTITY    *m_ppEntity;   // Sektor-Entities
    LPSPAWNPOINT *m_ppSpawn;   // Sektor-Spawns

    UINT        m_NumMeshs;    // Anz. Meshs
    UINT        m_NumPsInMeshs; // Anz. MeshsPolys
    UINT        m_NumPorts;    // Anz. Portale
    UINT        m_NumLights;   // Anz. Lichter
    UINT        m_NumEntys;    // Anz. Entities
    UINT        m_NumSpawns;   // Anz. Spawns

    // temporäre Pointer
    LPLEVELOBJECT *m_ppLob;
    UINT          *m_pC;
```


Das Geheimnis der temporären Pointer werden wir nachher noch lüften. Hier sei nur so viel erwähnt, dass sie lediglich dazu dienen, auf andere Attribute der Klasse zu zeigen, nämlich auf eine der Listen von LOBs und den zugehörigen Zähler. Wir werden auch sogleich noch zwei Methoden für die Klasse entwickeln, die dieses »Umbiegen« der Zeiger übernehmen. Das sind die folgenden beiden:

*Temporäre
Pointer*

```
bool CSector::SetPointers(SELECTMODE sm) {
    if (sm==SM_POLYGON) {
        m_pC = &m_NumPolys;
        m_ppLob = (LPLEVELOBJECT*)m_ppPoly;
    }
    else if (sm==SM_MESH) {
        m_pC = &m_NumMeshes;
        m_ppLob = (LPLEVELOBJECT*)m_ppMesh;
    }
    else if (sm==SM_PORTAL) {
        m_pC = &m_NumPorts;
        m_ppLob = (LPLEVELOBJECT*)m_ppPortal;
    }
    else if (sm==SM_LIGHT) {
        m_pC = &m_NumLights;
        m_ppLob = (LPLEVELOBJECT*)m_ppLight;
    }
    else if (sm==SM_ENTITY) {
        m_pC = &m_NumEntys;
        m_ppLob = (LPLEVELOBJECT*)m_ppEntity;
    }
    else if (sm==SM_SPAWN) {
        m_pC = &m_NumSpawns;
        m_ppLob = (LPLEVELOBJECT*)m_ppSpawn;
    }
    else return false;
    return true;
} // SetPointers
/*-----*/

bool CSector::SetPointers(UINT t) {
    if ( (t <= 0) || (t >= 6)) return false;
    else if (t == 1) {
        m_pC = &m_NumPolys;
        m_ppLob = (LPLEVELOBJECT*)m_ppPoly;
    }
    else if (t == 2) {
        m_pC = &m_NumPorts;
        m_ppLob = (LPLEVELOBJECT*)m_ppPortal;
    }
    else if (t == 3) {
        m_pC = &m_NumLights;
        m_ppLob = (LPLEVELOBJECT*)m_ppLight;
    }
}
```



```

    }
    else if (t == 4) {
        m_pC = &m_NumEntys;
        m_ppLob = (LPLEVELOBJECT*)m_ppEntity;
    }
    else if (t == 5) {
        m_pC = &m_NumSpawns;
        m_ppLob = (LPLEVELOBJECT*)m_ppSpawn;
    }
    return true;
} // SetPointers

```

SELECT-MODE

Einmal verwenden wir als Parameter den Typ `SELECTMODE`, von dem wir hier alle gültigen Fälle mit Ausnahme von `SM_NONE` und `SM_VERTEX` aufgelistet haben. Dieser Typ dient dazu, den Editor in einen bestimmten Modus zu versetzen. Bevor man etwas auswählt und editiert, muss man im User-Interface den entsprechenden Modus umschalten. Das hat den Vorteil, dass man beispielsweise nicht auf Lichtquellen achten muss, wenn man ein Selektionsrechteck in den orthogonalen Ansichten über einen großen Teil des Levels aufzieht und nur Polygone auswählen will. Wenn man im Modus `SM_POLYGON` ist, dann können beliebig viele LOBs eines anderen Typs im Selektionsrechteck liegen; sie werden bei den Operationen dann einfach ignoriert, als wenn sie versteckt wären. Die zweite Version der Methode `CSector::SetPointers` nimmt nur einen `UINT`-Wert als Parameter entgegen. Hierbei ist es mehr oder weniger egal, bei welchem Wert auf was verwiesen wird. Es ist hier nur wichtig, dass wir alle verschiedenen LOBs einmal erwischen, mit Ausnahme der Meshs. Ihr ahnt bereits, dass wir das einsetzen, wenn wir eine Operation auf allen verschiedenen LOBs in einem Sektor in einer Schleife über die verschiedenen LOB-Typen ausführen wollen. Die Meshs sind dabei ein Spezialfall, weil sie Polygone der Level-Geometrie beherbergen, und werden beim Stand von 0 des Schleifenzählers direkt angesprochen. Aber das sehen wir später in anderen Methoden der Klasse `CSector`.

Sektoren-Taufe

Eines möchte ich noch aus der Klassendeklaration ausschneiden und hier vorab zeigen, und zwar die Taufe von Sektoren, also das Vergeben eines Namens. Die Klasse `CSector` bietet die folgenden Methoden für die Namensgebung an, die der Editor nutzt, um den Namen zuzuteilen, die eventuelle Existenz eines zu vergebenden Namens für einen neuen Sektor zu prüfen und den Namen eines Sektors abzufragen:



```

void CSector::SetName(const char *chName) {
    int nL = strlen(chName)+1;
    if (nL>256) memcpy(m_chName, chName, 256);
    else memcpy(m_chName, chName, nL);
} // SetName

/*-----*/

void CSector::GetName(char *chName) {

```

```

        memcpy(chName, m_chName, strlen(m_chName)+1);
    } // GetName

/*-----*/

bool CSector::NameEqual(const char *chName) {
    if (strnicmp(chName, m_chName, 65535)==0)
        return true;
    return false;
} // NameEqual

```

Nun haben wir schon eine ganze Menge gesehen, was die Klasse `CSector` so alles an Attributen und Methoden beinhaltet. Doch da kommt gleich noch viel mehr. Jetzt sieht die Deklaration der Klasse aber nicht mehr so schlimm aus. Man beachte, dass in der folgenden Darstellung die eben abgehandelten Member der `CSector`-Klasse **nicht mehr mit aufgeführt** werden, aber trotzdem in der kompletten Deklaration der Klasse nicht fehlen dürfen.

Klassendeklaration CSector

```

class CSector : public CLevelObject {
public:
    CSector(const char *chName);
    ~CSector(void);

    void Render(ZFXVertexCacheManager*, DWORD,
               bool bHide);
    void RenderLit(ZFXRenderDevice*, bool bHide);
    void Render(ZFXVertexCacheManager *pVCM, bool b)
        { Render(pVCM, LOB_ALL, b); }

    ZFXCOLOR GetAmbient(void) { return m_Ambient; }
    void SetAmbient(float fR, float fG, float fB)
        { m_Ambient.fR = fR;
          m_Ambient.fG = fG;
          m_Ambient.fB = fB; }

    void SelectAll(SELECTMODE, CSelectionBuffer*);
    void SelectInvert(SELECTMODE, CSelectionBuffer*);
    void SelectIntersecting(SELECTMODE,
                           CSelectionBuffer*,
                           const ZFXAabb&, ZFXAXIS);

    void RecalcVertexNormals(void);

    void HideNone(void);
    void HideSelected(SELECTMODE);
    void HideUnselected(void);
    void HideInvert(void);

    void Pick(SELECTMODE, const ZFXRay&,
              CSelectionBuffer*);

```



```

LPLEVELOBJECT AddObject(const CLevelObject&);
void RemoveObject(LPLEVELOBJECT);

void RemovePortalConnections(UINT id);
void NewPortalInTown( CPortal* );
void NewDoorInTown( CEntity* );

void Save(FILE *pFile);
bool Load(FILE *pFile);
void LoadPrefab(const char*, ZFXRenderDevice*,
                CSelectionBuffer*);

private:
    void Init(void);
    void Release(void);
    void CalcBoundingBox(void);
}; // class
typedef class CSector *LPSECTOR;

```

So weit, so gut

Das sieht doch schon mal ganz gut aus. Die Klasse ist immer noch recht umfangreich, aber viele Sachen können wir durch sehr wenige Zeilen Code erschlagen, denn wir loopen einfach durch die Listen der LOBs und delegieren die Arbeit entsprechend. Ein guter Manager weiß, wann er eine Arbeit durch einen kompetenten Mitarbeiter besser erledigen lassen kann, als wenn er es selbst machen würde. Na, dann auf ins Gefecht, den letzten beißen die Hunde.

Eine Instanz erzeugen

*Sektoren sind
nur leere
Ordnungs-
einheiten*

Eine Instanz der Klasse `CSector` zu erzeugen ist nicht weiter aufwändig. Ein Sektor wird über einen Mausklick und die Eingabe eines Namens über einen Dialog im Editor angelegt. Dabei bleibt der Dialog so lange beharrlich, bis der Level-Designer einen noch nicht verwendeten Sektorennamen angibt oder den Vorgang abbricht. Dann kommt der Konstruktor zum Zuge, setzt den Namen und ruft die `CSector::Init`-Methode auf. Der Destruktor verweist wie üblich einfach auf eine entsprechende `CSector::Release`-Methode.



```

CSector::CSector(const char *chName) {
    Init();
    m_ID = 0;
    SetName(chName);
} // constructor

/*-----*/

CSector::~CSector(void) {
    Release();
} // destructor

/*-----*/

void CSector::Init(void) {

```

```

    m_NumVerts      = 0;
    m_NumIndis      = 0;
    m_NumPorts      = 0;
    m_NumPolys      = 0;
    m_NumMeshs      = 0;
    m_NumPsInMeshs  = 0;
    m_NumLights     = 0;
    m_NumEntys      = 0;
    m_NumSpawns     = 0;
    m_bHidden       = false;
    m_bSelected     = false;
    m_pC             = NULL;
    m_ppLob         = NULL;
    m_ppPoly        = NULL;
    m_ppMesh        = NULL;
    m_ppPortal      = NULL;
    m_ppLight       = NULL;
    m_ppEntity      = NULL;
    m_ppSpawn       = NULL;
    m_Type          = LOB_SECTOR;
    m_Ambient.fr     = 1.0f;
    m_Ambient.fg     = 1.0f;
    m_Ambient.fb     = 1.0f;
    m_Ambient.fA     = 1.0f;
} // Init

/*-----*/

void CSector::Release(void) {
    UINT i=0;

    for (i=0; i<m_NumPolys; i++)
        SAFE_DELETE(m_ppPoly[i]);
    for (i=0; i<m_NumMeshs; i++)
        SAFE_DELETE(m_ppMesh[i]);
    for (i=0; i<m_NumPorts; i++)
        SAFE_DELETE(m_ppPortal[i]);
    for (i=0; i<m_NumLights; i++)
        SAFE_DELETE(m_ppLight[i]);
    for (i=0; i<m_NumEntys; i++)
        SAFE_DELETE(m_ppEntity[i]);
    for (i=0; i<m_NumSpawns; i++)
        SAFE_DELETE(m_ppSpawn[i]);

    SAFE_FREE(m_ppPoly);
    SAFE_FREE(m_ppMesh);
    SAFE_FREE(m_ppPortal);
    SAFE_FREE(m_ppLight);
    SAFE_FREE(m_ppEntity);
    SAFE_FREE(m_ppSpawn);
} // Release

```

**Speicheralloka-
tionsverfahren
beachten**

Beim Freigeben des Speichers achten wir natürlich darauf, wie wir den Speicher bereitgestellt haben: über Methoden oder Operatoren. Die zweite Variante bevorzugen wir für Instanzen von Objekten, und die erste Variante, wenn wir die Menge des Speichers zur Laufzeit ständig ändern müssen. Das mag verwirrend erscheinen, aber ich hatte ja bereits auf den Sinn und Zweck eines selbst geschriebenen Memory-Managers hingewiesen.

Laden und speichern

**Jetzt kommt es
ganz dick**

Das Laden und Speichern gestaltet sich vom Prinzip wie bei den bisher gezeigten Methoden. Aber natürlich kann man davon ausgehen, dass eine so umfangreiche Klasse auch recht viele Attribute abzuarbeiten hat; daher sind die Methoden um einiges länger als die, die wir bisher gesehen haben.

**Speichern eines
Sektors – Skins
zählen**

Eigentlich müssen wir beim Speichern eines Sektors nichts anderes tun, als die Listen der LOBs zu durchlaufen und dort jede einzelne Instanz zu bitten, sich in die Datei zu speichern. Aber wir investieren hier noch ein wenig mehr Arbeit, die für den Editor selbst an sich unnötig wäre. Wir nehmen nämlich erst einmal die Liste der ungruppierten Polygone und sortieren sie nach der Skin, die sie verwenden. Dann zählen wir die Anzahl an verschiedenen Skins, die von diesen Polygonen verwendet werden. Im Anschluss daran zählen wir, wie viele verschiedene Skins, die wir bisher noch nicht gezählt haben, die Meshs noch beizusteuern haben. So erhalten wir die Anzahl aller verschiedenen Skins im gesamten Sektor und speichern diese auch mit in die Datei. Das Ergebnis benötigen wir hier wie gesagt nicht, aber es erleichtert einem Spiel die Arbeit, seine Daten in verschiedenen Buffer zu organisieren, die nach Texturen geordnet sind. Im folgenden Kapitel über das Spiel *Pandoras Legacy* werden wir das beim Laden der Level noch sehen. Der Rest der Speicher-Methode ist trivial.



```
void CSector::Save(FILE *pFile) {
    UINT i=0, NumSkins=0;
    int NewSkin=-1;
    int n=0;

    UINT aSkins[400];

    if (m_NumPolys > 0) {
        // Sortiere Polygone nach Textur
        qsort(m_ppPoly, m_NumPolys, sizeof(LPPOLYGON),
            SortBySkin);

        // Anzahl verschiedener Skins im Sektor
        UINT OldSkin = m_ppPoly[0]->GetSkinID();
        aSkins[NumSkins++] = OldSkin;
        for (i=1; i<m_NumPolys; i++) {
            NewSkin = m_ppPoly[i]->GetSkinID();
            if (NewSkin != OldSkin) {
```

```

        aSkins[NumSkins++] = NewSkin;
    }
    OldSkin = NewSkin;
}

// Anzahl anderer Skins in Meshs
for (UINT m=0; m<m_NumMeshs; m++) {
    m_ppMesh[m]->CountNonRedundantSkins(
        aSkins, &NumSkins );
}

n = strlen(m_chName);
fwrite(&n, sizeof(int), 1, pFile);
fwrite(m_chName, n, 1, pFile);

fwrite(&NumSkins,      sizeof(UINT),    1, pFile);
fwrite(&m_NumVerts,    sizeof(UINT),    1, pFile);
fwrite(&m_NumIndis,    sizeof(UINT),    1, pFile);
fwrite(&m_NumPolys,    sizeof(UINT),    1, pFile);
fwrite(&m_NumPsInMeshs, sizeof(UINT),    1, pFile);
fwrite(&m_NumMeshs,    sizeof(UINT),    1, pFile);
fwrite(&m_NumPorts,    sizeof(UINT),    1, pFile);
fwrite(&m_NumLights,   sizeof(UINT),    1, pFile);
fwrite(&m_NumEntys,    sizeof(UINT),    1, pFile);
fwrite(&m_NumSpawns,   sizeof(UINT),    1, pFile);
fwrite(&m_Aabb,        sizeof(ZFXAabb),  1, pFile);
fwrite(&m_Ambient,     sizeof(ZFXCOLOR), 1, pFile);
fflush(pFile);

for (i=0; i<m_NumMeshs; i++)
    m_ppMesh[i]->Save(pFile);
for (i=0; i<m_NumPolys; i++)
    m_ppPoly[i]->Save(pFile);
for (i=0; i<m_NumPorts; i++)
    m_ppPortal[i]->Save(pFile);
for (i=0; i<m_NumLights; i++)
    m_ppLight[i]->Save(pFile);
for (i=0; i<m_NumEntys; i++)
    m_ppEntity[i]->Save(pFile);
for (i=0; i<m_NumSpawns; i++)
    m_ppSpawn[i]->Save(pFile);
} // Save

```

Die Methode `CPolymesh::CountNonRedundantSkins` haben wir bisher noch nicht gesehen. Wir müssen sie in der Klasse `CPolymesh` noch ergänzen. Die Implementierung könnt ihr euch auf der CD-ROM anschauen, sie ist super simpel. Die Methode bekommt ein Array mit SkinIDs und die Anzahl der Elemente in dem Array übergeben. Nun durchläuft ein Mesh alle seine Poly-

*Skins in Meshs
zählen*

gone und schaut nach, ob die SkinID jeweils bereits in der Liste ist. Falls nicht, wird sie hinzugefügt und der Zähler heraufgesetzt.

*Laden eines
Sektors*

Das Laden eines Sektors gestaltet sich nun wieder so einfach, wie wir uns das vorgestellt haben. Wir saugen einfach die Daten in der entsprechenden Reihenfolge aus der offenen Datei und greifen dabei insbesondere auf die Lade-Methoden der einzelnen LOB-Klassen zurück. Natürlich vergessen wir auch nicht, über die Methoden der Basisklasse die Menge an Speicher anzufordern, die wir für die Listen der LOBs brauchen.



```
bool CSector::Load(FILE *pFile) {
    UINT i=0, NumSkins=0;
    int n=0;

    fread(&n, sizeof(int), 1, pFile);
    fread(m_chName, n, 1, pFile);
    m_chName[n] = '\\0';

    fread(&NumSkins,      sizeof(UINT), 1, pFile);
    fread(&m_NumVerts,    sizeof(UINT), 1, pFile);
    fread(&m_NumIndis,    sizeof(UINT), 1, pFile);
    fread(&m_NumPolys,     sizeof(UINT), 1, pFile);
    fread(&m_NumPsInMeshs, sizeof(UINT), 1, pFile);
    fread(&m_NumMeshs,    sizeof(UINT), 1, pFile);
    fread(&m_NumPorts,    sizeof(UINT), 1, pFile);
    fread(&m_NumLights,   sizeof(UINT), 1, pFile);
    fread(&m_NumEntys,    sizeof(UINT), 1, pFile);
    fread(&m_NumSpawns,   sizeof(UINT), 1, pFile);
    fread(&m_Aabb,        sizeof(ZFXAabb), 1, pFile);
    fread(&m_Ambient,     sizeof(ZFXCOLOR), 1, pFile);

    m_ppMesh  = (LPPOLYMESH*) InitMem( (LPVOID*)
                                       m_ppMesh, m_NumMeshs,
                                       sizeof(LPPOLYMESH));
    m_ppPoly  = (LPPOLYGON*)  InitMem( (LPVOID*)
                                       m_ppPoly, m_NumPolys,
                                       sizeof(LPPOLYGON));
    m_ppPortal = (LPPORTAL*)  InitMem( (LPVOID*)
                                       m_ppPortal, m_NumPorts,
                                       sizeof(LPPORTAL));
    m_ppLight  = (LPLIGHT*)   InitMem( (LPVOID*)
                                       m_ppLight, m_NumLights,
                                       sizeof(LPLIGHT));
    m_ppEntity = (LPENTITY*)   InitMem( (LPVOID*)
                                       m_ppEntity, m_NumEntys,
                                       sizeof(LPENTITY));
    m_ppSpawn  = (LPSPAWNPOINT*)InitMem( (LPVOID*)
                                       m_ppSpawn, m_NumSpawns,
                                       sizeof(LPSPAWNPOINT));
}
```



```

for (i=0; i<m_NumMeshs; i++) {
    m_ppMesh[i] = new CPolymesh();
    m_ppMesh[i]->Load(pFile);
}
for (i=0; i<m_NumPolys; i++) {
    m_ppPoly[i] = new CPolygon();
    m_ppPoly[i]->Load(pFile);
}
for (i=0; i<m_NumPorts; i++) {
    m_ppPortal[i] = new CPortal();
    m_ppPortal[i]->Load(pFile);
}
for (i=0; i<m_NumLights; i++) {
    m_ppLight[i] = new CLight();
    m_ppLight[i]->Load(pFile);
}
for (i=0; i<m_NumEntys; i++) {
    m_ppEntity[i] = new CEntity();
    m_ppEntity[i]->Load(pFile);
}
for (i=0; i<m_NumSpawns; i++) {
    m_ppSpawn[i] = new CSpawnPoint();
    m_ppSpawn[i]->Load(pFile);
}
return true;
} // Load

```

Ein PreFab laden

Nun, da wir schon wissen, wie man einen Level laden kann, und wo wir gerade beim Laden sind, schauen wir uns auch gleich noch an, wie man ein PreFab laden kann. Der aufmerksame Beobachter hat bereits gemerkt, dass die Klasse `CSector` in der Tat PreFabs nur laden, sie aber nicht speichern kann. Das liegt daran, dass wir alle LOBs immer nur einem bestimmten Sektor zuordnen. Wenn wir ein PreFab laden, dann laden wir es gleich in den Sektor, in dem es auch sein soll. Wenn man ein PreFab speichern möchte, dann muss man eine beliebige Menge an Polygonen im Level selektieren und diese dann als PreFab abspeichern. Das können natürlich auch Polygone aus einem Mesh sein.

*PreFabs sind auch
nur Polygone*

Doch die Menge an Polygonen, die man gerade selektiert hat und als PreFab speichern möchte, ist bereits exakt im Selektions-Buffer, also einer Instanz von `CSelectionBuffer`, enthalten, und daher verfügt diese Klasse dann entsprechend über die Methode, ein PreFab abzuspeichern. Aber schauen wir uns erst einmal die Methode zum Laden eines PreFabs an:

*Speichern über
den Selektions-
Buffer*



```

void CSector::LoadPrefab(const char *ch,
                        ZFXRenderDevice *pDevice,
                        CSelectionBuffer *pSB) {
    LPPOLYGON    pPoly = new CPolygon();
    LPLEVELOBJECT pLob = NULL;
    UINT NumPolys=0, n=0;
    int  nIndex=-1;
    char buffer[2048];

    FILE *pFile = fopen(ch, "rb");
    if (!pFile) return;

    pPoly->Reset();

    // Wie viele Polygone kommen hinzu?
    fread(&NumPolys, sizeof(UINT), 1, pFile);

    if (NumPolys == 0) return;

    for (UINT i=0; i<NumPolys; i++) {
        pPoly->Reset();

        // Name der diffusen Textur
        fread(&n, sizeof(UINT), 1, pFile);
        fread(buffer, n, 1, pFile);
        buffer[n] = '\\0';

        // Ist Textur schon im aktuellen Level geladen?
        nIndex = cbhContainsString(CApplication::m_G.hGUI[
                                CSK_LIST], buffer);

        // Lade Polygone
        pPoly->Load(pFile);

        // Lade Textur, falls nötig
        if (nIndex == -1) {
            nIndex = CApplication::m_G.Level.CreateSkinID(
                    pDevice, buffer, 1.0f );
            cbhAddItem(CApplication::m_G.hGUI[CSK_LIST],
                    buffer, NULL, true);
        }

        pPoly->SetSkinID(nIndex);
        pLob = AddObject( *pPoly );
        pSB->AddLob( pLob );
    }

    delete pPoly;
    fclose(pFile);
} // LoadPrefab

```

Da PreFabs nur aus Polygonen bestehen, speichern wir sie auch nur als einzelne Polygone, auch wenn diese vorher beim Abspeichern Teil eines Meshs gewesen sein sollten. In der Methode `CSector::LoadPrefab` hier müssen wir also lediglich schauen, wie viele Polygone in der Datei des PreFabs stehen, und diese dann in einer Schleife einzeln auslesen. Dazu bemühen wir die Methode `CPolygon::Load` und fügen die neue Instanz eines Polygons über die Methode `CSector::AddObject` in den Sektor ein. Zudem fügen wir auch noch mit `CSelectionBuffer::AddLob` eine Referenz auf das eingefügte Polygon zu dem Selektions-Buffer hinzu. Auf diese Weise werden alle Polygone des geladenen PreFabs nach dem Ladevorgang im Selektions-Buffer enthalten sein. Das heißt, man hat das geladene PreFab bereits selektiert und kann es gleich editieren.

Grundlegendes

Dann sehen wir noch zwei Funktionen mit dem Präfix `cbh` in der Methode. Diese Funktionen sind **Combo-Box-Helper-Funktionen** und erleichtern uns den Umgang mit einem WinAPI-Combo-Box-Steuerelement. Wir verwenden hier wieder das statische Element `m_G` der Klasse `CApplication`, in dem die wichtigsten Attribute einer Instanz des User-Interfaces unseres Editors zugänglich sind. Über das Feld `hGUI` erhalten wir das Handle der Combo-Box, an die wir eine Frage haben. Bei dieser Combo-Box handelt es sich um eine Liste aller Skins, die wir im aktuellen Level bereits geladen haben. Diese Skins werden anhand des Namens der diffusen Textur unterschieden. Für jedes Polygon im PreFab haben wir auch den Namen der diffusen Textur abgespeichert, die das Polygon verwendet. Nun fragen wir die Combo-Box zuerst, ob diese Textur des zu ladenden Polygons aus dem PreFab bereits geladen ist. Ist dies nicht der Fall, dann laden wir die Textur mit der Methode `CLevel::CreateSkinID` und fügen ihren Namen der Combo-Box hinzu. Anschließend setzen wir noch die nun aktuelle SkinID für das Polygon fest. Damit ist das Laden eines PreFabs abgeschlossen.

Texturen & Co

Kollisionsabfrage und Selektion von LOBs

Nun werden wir zum ersten Mal sehen, warum eine Basisklasse für die Level-Objekte wirklich sehr viel Sinn macht und uns hier einen Haufen Quellcode-Zeilen sparen kann. Die jetzt folgende Methode `CSector::Pick` sorgt dafür, dass der Level-Designer im aktuellen Sektor LOBs in der perspektivischen (dreidimensionalen) Ansicht mit der Maus anklicken kann, um sie auszuwählen. Wie weiter oben schon erläutert wurde, kann das natürlich immer nur ein LOB sein. Über den Parameter `SELECTMODE` teilen wir der Methode zudem mit, welcher Modus gerade im Editor eingeschaltet ist, da nur entsprechende LOBs ausgewählt werden können. Vertices können jedoch nicht in der perspektivischen Ansicht ausgewählt werden. Damit haben wir den Strahl von der Kameraposition in Blickrichtung entsprechend dem von 2D-Koordinaten zu einem 3D-Strahl umgerechneten Mausclick.

Die Power unserer Basisklasse

Nun verwenden wir die Methode `CSector::SetPointers`, um unsere temporären Zeiger vom Typ der Basisklasse `CLevelObject` und den `UINT`-Zähler auf die

Temporäre Zeiger

entsprechenden Attribute der Klasse umzubiegen, also auf die Liste von Polygonen und deren Zähler oder auf die Liste von Meshs und deren Zähler usw., je nachdem, welcher Selektionsmodus gerade aktiv ist. Danach loopen wir über die temporären Zeiger durch die entsprechend richtige Liste und versuchen dort, ein Objekt mit dem Strahl über die Methode zu picken, die die Basisklasse ihren Ableitungen zwingend vorgeschrieben hat. Sollte ein Pick auf ein LOB erfolgreich gewesen sein, so müssen wir trotzdem weiter-suchen, denn wir wollen ja das Objekt finden, das die geringste Distanz zum Ursprung des Strahls hat, weil das dem Betrachter am nächsten liegt. Anderenfalls würden wir ein Objekt selektieren, das zwar auch vom Strahl getroffen wird, aber in der perspektivischen Ansicht eigentlich durch ein dichter am Betrachter liegendes Objekt verdeckt ist.

**Spezialfall
Polygone**

Eines müssen wir aber noch berücksichtigen: Wenn der Level-Designer ein Polygon picken möchte, dann kann dies ein freies Polygon sein oder ein Polygon, welches Teil eines Meshs ist. Im Fall von Polygonen müssen wir also zusätzlich noch die Meshs durchlaufen und dort nach einem näher gelegenen Polygon picken. Die Methode, um ein beliebiges LOB aus einem Sektor zu picken, sieht wie folgt aus:



```
void CSector::Pick(SELECTMODE sm, const ZFXRay &Ray,
                  CSelectionBuffer *pSB) {
    LPLEVELOBJECT pPicked=NULL, pPoly = NULL;

    float fMin=65535.0f, fD=0.0f;

    // Polygon ist ein Spezialfall

    if (!SetPointers(sm)) return;

    // laufe durch alle LOBs in der Liste
    for (UINT i=0; i<(*m_pC); i++) {
        // ignoriere versteckte LOBs
        if (m_ppLob[i]->IsHidden()) continue;

        // versuche einen Pick
        if (m_ppLob[i]->Picked(Ray, &fD)) {
            if (fD < fMin) {
                fMin = fD;
                pPicked = m_ppLob[i];
            }
        }
    }

    // falls wir Polygone picken sollen, müssen wir
    // auch noch die Polygone in Meshs picken
    if (sm==SM_POLYGON) {
        for (i=0; i<m_NumMeshs; i++) {
```

```

        if (m_ppMesh[i]->IsHidden()) continue;

        if (pPoly=m_ppMesh[i]->PickPoly(Ray,fMin,&fD))
        {
            fMin = fD;
            pPicked = pPoly;
            pPoly = NULL;
        }
    } // if [POLYGON]

    if (pPicked) pSB->AddLob(pPicked);
} // Pick

```

Wie ihr schön sehen könnt, wird das gepickte LOB, sofern der Level-Designer eines mit der Maus getroffen hat, direkt in den Selektions-Buffer eingefügt. In unserem Editor ist es auch ohne weiteres möglich, durch ein gleichzeitiges Drücken der **[STRG]**-Taste und ein Anklicken eines LOBs in der perspektivischen Ansicht nacheinander mehrere Objekte auszuwählen.

*Direkt in den
Selektions-Buffer*

Der Level-Designer kann im User-Interface unseres Editors mit einem Mausklick auf einen Schlag alle LOBs des aktuell eingestellten Selektionstyps auswählen. Wenn man beispielsweise der Übersichtlichkeit halber alle Lichter verstecken möchte, dann kann man den Selektionstyp LICHTQUELEN einstellen und einfach per Mausklick alle Objekte auswählen und dann verstecken. Wir bieten hier sogar noch eine Methode an, mit der man per Mausklick den Selektionsstatus aller LOBs eines Typs in einem Sektor umdrehen kann. Die beiden folgenden Methoden sind selbsterklärend, da ganz ähnlich wie die eben gezeigte Methode für das Picking aufgebaut sind. Auch hier sind die Polygone jeweils ein Spezialfall, da wir auch die in den Meshs liegenden Polygone nicht vergessen dürfen.

*Alle Objekte eines
Typs auswählen*

```

void CSector::SelectAll(SELECTMODE sm,
                        CSelectionBuffer *pSB) {
    // Spezialfall Polygone
    if (sm==SM_POLYGON) {
        // zuerst Polygone in den Meshs auswählen
        for (UINT m=0; m<m_NumMeshs; m++) {
            if (m_ppMesh[m]->IsHidden()) continue;
            m_ppMesh[m]->SelectAllPolys( pSB );
        }
    }

    if (!SetPointers(sm)) return;

    for (UINT i=0; i<(*m_pC); i++) {
        if (m_ppLob[i]->IsHidden()) continue;
        pSB->AddLob( m_ppLob[i] );
    }
}

```



```

    } // SelectAll
/*-----*/

void CSector::SelectInvert(SELECTMODE sm,
                          CSelectionBuffer *pSB) {
    UINT i=0;

    // Spezialfall Polygone in Meshs
    if (sm == SM_POLYGON) {
        for (i=0; i<m_NumMeshs; i++) {
            if ( m_ppMesh[i]->IsHidden() ) continue;
            else m_ppMesh[i]->SelectInvert(pSB);
        }
    }
    // Vertices hier ignorieren
    else if (sm == SM_VERTEX) { return; }

    if (!SetPointers(sm)) return;

    for (i=0; i<(*m_pC); i++) {
        if (m_ppLob[i]->IsHidden()) continue;

        if ( !m_ppLob[i]->IsSelected() )
            pSB->AddLob( m_ppLob[i] );
        else m_ppLob[i]->DeSelect();
    }
} // SelectInvert

```

*Auswählen per
Selektions-
rechteck in ortho-
gonalen Ansichten*

Nun kommen wir zu der Selektion von Level-Objekten, die der Level-Designer in den orthogonalen Ansichten durch Drücken der linken Maustaste und Aufziehen eines Selektionsrechtecks mit der Maus durchführen kann. Bei den einzelnen LOBs hatte ich schon darauf hingewiesen, dass wir in diesem Fall das Selektionsrechteck als dreidimensionale Bounding-Box umsetzen, deren fehlende dritte Dimension von $+\infty$ bis $-\infty$ läuft, wobei wir einfach einen entsprechend hohen Wert einsetzen, um ∞ anzunähern. Diese Methode ist fast genauso simpel wie die beiden eben gesehenen. Neben dem Spezialfall der Polygone haben wir aber noch den Spezialfall der Vertices. Wenn der Level-Designer einzelne Vertices auswählen möchte, dann müssen wir diese einmal in den Polygonen und einmal in den Polygonen in Meshs suchen.



```

void CSector::SelectIntersecting(SELECTMODE sm,
                                CSelectionBuffer *pSB,
                                const ZFXAabb &aabb, ZFXAXIS axis) {
    UINT i=0;

    // 1. Spezialfall: Vertices in Polygonen und Meshs
    if (sm == SM_VERTEX) {
        for (i=0; i<m_NumMeshs; i++) {

```

```

        if ( m_ppMesh[i]->IsHidden() ) continue;
        else m_ppMesh[i]->GetIntersectingVerts(
                                aabb,pSB,axis);
    }
    for (i=0; i<m_NumPolys; i++) {
        if ( m_ppPoly[i]->IsHidden() ) continue;
        else m_ppPoly[i]->GetIntersectingVerts(
                                aabb, pSB, axis);
    }
} // Spezialfall VERTEX

// 2. Spezialfall: Polygone in Meshs
else if (sm == SM_POLYGON) {
    for (i=0; i<m_NumMeshs; i++) {
        if ( m_ppMesh[i]->IsHidden() ) continue;
        else m_ppMesh[i]->GetIntersectingPolys(
                                aabb,pSB,axis);
    }
} // Spezialfall POLYGON

if (!SetPointers(sm)) return;

for (i=0; i<(*m_pC); i++) {
    if (m_ppLob[i]->IsHidden()) continue;
    if ( m_ppLob[i]->Intersects(aabb, axis) )
        pSB->AddLob( m_ppLob[i] );
}
} // SelectIntersecting

```

Wie ihr seht, spart uns auch hier die Basisklasse für LOBs eine Menge Fallunterscheidungen. Wir können einfach über einen Pointer vom Typ der Basisklasse die entsprechenden Methoden aufrufen und müssen explizit nicht unterscheiden, um welchen LOB-Typ es wirklich geht.

Objekte im Sektor verstecken

Wie Objekte versteckt werden, haben wir schon mehrfach gesehen. Wir müssen dazu einfach das Flag eines LOBs auf den gewünschten Status setzen. Unser Editor bietet vier Optionen zum Verstecken in einem Sektor an, die wie folgt lauten:

Vier Quickies

- ➔ verstecke nichts (alles Versteckte wieder anzeigen)
- ➔ verstecke selektierte Objekte
- ➔ verstecke unselektierte Objekte
- ➔ invertiere den Versteckzustand bei allen Objekten

Die folgenden vier Quickies erledigen jeweils einen dieser Jobs. Auch hier müssen wir immer wieder einen Spezialfall beachten, und dieser hängt immer mit den Polygonen in Meshs zusammen. Aber das habt ihr nun schon verstanden, denke ich mal. Es sei noch erwähnt, dass wir hier an drei Stellen die Methode `CSector::SetPointers` mit dem `UINT`-Parameter und nicht mehr mit dem `SELECTMODE`-Parameter verwenden. Das kommt daher, weil wir den Selektionsmodus nur dann beachten müssen, wenn Elemente auch ausgewählt wurden. In den drei anderen Fällen müssen wir alle Listen einmal durchlaufen und lassen uns von den eben genannten Methode nach und nach die temporären Zeiger entsprechend setzen. Die vier Methoden zum Verstecken von Level-Objekten in einem Sektor sehen wie folgt aus:



```
void CSector::HideNone(void) {
    for (UINT t=0; t<6; t++) {
        // Spezialfall Polygone in Meshs
        if (t == 0) {
            for (UINT cm=0; cm<m_NumMeshs; cm++) {
                m_ppMesh[cm]->UnHide();
                m_ppMesh[cm]->HideNoPolys();
            }
            continue;
        }

        if ( !SetPointers(t) ) continue;

        for (UINT i=0; i<(*m_pC); i++)
            m_ppLob[i]->UnHide();
    }
} // HideNone
/*-----*/

void CSector::HideSelected(SELECTMODE sm) {
    UINT i=0;

    // Spezialfall Polygone in Meshs
    if (sm == SM_POLYGON) {
        for (i=0; i<m_NumMeshs; i++)
            m_ppMesh[i]->HideSelectedPolys();
    }
    else if (sm == SM_VERTEX) { return; }

    if (!SetPointers(sm)) return;

    for (i=0; i<(*m_pC); i++) {
        if ( m_ppLob[i]->IsSelected() )
            m_ppLob[i]->Hide();
    }
} // HideSelected
/*-----*/
```



```

void CSector::HideUnselected(void) {
    for (UINT t=0; t<6; t++) {
        // Spezialfall Meshs
        if (t == 0) {
            for (UINT m=0; m<m_NumMeshs; m++)
                m_ppMesh[m]->HideUnselectedPolys();
            continue;
        }

        if ( !SetPointers(t) ) continue;

        for (UINT i=0; i<(*m_pC); i++) {
            if ( ! m_ppLob[i]->IsSelected() )
                m_ppLob[i]->Hide();
        }
    } // HideUnselected
} /*-----*/

```

```

void CSector::HideInvert(void) {
    for (UINT t=0; t<6; t++) {
        // Spezialfall Meshs
        if (t == 0) {
            for (UINT cm=0; cm<m_NumMeshs; cm++) {
                if ( m_ppMesh[cm]->IsHidden() ) {
                    m_ppMesh[cm]->UnHide();
                    m_ppMesh[cm]->HideNoPolys();
                }
                else m_ppMesh[cm]->HideInvertPolys();
            }
            continue;
        }

        if ( !SetPointers(t) ) continue;

        for (UINT i=0; i<(*m_pC); i++) {
            if (m_ppLob[i]->IsHidden())
                m_ppLob[i]->UnHide();
            else
                m_ppLob[i]->Hide();
        }
    } // HideInvert
}

```

Objekte hinzufügen und entfernen

Das Entfernen eines Level-Objekts aus einem Sektor gestaltet sich recht einfach. Wir erhalten vom Selektions-Buffer die Adresse des Objekts, das wir löschen wollen. Wir müssen nun lediglich schauen, von welchem Typ das

*LOB aus einem
Sektor löschen*

Objekt ist, und die entsprechende Liste im Sektor nach diesem Pointer absuchen. Haben wir ihn gefunden, so löschen wir das dort gespeicherte Objekt und lassen das letzte Objekt aus der Liste an diese Position aufrücken, um die kontinuierliche Linearität der Liste zu gewährleisten. Doch was wäre die Welt ohne Spezialfälle. Wenn wir echte Geometrie löschen, also Polygone oder Meshs, dann müssen wir entsprechende Zähler für die Statistiken herunterzählen. Wenn wir Polygone löschen, dann müssen wir auch in Meshs nach diesen Polygonen suchen und ein Mesh auch löschen, wenn wir das letzte Polygon aus ihm herausgelöscht haben. Der Rest der Methode zum Entfernen eines Objekts aus einem Sektor und damit aus dem Level ist selbsterklärend.



```
void CSector::RemoveObject(LPLEVELOBJECT pLob) {
    LPPOLYMESH pMesh=NULL;

    LPLEVELOBJECT *ppLob = NULL;
    UINT *pC=NULL;

    // Suche Polygon auch in Meshs
    switch ( pLob->GetType() ) {
        case LOB_POLYGON:
            m_NumVerts -= pLob->GetNumVerts();
            m_NumIndis -= pLob->GetNumIndis();

            if ( ((LPPOLYGON)pLob)->IsPartOfMesh() ) {
                pMesh = ((LPPOLYGON)pLob)->GetParent();
                pMesh->RemovePolygon((LPPOLYGON)pLob);

                // lösche leere Meshs
                if ( !pMesh->GetNumPolys() )
                    RemoveObject(pMesh);
                m_NumPsInMeshs--;
                return;
            }

            pC = &m_NumPolys;
            ppLob = (LPLEVELOBJECT*)m_ppPoly;
            break;

        case LOB_MESH:
            m_NumVerts -= pLob->GetNumVerts();
            m_NumIndis -= pLob->GetNumIndis();
            m_NumPsInMeshs -= pLob->GetNumPolys();

            pC = &m_NumMeshs;
            ppLob = (LPLEVELOBJECT*)m_ppMesh;
            break;

        case LOB_PORTAL:
            pC = &m_NumPorts;
```

```

        ppLob = (LPLEVELOBJECT*)m_ppPortal;
        break;
    case LOB_LIGHT:
        pC = &m_NumLights;
        ppLob = (LPLEVELOBJECT*)m_ppLight;
        break;
    case LOB_ENTITY:
        pC = &m_NumEntys;
        ppLob = (LPLEVELOBJECT*)m_ppEntity;
        break;
    case LOB_SPAWNPOINT:
        pC = &m_NumSpawns;
        ppLob = (LPLEVELOBJECT*)m_ppSpawn;
        break;
    default: return;
} // switch

for (UINT i=0; i<(*pC); i++)
    if (ppLob[i] == pLob) break;

if (i>=(*pC)) return;

SAFE_DELETE(ppLob[i]);

if (i != ((*pC)-1)) ppLob[i] = ppLob[(*pC)-1];
(*pC)--;
return;
} // RemoveObject

```

So schlimm war das doch gar nicht, oder? Und das Hinzufügen eines Objekts läuft im Großen und Ganzen vollkommen analog ab. Hier müssen wir nur darauf achten, dass wir den vorhandenen Speicher für die entsprechende Liste prüfen und gegebenenfalls über die Methoden der Basisklasse erweitern müssen, um ein zusätzliches Objekt aufnehmen zu können. Aufgrund der Fallunterscheidungen für die verschiedenen Objekte ist diese Methode recht umfangreich, aber die Welt kann ja nicht nur aus Pfannkuchen bestehen.

*Ein Objekt zu
einem Sektor-
hinzufügen*

Erwähnenswert ist hier noch, dass wir jeweils die Adresse eines hinzugefügten Objekts an den Aufrufer zurückgeben. Das machen wir, damit der Editor, der als Reaktion auf den Input des Level-Designers diese Methode aufgerufen hat, die Adresse erhält und sie in den Selektions-Buffer einfügen kann. Ein neu in einen Sektor eingefügtes Objekt wird damit automatisch selektiert.

Rückgabewert



```

LPLEVELOBJECT CSector::AddObject(const CLevelObject
                                &Lob) {
    LPLEVELOBJECT pLob = (LPLEVELOBJECT)&Lob;

    switch ( pLob->GetType() ) {
        case LOB_POLYGON:
            m_ppPoly = (LPPOLYGON*) CheckMem( (LPVOID*)
                                                m_ppPoly, m_NumPolys,
                                                sizeof(LPPOLYGON) );

            m_ppPoly[m_NumPolys] = new CPolygon();
            m_ppPoly[m_NumPolys]->CopyOf( pLob );

            m_NumVerts += pLob->GetNumVerts();
            m_NumIndis += pLob->GetNumIndis();
            m_NumPolys++;
            return m_ppPoly[m_NumPolys-1];

        case LOB_MESH:
            m_ppMesh = (LPPOLYMESH*) CheckMem( (LPVOID*)
                                                m_ppMesh, m_NumMeshs,
                                                sizeof(LPPOLYMESH) );

            m_ppMesh[m_NumMeshs] = new CPolymesh();
            m_ppMesh[m_NumMeshs]->CopyOf( pLob );

            m_NumPsInMeshs += pLob->GetNumPolys();
            m_NumVerts += pLob->GetNumVerts();
            m_NumIndis += pLob->GetNumIndis();
            m_NumMeshs++;
            return m_ppMesh[m_NumMeshs-1];

        case LOB_ENTITY:
            m_ppEntity = (LPENTITY*) CheckMem( (LPVOID*)
                                                m_ppEntity, m_NumEntys,
                                                sizeof(LPENTITY) );

            m_ppEntity[m_NumEntys] = new CEntity();
            m_ppEntity[m_NumEntys]->CopyOf( pLob );
            m_NumEntys++;
            return m_ppEntity[m_NumEntys-1];

        case LOB_PORTAL:
            m_ppPortal = (LPPORTAL*) CheckMem( (LPVOID*)
                                                m_ppPortal, m_NumPorts,
                                                sizeof(LPPORTAL) );

            m_ppPortal[m_NumPorts] = new CPortal();
            m_ppPortal[m_NumPorts]->CopyOf( pLob );
            m_ppPortal[m_NumPorts]->SetItsSector(this);
    }
}

```

```

        m_NumPorts++;
        return m_ppPortal[m_NumPorts-1];

    case LOB_LIGHT:
        m_ppLight = (LPLIGHT*) CheckMem( (LPVOID*)
            m_ppLight, m_NumLights,
            sizeof(LPLIGHT) );

        m_ppLight[m_NumLights] = new CLight();
        m_ppLight[m_NumLights]->CopyOf( pLob );
        m_NumLights++;
        return m_ppLight[m_NumLights-1];

    case LOB_SPAWNPOINT:
        m_ppSpawn = (LPSPAWNPOINT*) CheckMem( (LPVOID*)
            m_ppSpawn, m_NumSpawns,
            sizeof(LPSPAWNPOINT) );

        m_ppSpawn[m_NumSpawns] = new CSpawnPoint();
        m_ppSpawn[m_NumSpawns]->CopyOf( pLob );
        m_NumSpawns++;
        return m_ppSpawn[m_NumSpawns-1];

    default: return NULL;
} // switch
} // AddObject

```

Rendern einer Instanz

Damit sind wir schon beim Rendern eines Sektors angekommen. Jedes LOB verfügt ja über seine eigene Render-Methode, und die Aufgabe der Klasse `CSector` ist es nun, diese Render-Methode entsprechend den Wünschen des Aufrufers anzusteuern. Als Parameter kann dabei ein `DWORD` übergeben werden, das den LOB-Typ angibt, der gerendert werden soll. Der Editor kann dabei also ganz gezielt einzelne LOB-Typen rendern lassen, dann gewisse Renderstates umstellen und den nächsten Typ rendern lassen usw. Zu beachten ist hierbei noch, dass selektierte Objekte nicht gerendert werden, weil der Selektions-Buffer seine eigene Render-Methode hat, um die selektierten Objekte noch etwas hervorzuheben.

*Rendern ausge-
wählter LOBs*

```

void CSector::Render(ZFXVertexCacheManager *pVCM,
    DWORD Type, bool bHide) {
    UINT i=0;
    static ZFXCOLOR clrL = { 1.0f, 1.0f, 0.0f, 1.0f };
    static ZFXCOLOR clrS = { 1.0f, 0.0f, 1.0f, 1.0f };
    static ZFXRENDERSTATE sm = RS_SHADE_LINES;

    if ((Type & LOB_MESH) || (Type & LOB_ALL)) {
        for (i=0; i<m_NumMeshs; i++) {

```



```

        if ( !m_ppMesh[i]->IsSelected() )
            m_ppMesh[i]->Render( pVCM, bHide );
    }
} // type

if ((Type & LOB_POLYGON) || (Type & LOB_ALL)) {
    for (i=0; i<m_NumPolys; i++) {
        if ( !m_ppPoly[i]->IsSelected() )
            m_ppPoly[i]->Render( pVCM, bHide );
    }
} // type

if ((Type & LOB_ENTITY) || (Type & LOB_ALL)) {
    for (i=0; i<m_NumEntys; i++) {
        if ( !m_ppEntity[i]->IsSelected() )
            m_ppEntity[i]->Render( pVCM, bHide );
    }
} // type

if ((Type & LOB_SPAWNPOINT) || (Type & LOB_ALL)) {
    for (i=0; i<m_NumSpawns; i++) {
        if ( !m_ppSpawn[i]->IsSelected() )
            m_ppSpawn[i]->Render( pVCM, bHide );
    }
} // type

if ((Type & LOB_LIGHT) || (Type & LOB_ALL)) {
    for (i=0; i<m_NumLights; i++) {
        if ( !m_ppLight[i]->IsSelected() )
            m_ppLight[i]->Render( pVCM, bHide );
    }
} // type

if ((Type & LOB_PORTAL) || (Type & LOB_ALL)) {
    for (i=0; i<m_NumPorts; i++) {
        if ( !m_ppPortal[i]->IsSelected() )
            m_ppPortal[i]->Render( pVCM, bHide );
    }
} // type
} // Render

```

Rendern mit Per-Pixel-Beleuchtung

Jetzt wird es wieder ein bisschen spannend, denn unser Editor bietet auch die Option, ein vereinfachtes Per-Pixel-Lighting im Sektor zu rendern. Vereinfacht deshalb, weil zum Teil auch Polygone von einem Licht auf ihrer Rückseite beleuchtet werden. Allerdings ist diese Darstellung im Editor sehr langsam, weil jedes Polygon hier dynamisch gerendert wird. Man sollte diese Funktionalität des Editors also nur dann nutzen, wenn man die Beleuchtung eines Sektors designt, und auch nur, um sich einen groben Eindruck vom späteren Look des Levels zu verschaffen. Das Rendern dieses

Per-Pixel-Lighting haben wir im Kapitel über die Shader-Technologie schon gesehen. Wir laufen einfach in einer Schleife über alle Lichter des Sektors und berechnen die notwendige Attenuation-Matrix. Damit rendern wir dann die Geometrie, also nur die Polygone und die Meshs.

```
void CSector::RenderLit(ZFXRenderDevice *pDevice,
                      bool bHide) {
    ZFXMatrix mata;
    UINT i=0;

    ZFXVertexCacheManager *pVCM =
        pDevice->GetVertexManager();

    // Loop über alle Lichter im Sektor
    for (UINT l=0; l<m_NumLights; l++) {

        if (m_ppLight[l]->IsHidden()) continue;

        mata = g_CalcTransAttenNoRot(
            m_ppLight[l]->GetPosition(),
            m_ppLight[l]->GetRadius());

        pDevice->SetShaderConstant(SHT_VERTEX, DAT_FLOAT,
                                   20, 4, (void*)&mata);

        pDevice->SetShaderConstant(SHT_PIXEL, DAT_FLOAT,
                                   0, 1, (void*)&m_ppLight[l]->GetColor());

        for (i=0; i<m_NumMeshs; i++) {
            if ( !m_ppMesh[i]->IsSelected() )
                m_ppMesh[i]->Render( pVCM, bHide );
        }
        for (i=0; i<m_NumPolys; i++) {
            if ( !m_ppPoly[i]->IsSelected() )
                m_ppPoly[i]->Render( pVCM, bHide );
        }
        pVCM->ForcedFlushAll();
    }
} // RenderLit
```

Man beachte hierbei, dass der Editor je nach den Wünschen des Level-Designers normalerweise die Methode `CSector::Render` aufruft, um den Sektor mit 100% ambientem Licht zu rendern. Möchte der Level-Designer aber den Level beleuchtet sehen, dann stellt die Klasse `CLevel` den korrekten Wert für ambientes Licht eines Sektors ein und rendert diesen mit der eben genannten Funktion für die Grundschanttierung. Dann stellt er automatisch auf additives Rendering um und ruft die Methode `CSector::RenderLit` auf, um die Per-Pixel-Beleuchtung umzusetzen.



*Steuerung über
CLevel*

Portale und Entities verbinden

New Guy in Town

Abschließend kommen wir noch zu den Methoden der Klasse `CSector`, die uns den Automatismus liefern, den wir benötigen, um sich berührende Portale miteinander zu verbinden bzw. um Entities, die Portale überlappen, mit denselben zu verbinden. Als Erstes brauchen wir eine Methode, mit der wir einem Sektor bekannt machen, dass ein neues Portal in den Level eingefügt wurde. Es ist dabei wichtig, dass wir nicht nur über die neuen Portale desselben Sektors informiert werden, denn der Sinn von Portalen ist ja i.d.R., dass verschiedene Sektoren durch ihre jeweils zugehörigen Portale verbunden werden sollen. Die von uns dazu definierte Methode `CSector::NewPortalInTown` läuft dann in einer Schleife über alle ihre eigenen Portale und prüft, ob diese das neue Portal im Level berühren. Falls ja, dann verbrüdernd sich die beiden Portale miteinander. Weiterhin prüft diese Methode dann auf eine Überlappung einer ihrer Türen mit dem Portal und verbindet die Tür dann mit dem entsprechenden Portal.

Tür in Level einfügen

Es gibt eine ganz analoge Methode `CSector::NewDoorInTown`, die dasselbe für eine neue Tür-Entity im Level macht. Hier prüft der Sektor auch wieder alle seine Portale auf eine Überlappung, um die Tür gegebenenfalls mit dem entsprechenden Portal zu verbinden. Man beachte, dass nur die Entity sich merkt, wenn ein Portal zu ihr gehört. Dem Portal ist das total egal, es hat keine Kenntnis davon, dass es sich mit einer Tür überlappt.

Portale löschen

Wenn ein Portal aus dem Level gelöscht wird, muss jeder Sektor auch dafür sorgen, dass alle Referenzen, die seine eigenen Portale und Tür-Entities auf das gelöschte Portal haben, entfernt werden. Das macht die Methode `CSector::RemovePortalConnections` für eine ganz spezifische Portal-ID, die ja level-weit eindeutig ist.



```
void CSector::NewPortalInTown(CPortal *pPortal) {
    ZFXAabb aabb1 = pPortal->GetAabb();
    ZFXAabb aabb2;

    for (UINT e=0; e<m_NumEntys; e++) {
        aabb2 = m_ppEntity[e]->GetAabb();
        if ( aabb1.Intersects(aabb2) )
            m_ppEntity[e]->ConnectTo( pPortal );
    }

    for (UINT p=0; p<m_NumPorts; p++) {
        aabb2 = m_ppPortal[p]->GetAabb();
        if ( aabb1.Intersects(aabb2) )
            m_ppPortal[p]->SetItsTarget( pPortal );
    }

    } // NewPortalInTown
/*-----*/
```



```

void CSector::NewDoorInTown(CEntity *pDoor) {
    ZFXAabb aabb1 = pDoor->GetAabb();
    ZFXAabb aabb2;

    for (UINT p=0; p<m_NumPorts; p++) {
        aabb2 = m_ppPortal[p]->GetAabb();
        if ( aabb1.Intersects(aabb2) )
            pDoor->ConnectTo( m_ppPortal[p] );
    }
} // NewDoorInTown
/*-----*/

void CSector::RemovePortalConnections(UINT id) {
    for (UINT e=0; e<m_NumEntys; e++)
        m_ppEntity[e]->RemovePortalConnections( id );

    for (UINT p=0; p<m_NumPorts; p++)
        m_ppPortal[p]->DisconnectFrom( id );

} // RemovePortalConnections

```

Ja, damit haben wir es wirklich geschafft. Das Arbeitstier unseres Editors, die Klasse `CSector`, ist damit komplettiert und einsatzbereit. Wir haben jetzt nur noch die Klasse `CLevel` vor uns, die aber nichts weiter macht, als Funktionsaufrufe an alle ihre Instanzen von `CSector` weiterzuleiten, und eigentlich nur zur Gruppierung von Sektoren zu einer Einheit, nämlich einem Level, dient. Als Zweites verbleibt noch der Selektions-Buffer, den wir dann im Anschluss kurz durchgehen werden. Damit sind wir dann bereit, uns die wichtigsten Aspekte der Implementierung des User-Interfaces `CApplication` anzusehen. Also einmal tief durchatmen, und weiter geht's.

Puh!

Alle zusammen: CLevel

Was soll man zu der Klasse `CLevel` Großartiges sagen? Nachdem wir uns bei den Sektoren so sehr ins Zeug gelegt haben, gibt es ja eigentlich nicht mehr viel zu tun. Schauen wir uns daher an, was noch zu tun bleibt.

*Viele Sektoren
sind ein Level*

Klassendeklaration

Wie bereits erwähnt, ist die Klasse `CLevel` nur der Top-Manager, der die ihm unterstehenden Abteilungen, also die Instanzen der Klasse `CSector`, auf Trab hält. Dabei werden fast alle Methoden-Aufrufe dieser Klasse in einer Schleife an alle Sektoren des Levels delegiert. Die anderen Methoden sind überwiegend trivial, und daher zeige ich sie hier nicht. Das Kapitel ist ohnehin schon lang genug, und wir haben noch ein wenig Arbeit vor uns. Hier ist es nur noch wichtig, darauf hinzuweisen, dass beim Editieren eines Levels immer ganz genau ein Sektor aktiv ist. Das ist derjenige, den der

Nichts Neues

Level-Designer aus einer Liste aller Sektoren des Levels in einer entsprechenden Combo-Box im User-Interface auswählt. Die Sektoren werden dabei anhand ihrer Namen unterschieden. Alle anderen Sektoren, die damit inaktiv sind, werden bei Selektionen und beim Einfügen von Objekten einfach ignoriert. Ob sie trotzdem im Editor angezeigt werden, kann der Level-Designer bequem über ein Menü steuern.

Die Deklaration der Klasse sieht wie folgt aus:



```
class CLevel {
public:
    CLevel(void);
    ~CLevel(void);
    void      ClearAll(ZFXSkinManager*);
    void      SetName(const char *ch);
    void      GetName(char *ch);
    bool      HasName(void);
    bool      SaveLevel(const char*,ZFXRenderDevice*);
    bool      LoadLevel(const char*, HWND, HWND,
                        ZFXRenderDevice*);
    void      SaveSkin(ZFXSkinManager*, FILE*, UINT);
    void      LoadSkin(HWND, ZFXSkinManager*, FILE*);
    void      RemovePortalConnections(UINT id);
    void      NewPortalInTown( CPortal* );
    void      NewDoorInTown( CEntity* );
    void      RecalcVertexNormals(void);
    UINT      CreatePortalID(void)
                {m_PID++;return m_PID;}
    UINT      CreateSkinID(ZFXRenderDevice*,
                        const char*,float);
    void      RenderActive(ZFXRenderDevice*,
                        DWORD, bool);
    void      RenderActiveLit(ZFXRenderDevice*,bool);
    void      RenderInActive(ZFXRenderDevice*,
                        DWORD,bool);
    HRESULT   AddSector(const char *ch);
    HRESULT   RemoveSector(const char *ch);
    bool      SectorExists(const char *ch);
    bool      SelectSector(const char *ch);
    CSector*  GetSelectedSector(void)
                { return m_ppSector[m_nActive]; }
    UINT      GetNumSectors(void)
                { return m_NumSectors; }
    void      GetNumLobs(UINT *pVerts, UINT *pIndis,
                        UINT *pPolys, UINT *pMesh,
                        UINT *pPorts, UINT *pLights,
                        UINT *pEntys);
private:
    char      m_chName[MAX_PATH];    // Level-Name
```

```

LPSECTOR *m_ppSector;           // Sektoren
UINT      m_NumSectors;         // Anz. Sektoren
UINT      m_nActive;           // Aktiver Sektor
UINT      m_PID;               // Portal-IDs
void      Init(void);
void      Release(void);
}; // class
typedef class CLevel *LPLEVEL;

```

Laden und speichern

Dass das Laden und Speichern eines Objekts in eine Binärdatei nicht weiter anspruchsvoll ist, wissen wir seit geraumer Zeit. Die Methode zum Speichern eines Levels übernimmt als Parameter zum einen den Namen, den der Level erhalten soll, und zum anderen ein Render-Device, über das er Daten der Texturen abfragen kann.

*Sektoren und
Skins*

Das Datei-Format für eine Level-Datei unseres Editors kennen wir eigentlich auch schon. Zuerst schreibt man die wichtigsten Attribute des Level-Objekts in die Datei. Danach werden nacheinander alle Sektoren gespeichert, und dann werden noch die Skins, also Materialdaten und Texturnamen, in die Datei geschrieben. Fertig. Die Sektoren wiederum speichern nacheinander ihre eigenen Listen mit Level-Objekten (wie beispielsweise Meshs, Polygone usw.) in die Datei, und die einzelnen Objekte schreiben dann ihre Vertex-Daten und relevanten Attribute. Beim Laden eines Levels machen wir dasselbe Spiel, nur eben umgekehrt. Zudem kommt auch hier wieder dazu, dass wir entsprechende Objekte neu anlegen und Skins erzeugen müssen. Die beiden Methoden zum Laden und Speichern eines Levels sehen wie folgt aus:

*Level-Datei-
Format*

```

bool CLevel::SaveLevel(const char *ch,
                      ZFXRenderDevice *pDevice) {
    FILE *pFile=NULL;
    UINT i=0, NumSkins=0;

    if ( !(pFile = fopen(ch, "wb")) )
        return false;

    RecalcVertexNormals();

    NumSkins = pDevice->GetSkinManager()->GetNumSkins();

    fwrite(&m_NumSectors, sizeof(UINT), 1, pFile);
    fwrite(&m_PID,        sizeof(UINT), 1, pFile);
    fwrite(&NumSkins,     sizeof(UINT), 1, pFile);

    for (i=0; i<m_NumSectors; i++) {
        m_ppSector[i]->Save(pFile);
    }
}

```



```

        fflush(pFile);
    }

    for (i=0; i<NumSkins; i++)
        SaveSkin(pDevice->GetSkinManager(), pFile, i);

    fclose(pFile);
    SetName(ch);
    return true;
} // SaveLevel

/*-----*/

bool CLevel::LoadLevel(const char *ch, HWND hcbSector,
                      HWND hcbSkins, ZFXRenderDevice *pDevice)
{
    FILE *pFile=NULL;
    char  buffer[1024];
    int   n=0;
    UINT  NumSkins=0, i=0;

    if ( !(pFile = fopen(ch, "rb")) )
        return false;

    ClearAll( pDevice->GetSkinManager() );
    cbhResetContent(hcbSector);
    cbhResetContent(hcbSkins);

    fread(&m_NumSectors, sizeof(UINT), 1, pFile);
    fread(&m_PID,        sizeof(UINT), 1, pFile);
    fread(&NumSkins,     sizeof(UINT), 1, pFile);

    n = ((int)(((float)m_NumSectors)/50)+1)*50;
    m_ppSector = (LPSECTOR*)malloc(sizeof(LPSECTOR)*n);

    for (i=0; i<m_NumSectors; i++) {
        m_ppSector[i] = new CSector("Empty");
        m_ppSector[i]->Load(pFile);

        m_ppSector[i]->GetName(buffer);
        cbhAddItem(hcbSector, buffer, NULL, true);
        SelectSector(buffer);
    }

    for (i=0; i<NumSkins; i++)
        LoadSkin(hcbSkins, pDevice->GetSkinManager(),
                 pFile);

    fclose(pFile);
    SetName(ch);
    return true;
} // LoadLevel

```

In der engeren Wahl: CSelectionBuffer

Als ich den Editor für dieses Buch entwarf, kam ich natürlich auch irgendwann an den Punkt, an dem es darum ging, wie man Objekte als selektiert markiert. Man muss ja irgendwie dazu in der Lage sein, Teile der Geometrie oder – allgemeiner gesagt – eine beliebige Menge der Level-Objekte in einem Level auszuwählen, um nur genau diese Level-Objekte zu editieren oder gar zu löschen. Ich spielte gedanklich mit Flags ähnlich wie die Versteckflags und überlegte, wie man dann am schnellsten alle LOBs in einem Sektor durchlaufen kann, wenn man Operationen auf den selektierten Objekten durchführen kann. In einem Gespräch mit Oli kam das Thema irgendwie auf die Selektion, und er sagte nur: »*Du verwendest doch sicherlich einen Selection-Buffer dafür?*« In dem Moment sauste mir ein Gedanke von einem Baum und einem Wald durch den Kopf, und ich sagte: »*Ja, klar mach' ich das! Also von jetzt ab wenigstens.*«

Wozu einen Selektions-Buffer?

So viel zu der kleinen Anekdote, wie der Selektions-Buffer in diesen Editor kam. Danke Oli. Die Idee ist natürlich genial einfach, wenn man sie erst mal hatte. Der Selektions-Buffer ist nichts anderes als eine Klasse, in deren Instanzen man Verweise auf beliebige Level-Objekte parken kann. Ein Selektions-Buffer ist damit so ein bisschen etwas wie ein Sektor, nur mit einem kleinen Unterschied: Wenn wir unseren Sektoren LOBs hinzufügen, so erzeugt der Sektor intern eine Kopie der Objekte und speichert diese dann in seinen Listen ab. Unser Selektions-Buffer speichert nur die Adressen der LOBs, die man ihm angibt, also nicht die Objekte selbst, sondern Referenzen auf die Objekte, die aber i. d. R. in den Sektoren selber gespeichert sind.

Was denn sonst?

Wenn der Level-Designer also im Level das Polygon X selektiert, das sich im Sektor Y befindet, so wird der Selektions-Buffer in seiner Liste von selektierten LOBs die Adresse des Polygons X speichern. Und nichts weiter. So weit, so gut. Vielleicht fällt uns nun auch auf, dass unsere LOB-Klassen wie beispielsweise `CPolygon` zwar eine Methode bieten, ein LOB z.B. zu verschieben. Aber ein LOB sitzt ja in einer Liste in einem Sektor, und dieser bietet keine Möglichkeit, die Translation für ein LOB in ihm durchzuführen. Ihr habt es sicherlich schon erraten: Der Selektions-Buffer bietet logischerweise die Schnittstellen zwischen den Editiermethoden der LOB-Klassen und dem User-Interface, von dem aus die Methoden aufgerufen werden. Wenn der Level-Designer nun das Polygon X verschieben möchte, zieht er mit der Maus das im Editor angezeigte Selektionsrechteck an die entsprechende Position. Der Editor berechnet dann den Weg, den die Maus zurückgelegt hat, transformiert diesen Weg zu einem Translationsvektor im 3D-Raum und sendet dem Selektions-Buffer die Anweisung, alle selektierten LOBs mit diesem Vektor zu verschieben. Der Selektions-Buffer durchläuft daraufhin fleißig seine Listen von Adressen selektierter LOBs und leitet den Translationsvektor jeweils an diese weiter. Und schwupps, ist das Polygon X tatsächlich verschoben.

Und so funktioniert's

Ich denke, das Prinzip ist nun klar. Schauen wir uns jetzt die wichtigsten Implementierungen unserer Klasse `CSelectionBuffer` an.

Klassendeklaration

The same old tea

Wenn wir uns nun gleich die Klassendeklaration anschauen, werden wir unweigerlich entdecken, dass uns eine ganze Reihe von Methoden sehr bekannt vorkommen werden. Dabei sind u.a. bekannte Namen wie `Translate`, `Rotate`, `Mirror`, `TransTextureCoords` und andere. Diese Methoden werden letztendlich vom Editor zum Editieren selektierter LOBs verwendet und vom Selektions-Buffer natürlich einfach nur an diese delegiert. Diese Methoden müssen wir uns also hier nicht anschauen. Konzentrieren wir uns lieber auf die wirklich neuen Dinge des Selektions-Buffers.

*Vertices sind
keine LOBs*

Weiter oben hatten wir ja schon gesehen, dass Vertices keine eigene Klasse haben und in dem Sinne keine LOBs sind. Dennoch kann man in unserem Editor auch einzelne Vertices selektieren und verschieben. Dazu brauchen wir aber ein kleines Helferlein, und zwar die Struktur `VERTEX_SB`, in der wir die Adresse eines selektierten Vertex speichern und zusätzlich noch die Adresse des Polygons, zu dem der Vertex gehört. An dieses Polygon muss man beim Editieren eines Vertex z.B. dann ran, wenn man den Vertex verschiebt und die Bounding-Box des Polygons neu berechnet werden muss.

Aber eins nach dem anderen. Hier sind die eben erwähnte Struktur und die Deklaration der Klasse für den Selektions-Buffer:



```
typedef struct VERTEX_SB_TYPE {
    VERTEX    *pV;    // Adresse des Vertex
    LPPOLYGON pPoly; // Adresse seines Polygons
} VERTEX_SB;

class CSelectionBuffer {
public:
    CSelectionBuffer(void);
    ~CSelectionBuffer(void);

    void Reset(void);

    void DestroyWithoutNotify(void);
    void DeleteSelectedObjects(CLevel*);
    void CopySelectedObjects(LPSECTOR);

    bool IsInside(const ZFXVector&, ZFXAXIS);

    UINT GetNumVerts(void) { return m_NumVerts; }
    UINT GetNumLobs(void)  { return m_NumLobs; }
    UINT GetNumObjects(void) { return m_NumLobs+
                                     m_NumVerts; }
```

```

ZFXVector GetCenter(void){return m_Aabb.vcCenter;}

LPPOLYGON      IsPolyQuad(void);
LPLEVELOBJECT  GetHeadPointer(void);

void Render(ZFXRenderDevice*, float, bool, const
            ZFXVector*, const ZFXCOLOR*);
void Translate(const ZFXVector&);

void AddLob(const LPLEVELOBJECT);
void AddVertex(VERTEX*,const LPPOLYGON);
void AddBoundingBox(const ZFXAabb&);

void MergePolysToMesh(LPSECTOR,HWND);
void MergeMeshs(LPSECTOR);
void FragmentMeshs(LPSECTOR);
void InsideOut(void);
void Rotate(ZFXAXIS,float);
void Mirror(ZFXAXIS);
void Snap(void);
void SaveAsPrefab(const char*);

void SetSkinTo(UINT nID);
void TextureRemap(ZFXAXIS);
void TransTextureCoords(float ftU, float ftV,
                        float frU, float frV);
bool GetTextureTrans(float *pftU, float *pftV,
                    float *pfrU, float *pfrV);
private:
    LPLEVELOBJECT *m_ppLob;    // selektierte LOBs
    VERTEX_SB *m_pVerts;      // selektierte Verts
    UINT      m_NumLobs;      // Anz. der LOBs
    UINT      m_NumVerts;     // Anz. der Vertices
    ZFXAabb    m_Aabb;        // AABB
    float      m_fD;          // AABB-Delta

    void Init(void);
    void Release(void);
    void RenderBoundingBox(ZFXRenderDevice*);
    void RenderVertices(ZFXRenderDevice*, float);
}; // class
typedef class CSelectionBuffer *LPSELECTIONBUFFER;

```

Hier sehen wir doch tatsächlich auch eine Render-Methode! Ja klar, wir haben doch auch weiter oben schon gesehen, dass wir selektierte Objekte über die normalen Render-Methoden eines Sektors eben nicht rendern. Über den Selektions-Buffer können wir alle selektierten Objekte dazu auffordern, sich zu rendern. Der Editor wird aber unser Render-Device auffordern, bestimmte Hervorhebungen wie beispielsweise eine knackigere Linienfarbe

*Rendern des
Buffers*

zu aktivieren, bevor er den Selektions-Buffer auffordert, alle selektierten Objekte zu rendern.

Es sei auch noch der Hinweis erlaubt, dass der Selektions-Buffer hier tatsächlich nur eine Liste vom Typ `CLevelObject` für die selektierten LOBs hat, im Gegensatz beispielsweise zu unserer Klasse `CSector`. Da wir aber alle Editier- und Render-Methoden über die Basisklasse erreichen, ist es hier unnötig, die abgeleiteten Klassen zu verwenden.

**Konstruktor und
Destruktor**

Beim Erzeugen und Zerstören eines Objekts gibt es nur eines zu beachten: Wenn man einen Selektions-Buffer freigeben möchte oder ihn einfach nur reinitialisieren will, dann muss man auf alle Fälle beachten, dass man die als selektiert markierten Objekte zunächst wieder als nicht selektiert markiert. Anderenfalls gelten diese Objekte auch weiterhin als vollwertig selektiert und werden entsprechend mit editiert, auch wenn man sie rein optisch im Editor nicht mehr als selektiert erkennt.



```
CSelectionBuffer::CSelectionBuffer(void) {Init(); }
CSelectionBuffer::~CSelectionBuffer(void) {Release();}
```

```
void CSelectionBuffer::Init(void) {
    m_NumVerts    = 0;
    m_NumLobs     = 0;
    m_pVerts      = NULL;
    m_ppLob       = NULL;
    m_fD          = 0.05f;
    memset(&m_Aabb, 0, sizeof(ZFXAabb));
} // Init

/*-----*/

void CSelectionBuffer::Release(void) {
    for (UINT i=0; i<m_NumLobs; i++)
        if (m_ppLob[i]) m_ppLob[i]->DeSelect();
    SAFE_FREE(m_ppLob);
    SAFE_FREE(m_pVerts);
} // Release

/*-----*/

void CSelectionBuffer::Reset(void) {
    Release();
    Init();
    m_Aabb.vcCenter = ZFXVector(0,0,0);
    m_Aabb.vcMax    = ZFXVector(0,0,0);
    m_Aabb.vcMin    = ZFXVector(0,0,0);
} // Init
```


Objekte hinzufügen

Die Strategie der Benutzerführung in unserem Editor dürfte klar sein. Der Level-Designer klickt im aktiven Level ein Objekt an oder zieht mit der Maus ein Selektionsrechteck auf. Über die Klasse `CSector` erhält der Editor dann im Falle des Picking eine Adresse eines gepickten LOBs zurück und fügt diese in seinen Selektions-Buffer ein. Im Falle des Selektionsrechtecks gibt der Editor dem Sektor gleich den Selektions-Buffer mit, und dieser füllt die entsprechenden Adressen in den Buffer. Die folgenden beiden Methoden erledigen diese Aufgabe einmal für alle Arten von Level-Objekten und einmal für Vertices. Dabei dürfen wir nicht vergessen, die Bounding-Box des Selektions-Buffers immer entsprechend zu erweitern. In der Klassen-Deklaration haben wir auch ein Attribut für einen Bounding-Box-Delta-Wert gesehen. Dieser kleine Wert wird immer zu der Bounding-Box addiert, damit die Box ein bisschen größer ist als die tatsächlich selektierten Objekte. So kann man die Box später sauber rendern, ohne die selektierten Objekte zu verdecken.

LOBs und Vertices

Für die selektierten Vertices addieren wir auch noch einen kleinen Spielraum als Bounding-Box um einen Vertex herum.

```
void CSelectionBuffer::AddLob(const LPLEVELOBJECT pLob)
{
    if ( pLob->IsSelected() ) return;

    if ( (m_NumLobs%50) == 0 )
        m_ppLob = (LPLEVELOBJECT*) realloc(
            m_ppLob,
            sizeof(LPLEVELOBJECT)
            * (m_NumLobs+50) );

    m_ppLob[m_NumLobs] = pLob;
    m_NumLobs++;

    // markiere LOB als selektiert
    pLob->Select();

    // AABB neu berechnen
    AddBoundingBox( m_ppLob[m_NumLobs-1]->GetAabb() );
} // AddLob

/*-----*/
```

```
void CSelectionBuffer::AddVertex(VERTEX *pVertex,
                                const LPPOLYGON pPoly)
{
    ZFXAabb aabb;

    for (UINT i=0; i<m_NumVerts; i++) {
```



```

        if (m_pVerts[i].pV == pVertex) return;
    }

    if ( (m_NumVerts%50) == 0 )
        m_pVerts = (VERTEX_SB*) realloc(m_pVerts,
                                         sizeof(VERTEX_SB) *
                                         (m_NumVerts+50));
    m_pVerts[m_NumVerts].pV = pVertex;
    m_pVerts[m_NumVerts].pPoly = pPoly;
    m_NumVerts++;

    aabb.vcMax = ZFXVector(pVertex->x, pVertex->y,
                          pVertex->z);
    aabb.vcMin = ZFXVector(pVertex->x, pVertex->y,
                          pVertex->z);
    aabb.vcCenter = ZFXVector(pVertex->x, pVertex->y,
                             pVertex->z);

    aabb.vcMax += 0.05f;
    aabb.vcMin -= 0.05f;
    AddBoundingBox(aabb);
} // AddVertex

```

Objekte löschen und kopieren

*Mitgefangen,
mitgehangen!*

Was wir einmal in unsrem Selektions-Buffer gespeichert haben, das können wir nach Belieben editieren. Aber wir können es natürlich auch duplizieren oder gar löschen. Wir schauen uns hier zuerst das Löschen von Objekten an. Dazu erhält der Selektions-Buffer als Parameter das CLevel-Objekt. Von diesem lässt er sich den aktiven Sektor geben, in dem gerade editiert wird, und durchläuft dann die Liste seiner Adressen von LOBs, die selektiert sind. Jede einzelne Adresse gibt er dem Sektor mit der Bitte, das Objekt zu löschen. Dieser Bitte kommt der Sektor natürlich nach. Am Ende der Methode setzen wir die Zähler des Buffers auf 0 und führen einen Reset durch. Dabei wird einfach nur der belegte Speicher freigegeben und der Selektions-Buffer neu initialisiert.

**!!
STOP**

In unserem Editor gibt es keine UNDO-Funktion. Das bedeutet, wenn man etwas gelöscht hat, ist es auch unwiederbringlich weg und kann höchstens über einen alten Speicherstand des Levels reproduziert werden. Man sollte also im Editor aufpassen, was für Operationen man durchführt und des Öfteren abspeichern.

**{
LIST**

```

void CSelectionBuffer::DeleteSelectedObjects(
    LPLEVEL pLevel) {
    UINT id=0;
    CSector *pSector = pLevel->GetSelectedSector();

    for (UINT i=0; i<m_NumLobs; i++) {

```

```

// Spezialfall Portal
if (m_ppLob[i]->GetType() == LOB_PORTAL) {
    id = ( (LPPORTAL)m_ppLob[i] )->GetID();
    pLevel->RemovePortalConnections( id );
}
pSector->RemoveObject(m_ppLob[i]);
}
m_NumVerts = 0;
m_NumLobs = 0;
Reset();
} // DeleteSelectedObjects

```

Neben dem wirklichen Löschen der selektierten Objekte gibt es aber auch eine andere Art des Löschens. Wir benötigen auch eine Methode, um den Inhalt des Selektions-Buffers zu löschen, ohne dabei die wirklichen LOBs im Level zu löschen oder darüber zu informieren, dass sie deselektiert wurden. Konkret brauchen wir diese Methode zum Leeren des Selektions-Buffers, wenn wir beispielsweise den Button zum Invertieren der Selektion anklicken. Dann leeren wir den Selektions-Buffer, lassen aber die Flags der selektierten Objekte unangetastet. Nun laufen wir durch den Level, drehen dort die Flags aller LOBs für die Selektion einfach um und fügen die nun selektierten Objekte zum Selektions-Buffer hinzu. Hätten wir den Selektions-Buffer mit seiner normalen `CSelectionBuffer::Reset`-Methode geleert, dann wären damit auch die Flags der selektierten Objekt zurückgesetzt worden.

*Selektions-Buffer
leeren*

```

void CSelectionBuffer::DestroyWithoutNotify(void) {
    SAFE_FREE(m_ppLob);
    SAFE_FREE(m_pVerts);
    Init();
} // DestroyWithoutNotify

```



Nun kommen wir zu einer sehr hilfreichen Methode. Wir können selektierte Objekte duplizieren, indem wir einfach Kopien dieser Objekte anfertigen und an einer leicht versetzten Position im Editor einfügen. Mit Hilfe unserer in allen LOBs implementierten Funktion `CopyOf()` ist das natürlich ein Klacks. Wir müssen nur den Selektions-Buffer bei dieser Operation auch korrekt behandeln. Wir haben eine Menge an Objekten selektiert und möchten diese kopieren. Am Ende der Operation sollen alle als Kopien eingefügten Objekte selektiert sein, damit man sie gleich weiterbearbeiten kann. Wir laufen also durch die Liste aller selektierten LOBs und fertigen in jedem Schleifendurchlauf eine Kopie des betrachteten Objekts an. Zuvor deselektieren wir das Original-Objekt jedoch. Dann fügen wir die Kopie in den aktiven Sektor ein und markieren sie zugleich als selektiert. Mit der Adresse, an der die Kopie im Sektor eingefügt wurde, überschreiben wir die im Selektions-Buffer gespeicherte Adresse des Original-Objekts, und schon haben wir das erreicht, was wir erreichen wollten. Hier ist die Methode `-CSelectionBuffer::CopySelectedObjects`:

Replikatoren



```

void CSelectionBuffer::CopySelectedObjects(
    LPSECTOR pSector) {
    DWORD    Type=LOB_NONE;
    CPolygon  Poly;
    CPolymesh Mesh;
    CLight    Light;
    ZFXVector vcT(0.2f,0.2f,0.2f);

    // deselektiere Objekte auch wieder!!!
    for (UINT i=0; i<m_NumLobs; i++) {
        m_ppLob[i]->DeSelect();

        switch ( m_ppLob[i]->GetType() ) {
            case LOB_POLYGON:
                Poly.CopyOf( (LPPOLYGON)m_ppLob[i] );
                Poly.Translate(vcT);
                m_ppLob[i] = (LPPOLYGON)
                    pSector->AddObject(Poly);
                break;
            case LOB_MESH:
                Mesh.CopyOf( (LPPOLYMESH)m_ppLob[i] );
                Mesh.Translate(vcT);
                m_ppLob[i] = (LPPOLYMESH)
                    pSector->AddObject(Mesh);
                break;
            case LOB_LIGHT:
                Light.CopyOf( (LPLIGHT)m_ppLob[i] );
                Light.Translate(vcT);
                m_ppLob[i] = (LPLIGHT)
                    pSector->AddObject(Light);
                break;
            default: break;
        } // switch
        m_ppLob[i]->Select();
    } // for
    m_Aabb.vcCenter += vcT;
    m_Aabb.vcMax     += vcT;
    m_Aabb.vcMin     += vcT;
    } // CopySelectedObjects

```

**Translations-
vektor**

In dem Code oben könnt ihr auch gut erkennen, dass wir die Kopien der Objekte jeweils um einen Translationsvektor verschieben. Das hat einfach optische Gründe, denn wenn man eine Kopie direkt an der Originalposition einfügen würde, dann würde man das im Editor nicht sehen können und hätte das Gefühl, dass der Button zum Kopieren nicht funktioniert. Schlimmstenfalls würde man gleich ein paar Mal mehr draufklicken und so unbewusst multiple Kopien erzeugen, allerdings alle auf derselben Stelle.

Wir lassen es auch nur zu, ganz bestimmte LOB-Typen zu kopieren, und zwar Polygone, Meshs und Lichtquellen. Insbesondere kann man keine Entities und Portale kopieren. Das hat den Sinn, dass diese bestimmten Verknüpfungen untereinander unterliegen und man diese Objekte nicht einfach so duplizieren sollte. Das mag zunächst als große Restriktion erscheinen, wenn man sich überlegt, dass man eine sehr detaillierte Tür als Entity modelliert hat und diese nun nicht duplizieren können soll. Aber man sollte hier zwischen dem Konzept der Entity und der Geometrie differenzieren. Man kann ja ganz einfach die entsprechende Entity selektieren, alle unselektierten Objekte verstecken, in den Polygon-Selektionsmodus umschalten und alle Polygone selektieren. So hat man die gesamte Geometrie ausgewählt, die zu dem Entity-Objekt gehört, und kann diese ganz normal kopieren. Mit zwei weiteren Mausklicks macht man die kopierten Polygone zu einem Mesh und dann zu einer Entity. Und fertig ist die Kopie.

*Nur bestimmte
LOBs kopieren*

Polys zu Meshs, Meshs zu Polys

Jetzt haben wir schon so viel darüber gehört, wie wir mit Polymeshs arbeiten können und dass wir sie aus einzelnen Polygonen erzeugen. Doch wie wir das im Editor machen, wissen wir immer noch nicht, auch wenn wir schon alle Lowlevel-Methoden haben, um Polygone zu Meshs hinzuzufügen. Dabei ist der Job ganz einfach erledigt: Wir selektieren einfach eine beliebige Menge an Polygonen und klicken im User-Interface auf den Button, der die selektierten Polygone zu einem einzigen Mesh verschmilzt. Diese Funktionalität implementiert also auch der Selektions-Buffer. Dazu legt er ein neues Mesh-Objekt an und durchläuft die Liste der Adressen selektierter LOBs. Findet er dort ein Polygon, dann fügt er eine Kopie dieses Polygons zu dem neuen Mesh-Objekt hinzu und löscht das Polygon aus dem Level. Ist er einmal durch alle selektierten LOBs gelaufen, fügt er das neue Mesh-Objekt einfach zu dem Level hinzu. So einfach ist das.

*Polygone werden
zu Meshs*

```
void CSelectionBuffer::MergePolysToMesh(LPSECTOR
                                     pSector, HWND hWnd) {
    LPPOLYGON pPoly=NULL;
    CPolymesh Mesh;
    bool      bAsked=false;
    bool      bUseEm=false;

    if (m_NumLobs==0) return;

    for (UINT i=0; i<m_NumLobs; i++) {

        if ( m_ppLob[i]->GetType() != LOB_POLYGON )
            continue;
        else pPoly = (LPPOLYGON) m_ppLob[i];

        if ( pPoly->IsPartOfMesh() ) {
            if (bAsked && !bUseEm) continue;

```



```

        if (!bAsked) {
            if ( MessageBox(hWnd,
                "At least one selected polygon is
                part of a mesh. Merge them also?",
                "PanEdit - Warning",
                MB_YESNO | MB_ICONQUESTION)
                == IDYES) bUseEm=true;
            bAsked=true;
        }
    }
    Mesh.AddPolygon( *pPoly );
    pSector->RemoveObject(pPoly);
}
pSector->AddObject(Mesh);
m_NumLobs = 0;
Reset();
} // MergePolysToMesh

```

**Polygone aus
bestehenden
Meshs**

Aber natürlich haben wir auch hier einen Spezialfall zu bedenken: Es kann sein, dass der Level-Designer auch zufällig oder absichtlich Polygone selektiert hat, die bereits Teil eines bestehenden Meshs sind. Sobald unsere Methode `CSelectionBuffer::MergePolysToMesh` ein solches Polygon findet, erzeugt sie eine WinAPI-Message-Box, die den Level-Designer darüber informiert. Hier kann er durch Anklicken des YES- bzw. NO-Buttons entscheiden, ob diese Polygone aus ihrer bisherigen Gruppe herausgelöst und zu der neuen hinzugefügt werden sollen. Die einmal getroffene Entscheidung gilt für alle Polygone, die Teil eines Meshs sind.

**Fragmentieren
eines Meshs**

Nun gehen wir den umgekehrten Weg. Natürlich kann man auch ein oder mehrere Polymeshs selektieren und die Gruppierung zerstören, ohne jedoch die Geometrie zu löschen. Vereinfacht ausgedrückt, löschen wir die Mesh-Objekte und behalten die Geometrie in Form freier, ungruppierter Polygone im Level. Der Quelltext hierzu ist sogar noch einfacher als der für das Gruppieren von Polygonen, da wir keine Spezialfälle zu beachten haben. Wir laufen einfach durch alle selektierten LOBs im Selektions-Buffer und überspringen alle, die kein Mesh sind. Wenn wir ein Mesh finden, dann laufen wir in einer Schleife über seine Polygone und fügen jeweils eine Kopie eines Polygons in den Level ein. Dann löschen wir das Mesh-Objekt aus dem Level.



```

void CSelectionBuffer::FragmentMeshs(LPSECTOR pSector) {
    LPPOLYMESH pMesh=NULL;
    LPPOLYGON pPoly=NULL;
    UINT n=0;

    for (UINT i=0; i<m_NumLobs; i++) {
        if ( m_ppLob[i]->GetType() != LOB_MESH ) continue;
        else pMesh = (LPPOLYMESH) m_ppLob[i];
    }
}

```

```

n = pMesh->GetNumPolys();

for (UINT j=0; j<pMesh->GetNumPolys(); j++) {
    pPoly = pMesh->GetPolygon(j);
    pSector->AddObject( *pPoly );
}
pSector->RemoveObject(pMesh);
}

m_NumLobs = 0;
Reset();
} // FragmentMeshs

```

Ein PreFabs speichern

Das Laden eines PreFabs haben wir ja direkt in der Klasse `CSector` implementiert, weil dort auch die Geometrie eingefügt wird. Als PreFab kann man bei uns lediglich selektierte Polygone abspeichern, aber natürlich auch Polygone aus einem oder mehreren selektierten Meshs. Wir speichern dort also keine Portale, Lichtquelle usw. mit ab, selbst wenn diese selektiert sein sollten. Unser PreFab ist damit schon ein spezielles Modellformat, vergleichbar mit dem *.x- oder dem *.3ds-Format in Bezug darauf, dass diese Formate auch die Daten über Vertices und Polygone eines 3D-Modells enthalten. Wir sehen hier also eine sehr einfache Form, wie wir geometrische Daten unseres Level-Editors auch verwenden können. Ich fordere euch hiermit nochmals auf, Export-Funktionen für euer bevorzugtes Modellformat für den Editor zu schreiben.

*Modellformat-
Exporter
schreiben*

Sobald Geometrie im Selektions-Buffer vorhanden ist, könnt ihr im Editor die Option zum Speichern als PreFab nutzen. Der Editor informiert den Selektions-Buffer dann über den Namen, mit dem das PreFab gespeichert werden soll. Die Methode `CSelectionBuffer::SaveAsPreFab` öffnet dann eine entsprechende Datei und beginnt die Speicher-Arbeit. Dazu durchläuft der Selektions-Buffer alle in ihm referenzierten LOBs. Zuerst zählt er die selektierten Polygone und Polygone selektierter Meshs, denn diese Information kommt als Erste in die Datei. Danach durchläuft er seine Objekte nochmals und speichert dabei alle Polygone und Polygone aus Meshs in die Datei. Die Methode `CPolymesh::SavePurePolys` haben wir ja weiter oben schon besprochen. Wichtig dabei ist, dass zu jedem Polygon der Name der Textur gespeichert wird, die das Polygon verwendet. Die eben erwähnte Methode zum Speichern der Polygone eines Meshs funktioniert ja auf jedem Polygon dort genauso wie hier das Speichern eines ungruppierten Polygons. Wir erinnern uns ja noch daran, dass wir den Namen der Textur aus der entsprechenden Combo-Box des User-Interfaces ziehen.

*Und so
funktioniert's*



```

void CSelectionBuffer::SaveAsPrefab(const char *ch) {
    UINT nNumPolys=0;
    UINT i=0, n=0, SkinID=0;
    char buffer[2048];

    FILE *pPrefab = fopen(ch, "wb");

    for (i=0; i<m_NumLobs; i++) {
        if (m_ppLob[i]->GetType() == LOB_POLYGON)
            nNumPolys++;
        else if (m_ppLob[i]->GetType() == LOB_MESH)
            nNumPolys += m_ppLob[i]->GetNumPolys();
    }

    fwrite(&nNumPolys, sizeof(UINT), 1, pPrefab);

    for (i=0; i<m_NumLobs; i++) {
        if (m_ppLob[i]->GetType() == LOB_MESH) {
            ((LPPOLYMESH)m_ppLob[i])->
                SavePurePolys(pPrefab);
        }

        else if (m_ppLob[i]->GetType() == LOB_POLYGON) {
            SkinID = ((LPPOLYGON)m_ppLob[i])->GetSkinID();
            SendMessage(CApplication::m_G.hGUI[CSK_LIST],
                CB_GETLBTEXT, SkinID,
                (LPARAM)(LPCSTR)buffer);
            n = strlen(buffer);
            fwrite(&n, sizeof(UINT), 1, pPrefab);
            fwrite(buffer, n, 1, pPrefab);

            ((LPPOLYGON)m_ppLob[i])->Save(pPrefab);
        }
    }
    fclose(pPrefab);
} // SetSkinTo

```

Weitere Modell- formate

Zugegebenermaßen ist dies ein sehr einfaches Modellformat. Normalerweise speichert man in anderen Formaten zuerst die gesamte Liste an Vertices, dann die Triangulierung einzelner Faces über Indices usw. Wenn man ein solches Modellformat mit unserem Editor exportieren möchte, dann muss man die Vertex-Daten aus den einzelnen Polygonen herausziehen und speichern, danach die Indices und dann die Texturkoordinaten etc. Das ist sicherlich ein bisschen mehr Aufwand, aber dann habt ihr mit diesem Tool nicht nur einen Level-Editor, sondern auch gleich einen Low-Polygon-3D-Modell-Editor. Was will man mehr?

Und wech

Das war schon alles, was es über den Selektions-Buffer zu sagen gibt. Und ob ihr es glaubt oder nicht, der Code für unseren *full-blown* Level-Editor ist

nun absolut komplett fix und fertig. Uns fehlt nur noch ein bisschen Code für das Graphics-User-Interface, und das ist alles langweiliger WinAPI-Kram: das Erzeugen von Buttons, das Abfragen, ob auf einen Button geklickt wurde, und das Aufrufen der zu dem Button gehörigen Methode beispielsweise zum Kopieren der selektierten Elemente. Aber es gibt dort auch ein paar interessante Aspekte, die wir uns im folgenden Abschnitt noch kurz anschauen.

14.4 Ausgewählte Aspekte des GUI

Das gesamte Projekt für den Editor *PanBox Edit* verfügt natürlich noch über etwas mehr als nur die hier gezeigten Klassen und die gleich noch kurz vorgestellte Klasse `CApplication`. Allerdings sind diese Dinge weder sehr umfangreich noch sehr komplex. Das Projekt enthält zusätzlich noch die folgenden Dinge:

Projekt-Überblick

- ➔ die Klasse `CFile` zur Kapselung des WinAPI File-Dialogs
- ➔ die Klasse `CToolBar` zur Kapselung einer WinAPI-Toolbar
- ➔ eine Combo-Box mit Helper-Funktionen zur Kapselung der WinAPI-Combo-Box
- ➔ Dialog-Callbacks, also CB-Funktionen für spezifische Dialoge des Editors

Die jeweiligen Implementierungen sind immer nur ein paar Zeilen lang und für WinAPI-Programmierer keine große Hürde. Die Implementierungen findet ihr natürlich komplett auf der CD-ROM.

Verbohrte C++-Cracks mögen jetzt gleich zu einem Strick greifen, denn das Verwenden globaler Variablen ist dort absolut verpönt. Natürlich kann man auf diese auch durch ein vernünftiges Design an vielen Stellen verzichten. Doch es gibt auch Situationen, in denen man sich das Leben ohne globale Variablen unnötig schwer macht, ohne dass man die Sicherheit im Programmablauf eines geschlossenen Projekts verbessert, beispielsweise bei der Interaktion von Callback-Methoden mit Instanzen von Klassen. Doch auch C++-Programmierer kennen und lieben globale Variablen, auch wenn man sie dort unter dem Deckmantel eines `static`-Keywords versteckt. Im Verlauf dieses Kapitels haben wir schon das Öffnen ein Attribut `CApplication::m_g` gesehen. Und nun lüften wir das Geheimnis: Dieses Attribut ist ein `static`-Attribut vom Typ der folgenden Struktur:

*Globale Variable
ohne globale
Variablen*



```
typedef struct GLOBALS_TYPE {
    HINSTANCE      hInst;
    HWND           hWnd;
    HWND           hWnd3D[4];
    HWND           hGUI[100];
    HWND           hTexDlg;
    bool           bChanged;
    bool           bNeedRedraw[4];
    FILE           *pLog;
    CFile           FileDlg;
    CLevel          Level;
    CSelectionBuffer SelBuf;
    char            Dir[MAX_PATH];
} GLOBALS;
```

Alles, was wichtig ist

In dieser oben gezeigten GLOBALS Struktur findet sich alles, was an anderen Stellen im Projekt einfach erreichbar sein muss, wenn beispielsweise die Callback-Funktionen an ein Steuerelement heranmüssen oder Dinge vom Level abfragen wollen. Tabelle 14.1 zeigt, welche Bedeutung die Felder der Struktur haben.

Tabelle 14.1:
Felder der Struktur
GLOBALS

Feld	Bedeutung
hInst	Instance-Handle der Anwendung
hWnd	Window-Handle des Hauptfensters
hWnd3D[4]	Handles der vier Child-Views mit Direct3D-Grafik
hGUI[100]	Handle der WinAPI-Steuerelemente
hTexDlg	Handle des nicht-modalen Texturkoordinaten-Dialogs
bChanged	Level geändert; Speicherung erforderlich
bNeedRedraw[4]	Child-Windows muss neu gerendert werden
pLog	Log-File für Fehlermeldungen
FileDlg	Kapselung des WinAPI-File-Dialogs
Level	aktuell geladener Level
SelBuf	Selektions-Buffer des Editors
Dir[MAX_PATH]	Pfad der Anwendung

Klassendeklaration

Weniger ist mehr

In diesem Abschnitt sehen wir uns die Klasse CApplication an, die das Interface zwischen unseren Lowlevel-Klassen und dem Level-Designer am Bild-

schirm darstellt. Erstaunlicherweise ist die Klasse, von ihren Methoden her betrachtet, nicht sonderlich umfangreich. Warum sollte sie es auch sein? Es geht ja hauptsächlich um das Abfangen von Events, die an WinAPI-Steuer-elementen auftreten, und um eine entsprechende Reaktion darauf über die Lowlevel-Klassen, die wir weiter oben in diesem Kapitel implementiert haben. Im Folgenden liste ich die Klassendeklaration auf. Auch wenn ich nicht die gesamte Implementierung der Klasse hier zeige, werdet ihr anhand der Erklärung gewisser Methoden und der Kommentare den nicht erläuterten Quelltext auf der CD-ROM schnell durchschauen.

```
class CApplication {
public:
    CApplication(void) { }
    virtual ~CApplication(void) { }

    HRESULT Init(HWND, HINSTANCE);
    void    Update(void);
    void    RenderTick(void);
    void    Release(void);

    void    Done(void)      { m_bDone = true; }
    bool    IsDone(void)    { return m_bDone; }
    void    Active(bool b) { m_bActive = b; }
    bool    IsActive(void) { return m_bActive; }

    void    InvalidateView(int);
    void    InvalidateAllViews(void);

    bool    MsgProcMenu(WORD);
    bool    MsgProcGUI(WORD);
    bool    MsgProcToolbar(WORD wCmd)
            { return m_Toolbar.MsgProc(wCmd); }

    LPZFXRENDERDEVICE GetRenderDevice(void)
            { return m_pRDevice; }

    void    OnViewPerspective(void);
    void    OnViewOrthogonal(UCHAR);
    bool    OnClose(void);
    void    OnDeleteKey(void);
    void    OnMouseMove(WPARAM, LPARAM);
    void    OnMouseClick(bool, WPARAM, LPARAM);
    void    OnSelBufChange(void);

    ZFXSkinManager* GetSkinMan(void)
            { return m_pRDevice->GetSkinManager(); }

    CToolbar* GetToolbar(void) { return &m_Toolbar; }
    bool    CheckForcedSave(void);
}
```



```

bool        MouseInside(HWND, POINT*);

static GLOBALS m_G;

private:
    DWORD        m_OldTime;        // Frame Counter
    DWORD        m_NewTime;        // Frame Counter
    float        m_dt;              // vergang. Zeit
    LPZFXRENDERER m_pRenderer;
    LPZFXRENDERDEVICE m_pRDevice;
    CToolBar      m_Toolbar;
    HMENU         m_hm;
    POINT         m_ptCursorNew;    // Maus-Position
    POINT         m_ptCursorOld;    // Alt&neu-Frame
    int           m_CurView;        // Viewport
    UINT          m_nFontID;        // Font
    bool          m_bDone;
    bool          m_bActive;
    UINT          m_nZFXFont;

    // User Settings für den Editor
    bool          m_bDrawLight;    // Licht rendern
    bool          m_b3DHide;        // Hide in 3DView
    bool          m_bDrawGrid[2];
    bool          m_bDrawPoints[2];
    bool          m_bDrawWired[2];
    bool          m_bDrawInact[2];
    bool          m_bDrawInactW;
    bool          m_bSnap;
    UCHAR         m_bIsSelecting;
    UCHAR         m_bIsMovingSelection;
    UCHAR         m_bIsCreating;

    RECT          m_rcOrthoSel;
    ZFXVector     m_vcSel[2];
    ZFXAXIS       m_Axis[3];
    ZFXVector     m_vcEndMove;
    ZFXVector     m_vcStartMove;

    SELECTMODE    m_SelectMode;
    TOOL          m_CurrentTool;

    // View-Transformation Kram
    ZFXMatrix     m_World[4];
    ZFXVector     m_vcFix;
    float         m_fRotX;
    float         m_fRotY;
    float         m_fDistOrtho;
    float         m_fScale[3];
    float         m_fPosX[4];

```

```

float          m_fPosY[4];
float          m_fPosZ[4];

VERTEX        m_vVerts[256];
UCHAR         m_NumVerts;

// Shader-IDs
UINT          m_BaseVSH;
UINT          m_OmniVSH;
UINT          m_BasePSH;
UINT          m_OmniPSH;

// private Methoden
HRESULT        InitRenderer(void);
HRESULT        InitShaders(void);
void          CreateGUI(void);
void          UpdateCursor(void);
void          SetSelectionMode(SELECTMODE);
bool          ChangeCheckState(UINT MenuID);
void          OrthoSelection(bool,POINT);
void          CheckTbState(UINT,UINT,
                           SELECTMODE);
void          RenderOrthoGrid(UCHAR n);
void          RenderPerspGrid(void);
void          RenderPolyUnderConstr(UINT);
void          RenderSelectionRect(void);
void          RenderInActive(bool);
void          RenderGeometry(bool,float);
void          RenderPortals(bool);
void          RenderLights(bool);
void          RenderEntities(bool);
void          RenderSpawns(bool);

bool          InvalidateSelectionRect(bool);
void          OnPolyCreationEvent(bool bEnd,
                                   const ZFXVector*);
}; // class

```

Wichtige Attribute

Ein paar Attribute der Klasse sind nicht auf Anhieb verständlich, aber eigentlich extrem simpel in ihrer Anwendung. Sie steuern wesentliche Abläufe und Zustände des Editors und sollen daher hier in einer Tabelle erläutert werden.

Auf einen Blick

Tabelle 14.2:
Wichtige Attribute
der Klasse
CApplication

Datentyp	Attribut	Bedeutung
UCHAR	m_blsSelecting	Steht diese Variable auf dem Wert 255, so hat sie keine Bedeutung. Hat sie einen Wert von 0 bis 2, so sagt sie aus, dass der Benutzer in den Child-Windows, die von oben nach unten und von links nach rechts durchnummeriert sind, gerade ein Selektionsrechteck aufzieht. Der Wert gibt dabei das Child-Window an, in dem dieser Prozess begonnen wurde.
UCHAR	m_blsMovingSelection	Analog zu dem oben Stehenden, nur dass es sich hierbei um das Verschieben eines bereits angelegten Selektionsrechtecks handelt.
UCHAR	m_blsCreating	Analog zu den oben Stehenden, nur dass der Benutzer gerade dabei ist, Vertices für ein neues Polygon durch Klicken mit der linken Maustaste zu setzen.
RECT	m_rcOrthoSel	Diese WinAPI-Struktur enthält die zweidimensionalen Abmessungen des Selektionsrechtecks, falls es vorhanden ist, ansonsten jeweils den Wert -1 für alle Felder.
ZFXVector	m_vcStartMove	Startpunkt der Verschiebung des Selektionsrechtecks im 3D-Raum.
ZFXVector	m_vcEndMove	Endpunkt der Verschiebung des Selektionsrechtecks im 3D-Raum.
VERTEX	m_vVerts[256]	Temporärer Speicher für Vertices eines Polygons, das gerade erstellt wird.
UCHAR	m_NumVerts	Anzahl der vorhandenen Vertices im obigen Array.
POINT	m_ptCursorOld	Position des Maus-Cursors im letzten Frame.
POINT	m_ptCursorNEW	Position des Maus-Cursors im aktuellen Frame.
int	m_CurView	Gibt an, in welchem Child-Window sich der Cursor befindet [0, ..., 3], von oben nach unten bzw. von links nach rechts durchnummeriert.

Datentyp	Attribut	Bedeutung
TOOL	m_CurrentTool	Beschreibt das aktuell aktive Tool im Editor. Bisher gibt es nur die beiden Tools TL_SELECT, so dass die Maus zur Markierung von Objekten verwendet werden kann, bzw. TL_POLYGON, so dass man mit der Maus Vertices für ein neues Polygon in den orthogonalen Ansichten setzen kann.

Tabelle 14.2:
Wichtige Attribute
der Klasse
CApplication
(Forts.)

Update-Methode

Die Performance in unserem Editor ist ein kritisches Element, da wir hier sehr unoptimiert rendern müssen, weil die gesamte Geometrie dynamisch ist und sich jederzeit ändern kann – nicht zuletzt auch deshalb, weil wir eigentlich alles viermal rendern: in drei orthogonalen Ansichten und einer perspektivischen. Die schlechte Performance würde sich zum Beispiel in Verzögerungen bei Reaktionen auf Button-Klicks äußern oder durch Ruckeln, wenn man Objekte verschiebt und editiert. Allerdings können wir das effektiv bekämpfen. Im Gegensatz zu einer interaktiven Darstellung brauchen wir kein megaschnelles Update der Back-Buffer so oft wie möglich. Im Gegenteil, wir können es hier ruhig angehen lassen. Wir brauchen unsere Ansichten nur immer dann neu zu rendern, wenn sich etwas verändert hat. Dazu haben wir ja auch Attribute in unserer Klasse, die das angeben. Zudem führen wir Folgendes ein: Wir zählen die Frames unseres Editors und rendern einfach in einem beliebigen Abstand. Im folgenden Code habe ich 120 Frames als beliebigen Wert eingesetzt. Schon steigt die Performance unseres Editors um ein Vielfaches an, und wir können selbst große Level *smooth* editieren.

Render mich nicht

```
void CApplication::Update(void) {
    static float fCount = 0.0f;

    UpdateCursor();

    if (fCount > 120.0f) {
        m_OldTime = timeGetTime();
        RenderTick();
        fCount = 0.0f;
        m_NewTime = timeGetTime();
        m_dt = ((float)(m_NewTime-m_OldTime))/1000.0f;
    }
    else fCount += 1.0f;
} // Update
```



Rendern Die Methode `CApplication::RenderTick` ruft einfach nur die Methoden zum Rendern der entsprechenden Objekte auf, wobei wir nach Geometrie, Portalen, Lichtquellen usw. mit jeweils eigenen Render-Methoden in dieser Klasse unterscheiden. Ich schreibe deshalb »einfach«, weil wir vor dem Weiterleiten der Render-Aufrufe an die entsprechenden Methoden der LOB-Klassen bestimmte Render-States wie beispielsweise auch die Linienfarben der Drahtgitter-Ansicht einstellen, so dass Meshs in blauen Linien gerendert werden, Polygone hingegen in hellgrauen usw.

Ein Polygon erstellen

**Jeder Klick ein
Punkt des
Polygons**

Das Erstellen eines Polygons realisieren wir so, dass der Level-Designer einen Button anklickt, um anzuzeigen, dass er ein Polygon erstellen möchte. Dann wird das Attribut `m_CurrentTool` auf `TL_POLYGON` gestellt. Wenn nun in unserer WinAPI-Applikation das Event eines Mausklicks in einer der orthogonalen (zweidimensionalen) Ansichten auftritt und wir uns auch im Polygonerstellungsmodus `TL_POLYGON` befinden, dann wird die Methode `CApplication::OnPolyCreationEvent` aufgerufen. Bei einem Klick mit der rechten Maustaste wird der erste Parameter auf `true` gesetzt, da dies die Erstellung eines Polygons beendet. Ansonsten ist der Parameter `false`. In beiden Fällen wird für den zweiten Parameter die Position des Mausklicks, umgerechnet in den 3D-Raum mit der fehlenden Koordinate auf dem Wert 0, übergeben.

Wenn der Level-Designer nun also mit der linken Maustaste klickt, fügen wir einen Vertex zur temporären Liste für das neue Polygon hinzu. Klickt er mit der rechten Maustaste, dann beenden wir den Erstellungsprozess, erzeugen ein neues Polygon-Objekt aus der Vertexliste, berechnen seine Texturkoordinaten mit planarem Texture-Mapping auf der entsprechenden Achse und fügen das neue Polygon in den Level ein. *That's no kind of magic.*



```
void CApplication::OnPolyCreationEvent(bool bEnd,
                                       const ZFXVector *pvc) {
    LPPOLYGON pPoly=NULL;
    CPolygon  NewPoly;
    CSector   *pSector=NULL;

    // Merke in welchem Child-Window der Prozess startet
    if (m_NumVerts == 0) m_bIsCreating = m_CurView;

    // Ende des Erstellens
    if (bEnd) {
        // mindestens dreiseitiges Polygon?
        if (m_NumVerts>2) {
            // Erstelle neues Polygon
            NewPoly.Reset();
            NewPoly.SetVertices(m_vVerts, m_NumVerts);
```



```

        NewPoly.Triangulate();
        NewPoly.CalcTextureCoords(
            m_Axis[m_CurView],NULL);
        pSector = m_G.Level.GetSelectedSector();
        pPoly = (LPPOLYGON)pSector->AddObject(NewPoly);

        m_G.SelBuf.Reset();
        m_G.SelBuf.AddLob(pPoly);
        m_G.bChanged = true;
        OnSelBufChange();
        m_bIsCreating = 255;
    }

    // simulierte Button-Klicks, um Selektionsmodus
    // auf Polygone zu stellen
    SendMessage(m_G.hGUI[CSM_POLYGON],BM_CLICK,0,0);
    SendMessage(m_G.hGUI[CT_SEL],BM_CLICK,0,0);
    m_CurrentTool = TL_SELECT;
    m_NumVerts = 0;
}
// Erstellungsprozess
else if (!bEnd) {
    // zu viele Vertices
    if (m_NumVerts > 255) return;

    // Koordinaten des Mausklicks
    m_vVerts[m_NumVerts].x = pvc->x;
    m_vVerts[m_NumVerts].y = pvc->y;
    m_vVerts[m_NumVerts].z = pvc->z;
    m_vVerts[m_NumVerts].tu = 0.0f;
    m_vVerts[m_NumVerts].tv = 0.0f;
    m_NumVerts++;
}
} // OnPolyCreationEvent

```

Nun können wir tatsächlich damit anfangen, beliebige Geometrie in den Level einzufügen. Mehr als das Erzeugen von Polygonen brauchen wir dazu nicht, und diese Fähigkeit haben wir nun. Ich erwähne aber gern noch einmal, dass wir auch die Klasse `CGeometry` haben, die über statische Methoden bestimmte geometrische Objekte als Liste von Polygonen oder als Mesh erzeugt. Auch wenn wir dort bisher nur zwei Formen haben, so kann man sehr schön sehen, wie man das Einfügen eines Würfels, einer Kugel, eines Torus und jeder anderen beliebigen geometrischen Form vereinfachen kann, ohne dass man diese Polygon für Polygon erstellen muss. Das wäre doch ziemlich schwierig. Das Fundament ist also da, und ihr könnt nun entsprechend euren Bedürfnissen weiterbauen.

Level-Bau

Und Schluss! Es gäbe sicherlich noch das eine oder andere Interessante zu zeigen, aber irgendwann muss auch mal Schluss sein. Das Kapitel platzt jetzt schon aus allen Nähten. Mit allem, was ich hier gezeigt habe, solltet ihr den Editor vom Quelltext her verstehen und ihn entsprechend erweitern können.

14.5 Ein Blick zurück, zwei Schritt nach vorn

Disengage Warp-Engines

Hui, das war eine rasante Fahrt. Entwurf, Design und Implementierung eines vollwertigen Low-Polygon-Editors auf nur knapp 120 Seiten. Da zeigt sich mal wieder, dass alles nur so schwer ist, wie man es sich selbst macht. Das Handling unseres Editors mag sich zwar von dem anderer Level-Editoren, insbesondere solcher mit CSG-Ansatz, dramatisch unterscheiden, aber trotzdem können wir damit Level und 3D-Modelle bauen. Das Spiel, das wir im nächsten Kapitel basierend auf diesen Leveln programmieren werden, spricht für sich.

Viel Potenzial

Ich will nicht schon wieder auf das Potenzial verweisen, das in unserem Editor schlummert. Ich denke, ihr habt schon Tausende von Ideen im Kopf, wie man den Editor erweitern und verbessern kann. Bevor ihr nun das nächste Kapitel durchhackert, empfehle ich euch, wenigstens einen kurzen Blick auf die vollständige Implementierung von `CApplication` zu werfen und deren Arbeitsweise nachzuvollziehen. Vorbei ist damit die Zeit, in der man für die Definition eines Levels auf primitive Textdateien zurückgriff, in die man per Hand Koordinaten eintrug, oder in der man versuchte, aus Bilddateien geometrische Strukturen zu extrahieren. In einem echten kommerziellen Projekt wird ein großer Brocken der verfügbaren Zeit nicht zuletzt dafür verwendet, ordentliche Tools zu implementieren, mit denen man die notwendigen Daten für ein Spiel komfortabel erzeugen kann.

Und es ward Licht

Doch nun, da wir die entsprechenden Dateien für Level über eine grafische Benutzeroberfläche erzeugen können, ist es an der Zeit, das Licht in unser virtuelles Universum zu lassen und ein interaktives Spiel mit diesen Leveln zu implementieren. Noch einmal tief durchatmen, und dann bitte umblättern.

15 Pandoras Legacy

»Hoher Sinn liegt oft in kindischem Spiel.« (Friedrich Schiller)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Definition eines einfachen Game-Designs
- ➔ Laden und effizientes Speichern von Level-Daten
- ➔ Rendern eines Levels mit dynamischer Beleuchtung
- ➔ automatisiertes Verbinden der Portale im Level
- ➔ Implementierung des Scene-Managements über Portale

15.1 Der Deathmatch-Ego-Shooter Pandoras Legacy

Viele Hunderte von Seiten lang haben wir uns durch dieses Werk gequält und eine wunderschöne Engine entwickelt. Doch was ist ein Motor, wenn man keine schicke Karosserie hat, in die man den Motor einbauen und ihn mal richtig ausfahren kann? Wir haben im Verlauf dieses Buches einen Haufen kleiner Demos gesehen, die uns dieses und jenes zeigen. Wir haben ein umfassendes Tool entwickelt, mit dem man 3D-Modelle und Level erzeugen kann. Alles schön und gut, aber letzten Endes sind das nur die Back-Bones, die nachher kein Spieler je zu Gesicht bekommt – es sei denn, man möchte die Modder¹ unterstützen und bietet den Editor als Tool zusammen mit dem Spiel an.

*Jetzt geht's um
die Wurst*

Was uns fehlt, ist das Frontend. Das hier ist ein Buch über die Programmierung von Computerspielen, und dann soll in diesem Buch nicht mal ein Spiel vorkommen? Nein, so können wir das natürlich nicht enden lassen. In diesem Kapitel werden wir zusammen ein einfaches Spiel entwickeln, bei dem wir unsere Level interaktiv durchwandern und uns heiße Deathmatch-Gefechte mit anderen Spielern liefern können.

Frontend

¹ Als »Modder« bezeichnet man im Fachjargon Leute, die sogenannte Mods zu einem Computerspiel entwickeln. Das Wörtchen Mod steht dabei für *Modification*, also eine Veränderung des Originals durch Austausch von Modellen, Grafiken, Sounds usw. um dem Spiel ein vollkommen anderes Flair zu verpassen.

Einfaches Game-Design

Quick and Dirty

Über das Game-Design gibt es hier nicht viel zu sagen. Wir werden es aufgrund des begrenzten Platzes in diesem Buch sehr einfach halten. Wir werden nur die Level unseres Editors aus dem vorangegangenen Kapitel laden und als interaktive Spielumgebung in einem Netzwerk bzw. über das Internet nutzbar machen – nicht mehr, aber auch nicht weniger. Dazu ist nämlich schon die eine oder andere Zeile Code erforderlich. In diesem Kapitel werde ich auch nicht das gesamte Projekt Zeile für Zeile zeigen können, aber natürlich befindet sich alles auf der CD-ROM zu diesem Buch.

Level-Editor-Code als Grundlage?

Nun wird der eine oder andere sicherlich sagen, dass wir ja den Code aus dem letzten Kapitel schon zur Verfügung haben. Mit diesem können wir bereits die Leveldaten laden und rendern lassen. Dann würde nur noch ein wenig Game-Play fehlen, oder? Ganz so einfach ist es natürlich nicht. Der Level-Editor ist ein sehr funktionales Tool, und seine Datenstrukturen dienen hauptsächlich dem Zweck, leicht editierbar zu sein. Daher sind auch die Render-Methoden des Editors für die Echtzeitdarstellung in einem Spiel nicht zu gebrauchen. Wenn wir einen solchen Level in Echtzeit interaktiv darstellen wollen, müssen wir die Daten aus dem Level in einer sehr viel effizienteren Weise in Strukturen speichern und rendern, als wir das im Editor mit der Polygon-für-Polygon-Strategie gemacht haben.

Neu, aber mit Optimierungs- potenzial

Folglich werden wir auch die Render-Methoden und die Datenstrukturen zur Speicherung der Daten nach dem Laden eines Levels komplett neu schreiben müssen. Zur Vereinfachung werde ich ein paar Abkürzungen nehmen, die uns ein wenig Performance kosten werden. Beispielsweise behandeln wir die beweglichen Door-Entities immer noch auf einer Polygon-für-Polygon-Basis. Die gesamte statische Level-Geometrie wird aber effizient in einem Rutsch je Textur und je Sektor zu rendern sein, was die Performance enorm erhöht. Diese Verbesserung solltet ihr dann auch für die Türen hinzufügen.

Licht und Schatten

Die Anwendung des dynamischen Per-Pixel-Lightings durch die Omni-Lights werden wir hier auch wesentlich performanter umsetzen als in unserem Level-Editor. Und hier werden wir auch endlich Schatten zur Verfügung haben. Dabei folgen wir einer einfachen Regel: Wir können in unserem Editor ja eine bestimmte Menge an Polygonen zu einem Mesh zusammenfügen. Jedes Mesh-Objekt, das in unserem Level gespeichert ist, wird im Spiel nachher einen Schatten werfen, sofern es im Radius einer Lichtquelle liegt. Natürlich sind auch multiple Schatten bei multiplen Lichtquellen möglich.

Ich erinnere an dieser Stelle noch einmal daran, dass wir in unserem Editor Lichtquellen und Schattenwerfer (engl. Shadow Caster) getrennt platzieren können. Der Schattenwurf für die Mesh-Objekte richtet sich im Spiel also nicht nach den Lichtquellen, sondern nach den Schattenwerfern.



Altbekannte Klassen neu aufgelegt

Natürlich können wir auf dem Design des Level-Editors aufbauen, auch wenn wir nicht die Methoden mitsamt ihren Implementierungen übernehmen können. Aber wir werden dieselbe hierarchische Struktur in unserem Game-Code verwenden, die auch im Level-Editor steckt. Wir werden also auch eine Klasse für Level, eine für Sektoren usw. haben. Tabelle 15.1 zeigt eine Gegenüberstellung der entsprechenden Klassen aus dem Editor und ihrer Äquivalente aus dem Game-Code, die wir nachher entwickeln werden. Wie ihr seht, werden wir insbesondere für Polygone, Polymeshs und Spawn-Points keine eigene Klasse im Game-Code haben.

Redundante Aufgaben

Klasse des Level-Editors	Klasse im Game-Code
CLevel	CGameLevel
CSector	CGameSector
CPortal	CGamePortal
CLight	CGameLight
CPolygon	n/a
CPolymesh	n/a
CSpawnPoint	n/a

Tabelle 15.1:
Liste der Klassen aus dem Level-Editor, für die wir eine Entsprechung in unserem Game-Code entwickeln werden

Diese redundanten Klassen ähneln sich natürlich gerade im Bereich des Ladens der Daten aus der Level-Datei. Wir werden im Game-Code allerdings nicht alle Informationen benötigen, die in der Level-Datei zu finden sind und die für den Editor notwendig wären. Auch vom Set der Methoden her sind die Klassen scheinbar recht ähnlich, denn wir brauchen in jedem Paar natürlich Methoden für das Rendern, Kollisionsabfragen usw. Die Unterschiede liegen aber insbesondere in der Implementierung der entsprechenden Methoden. Letzten Endes haben die Klassen also nicht viel gemeinsam, außer dem grundsätzlichen Aufbau, dass eine Instanz der Klasse CGameLevel die meisten Methoden-Aufrufe einfach an CGameSector delegiert und diese Klasse auch hier die Hauptarbeit erledigt.

Ähnlichkeiten und Unterschiede

Doch bevor wir uns daran machen, diese neuen Klassen zu entwerfen und zu implementieren, kümmern wir uns noch um ein ganz anderes Thema. Unser Level-Editor bietet zwar schon die Möglichkeit einer dynamischen

Beleuchtung per Pixel an, aber uns fehlt dort noch eine Sache, die immer dann unausweichlich auftritt, wenn Licht und solide Objekte aufeinander treffen: *Schatten*.

15.2 Im Schatten unser selbst

*Licht erzeugt
nicht automatisch
Schatten*

In der Welt der Computergrafik ist es natürlich so, dass wir alles selbst berechnen müssen. Auch wenn wir schon einen wunderbaren Beleuchtungsalgorithmus haben, der hübsche Lichtkegel auf unsere Geometrie zaubert, so werden wir keinen Schattenwurf in unseren Programmen entdecken. Wie sollten wir auch, denn wir haben den Schatten ja noch nicht selbst berechnet. In der Natur funktioniert das natürlich ganz einfach. Grundsätzlich ist alles stockdunkel. Damit liegt alles, was existiert, zunächst im Schatten, denn diesen kann man als die Abwesenheit von Licht definieren. Eine Lichtquelle emittiert Photonen in das Universum. Diese prallen auf verschiedene Objekte und erhellen diese durch Übertragung von Energie. Die Photonen werden dabei reflektiert, es sei denn, das Objekt absorbiert das Photon. Erst wenn das Photon einen optischen Scanner trifft (wie beispielsweise ein Auge), wird es registriert und kann als Punkt einer bestimmten Helligkeit in einer bestimmten Farbe zu einem Bild hinzugefügt werden – so wie es unser Gehirn mit den Signalen tut, die es von den Stäbchen der Augen erhält, die das Photon absorbiert haben.

*Helle Schatten
durch ambientes
Licht*

Damit ist auch schon klar, woher Schatten kommt. Als nicht beleuchtete Stellen (also als Schatten) interpretieren wir alle Stellen im Universum, von denen wir keine Photonen-Reflexionen wahrnehmen können, entweder weil keine Photonen an diese Stellen gelangt sind oder weil diese Stellen die Photonen vollkommen absorbiert haben. Nun sind aber nicht alle Schatten nachtschwarz, sondern oftmals nur ein wenig dunkler als die Umgebung. Hier spielt das ambiente Licht eine entscheidende Rolle. Denn kaum eine Stelle auf der Welt ist wirklich vollkommen ohne Photonenbeschuss, wenn eine Lichtquelle in Reichweite aktiv ist. Natürlich ist diese indirekte Beleuchtung über ambientes Licht wesentlich schwächer, als die direkte Beleuchtung durch die Lichtquelle es wäre. Daher nehmen wir diesen Helligkeitsunterschied in der Beleuchtung gleichsam als Schatten wahr.

Das führt uns auch schon zu der Implementierung von Schatten. Schatten entsteht ja nur dort, wo ein *Occluder* (dt. Sicht-Blockierer) die Photonen einer Lichtquelle davon abhält, den Bereich des Raums zu erhellen, den er aus Sicht der Lichtquelle verdeckt.

Verschiedene Verfahren zum Schattenwurf

Wie so oft in der Computergrafik ist die grundlegende Theorie über die Abläufe in der realen Natur total simpel. Die Modellierung eines solchen Verfahrens ist aber sehr kompliziert. Die intuitivste Variante wäre es natürlich, den entsprechenden Ablauf direkt aus der Natur auf den Computer zu übertragen. Dies versucht man bei Verfahren wie *Ray-Tracing*, *Radiosity* oder dem *Photon-Mapping*. Der Haken dabei ist aber, dass diese Verfahren auf einem Computer viel zu langsam sind, um in Echtzeit eingesetzt zu werden.

*Die Theorie ist
einfach*

Aufgrund ihrer einfachen Eleganz und ihrer guten Ergebnisse werden Verfahren wie z.B. Radiosity oft dazu eingesetzt, mit einem Tool statische Lightmaps für einen Level vorherzuberechnen, die dann später durch Multitexturing im Spiel verwendet werden. Der Nachteil dabei ist aber die Statik des Verfahrens. Weder die Lichtquellen noch die Level-Geometrie dürfen sich dann zur Laufzeit des Spiels ändern.



Glücklicherweise gibt es aber auch einige Verfahren, die man in Echtzeit einsetzen kann. Die einfachste Technik sind die projizierten, ebenen Schatten. Dabei wird ein Objekt, das Schatten werfen soll, durch eine geeignete Projektionsmatrix aus Sicht der Lichtquelle auf eine Ebene projiziert. Man formt dadurch das Bild nach, das der Schatten des Objekts ergeben würde. Leider kann dieses Verfahren selten überhaupt sinnvoll angewendet werden, denn man muss dazu die Ebene kennen, auf die der Schatten projiziert werden soll. Zudem muss man natürlich für jede Ebene mit einer eigenen Projektionsmatrix arbeiten und das projizierte Objekt auf die Geometrie beschränken, die in dieser Ebene liegt. Damit scheidet ein solches Verfahren schon ganz von selbst für komplexere Objekte wie beispielsweise ganze Level aus. Hier jede einzelne Ebene herauszufiltern, die einen Schatten empfangen soll, wäre viel zu aufwändig.

*Realtime-fähige
Techniken*

Ein Verfahren, das szenenbasiert funktioniert, also für die gesamte virtuelle Welt in einem Rutsch, ist das Verfahren der Shadow-Maps. Dabei rendert man die Szene zuerst aus der Sicht der Lichtquelle, die einen Schatten werfen soll. Das Rendern erfolgt dabei nicht in den Frame-Buffer, sondern in einen speziellen Buffer. Es werden auch nicht die Farben eines jeden Pixels geschrieben, sondern sein Tiefen-Wert. In diesem Buffer befindet sich also quasi nach dem Rendern eine Depth-Buffer-analoge Füllung aus Sicht der Lichtquelle. Diesen Buffer nennt man Shadow-Map. Nun rendert man die Szene aus Sicht des Betrachters in den Frame-Buffer. Für jeden Pixel prüft man, ob er weiter von der Lichtquelle entfernt ist als der Wert in der Shadow-Map, die wie eine Textur auf die gesamte Szene projiziert ist. Ist das der Fall, dann liegt der Pixel aus Sicht der Lichtquelle hinter einem anderen Objekt mit einem kleinen Depth-Wert und muss folglich im Schatten liegen.

Shadow-Maps

Dieses Verfahren klingt wirklich gut, hat aber auch ein paar Probleme. Insbesondere ist es mit DirectX nicht so leicht umzusetzen wie mit OpenGL. Auch die Auflösung der Shadow-Map spielt eine entscheidende Rolle. Ist die Auflösung zu klein, dann erhält man einen abgestuften Schatten mit deutlich sichtbaren *Pixel-Treppchen*, weil die Shadow-Map beim Projizieren zu sehr gedehnt wurde.

Stencil Shadow Volumes

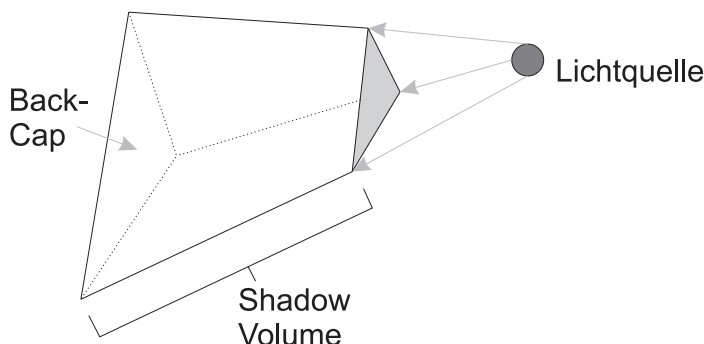
Einen vollkommen anderen Ansatz, der objektbasiert ist, bieten die *Stencil Shadow Volumes*. In Kombination mit dem Stencil-Buffer wird hier ein cleverer Trick genutzt, um Schatten zu erzeugen. Dieses Verfahren werden wir hier implementieren, und daher widmen wir ihm einen eigenen Abschnitt.

Zur Theorie der Shadow-Volumes

Ausziehen der Objekte

Was genau eigentlich ein Shadow-Volume ist, das ist recht schnell erklärt. Abbildung 15.1 verdeutlicht uns dies zudem optisch. Wir sehen dort ein Dreieck und eine Lichtquelle. Nun möchten wir herausfinden, wohin das Dreieck seinen Schatten in Relation zu der Lichtquelle wirft. Dazu schießen wir einfach einen Strahl von der Lichtquelle aus zu den einzelnen Vertices des Dreiecks und verlängern diese Strahlen in Laufrichtung des Lichts. Dadurch wird das Dreieck zu einem Volumen ausgezogen, und dieses Volumen nennt sich nun Shadow-Volume. In der Grafik ist auch die so genannte *Back-Cap* (dt. etwa Rückseiten-Abdeckung) dargestellt. Dabei handelt es sich um eine Menge von Polygonen (hier nur ein Dreieck), die das Volumen auf der Rückseite schließt. Analog nennt man die Verschluss-Geometrie auf der Vorderseite die *Front-Cap*. In diesem Fall wäre es das originale Dreieck selbst.

Abbildung 15.1:
Ausziehen des Shadow-Volumes aus einem Dreieck in Relation zu einer Punktlichtquelle. Das Dreieck selbst formt dabei die Front-Cap, das projizierte Dreieck bildet die Back-Cap.



Vergleich zum View-Frustum

Der Witz an der Sache ist sicherlich jedem klar: Alles, was innerhalb dieses Shadow-Volumes liegt, befindet sich im Schatten des Dreiecks in Relation zu der vorhandenen Lichtquelle. Hier drängt sich ein bisschen der Vergleich zu dem View-Frustum auf. Alles, was innerhalb des View-Frustums liegt, befindet sich im sichtbaren Bereich des Betrachters. Beim Shadow-Volume ist es halt umgekehrt: Alles, was sich im Shadow-Volume befindet, kann der

Betrachter, der in diesem Fall die Lichtquelle ist, *nicht* sehen und damit auch nicht beleuchten. So weit, so gut. Die Implementierung dieser Technik ist aber nicht ganz so einfach, wie es hier scheint. Wir müssen schließlich später auf einer per-Pixel Basis bestimmen, was im Schatten liegt und was nicht. Ein einfaches objekt- oder polygonbasiertes Culling können wir dementsprechend nicht machen. Also machen wir uns mal schlau über die Algorithmen, mit denen man diese Technik umsetzen kann. Dazu gibt es zwei bedeutende Methoden, zu denen wir gleich kommen.

Der Stencil-Buffer (dt. Schablonen-Puffer) kann dazu verwendet werden, bestimmte Bereiche auf dem Render-Target zu maskieren und dadurch bestimmte Spezial-Effekte zu ermöglichen. Rein technisch gesehen ist der Stencil-Buffer auf modernen Grafikkarten als Teil des Depth-Buffers umgesetzt.



Bevor wir aber dazu kommen, schauen wir uns an, wie man das Shadow-Volume-Verfahren zusammen mit dem Stencil-Buffer einsetzt. Der Ablauf ist dabei für beide Varianten nahezu gleich und unterscheidet sich nur beim Betanken des Stencil-Buffers. Der Ablauf ist allgemein wie folgt:

- ➔ Rendern der Szene ohne Schatten in den Depth-Buffer
- ➔ Neuberechnung der Volumes, falls nötig
- ➔ Rendern der Shadow-Volumes in den Stencil-Buffer
- ➔ Rendern der Szene in den Frame-Buffer

Wir rendern das Shadow-Volume in den Stencil-Buffer, damit wir im Stencil-Buffer alle Pixel maskieren können, die im Schatten liegen. Wie das genau funktioniert, sehen wir gleich bei den beiden Varianten, die es dazu gibt. Nach dem Rendern des Shadow-Volumes hat man aber eine Maske im Stencil-Buffer, die genau die Bereiche maskiert, die im Schatten liegen. Wenn man danach die Szene in den Frame-Buffer rendert, so werden diejenigen Pixel nicht gerendert, die im Schatten liegen. Dies kann man natürlich noch variieren. Beispielsweise kann man die Szene zuerst in ambientem Licht rendern, dann das Shadow-Volume bzw. die Shadow-Volumes rendern und schließlich die Szene mit allen anderen Beleuchtungseffekten rendern. So hat man nicht einfach schwarze Stellen da, wo der Schatten ist, sondern man hat dort den Farbwert des ambienten Pass.² Und das sieht in der Regel besser aus.

*Schatten-
bereiche sind
maskiert*

² Als »ambienten Pass« bezeichnet man das Rendern der Szene mit ambientem Licht in den Frame-Buffer. Dazu kommen dann noch andere Rendering-Passes, beispielsweise je einer pro aktiver Lichtquelle, die mit dem ambienten Pass verrechnet werden.

Nachdem das geklärt ist, schauen wir uns nun an, wie wir das Shadow-Volume in den Stencil-Buffer rendern, um die gewünschte Maske zu erhalten. Wie bereits erwähnt, gibt es dazu zwei Varianten.



Auf *Gamedev.net* gibt es einen ausgezeichneten Artikel »The Theory of Stencil Shadow Volumes« von Hun Yen Kwoon. Aus diesem sehr guten Artikel habe ich auch einige der Abbildungen in leicht abgewandelter Form übernommen.

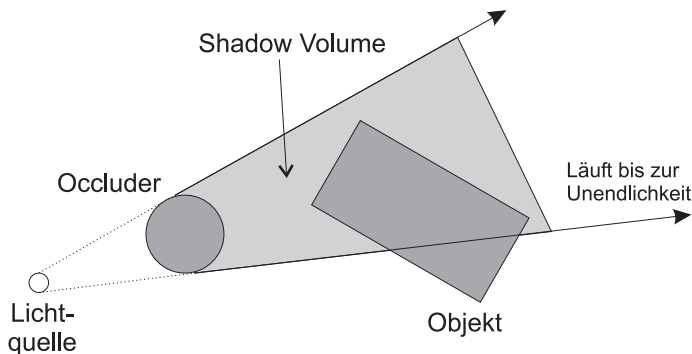
Quelle: <http://www.gamedev.net/columns/hardcore/shadowvolume>

Depth-Pass-Variante

Stencil-Tricks

Der erste Algorithmus für *Stencil Shadow Volumes* nennt man Depth-Pass-Variante oder manchmal auch Z-Pass-Variante. Warum das so ist, werden wir gleich sehen. Zunächst aber betrachten wir Abbildung 15.2, die eine Szene aus der Vogelperspektive zeigt. Dabei lassen wir die Front-Cap und die Back-Cap mal außen vor und erhalten damit ein Volume, das vorn offen ist und sich hinten bis in alle Unendlichkeit erstreckt und dort auch nicht geschlossen ist.

Abbildung 15.2: Ansicht einer 3D-Szene mit einer Lichtquelle und einem Occluder (dunkelgrauer Kreis) und dessen Shadow-Volume (hellgraue Fläche) aus der Vogelperspektive. Das Shadow-Volume verläuft weiter bis in die Unendlichkeit, was hier durch Pfeile angedeutet ist. Ein weiteres geometrisches Objekt, das selbst keinen Schatten wirft, ist hier zu sehen.



Stencil-Werte setzen

Nun rendern wir den Shadow-Volume in den Stencil-Buffer. Dabei ist zu beachten, dass der Depth-Buffer bereits mit den korrekten Depth-Werten der Szene des aktuellen Frames betankt sein muss. Dann tun wir Folgendes:

- ➔ Wir rendern die Frontfaces des Volumes. Wenn der Depth-Test eines Pixels erfolgreich verläuft, dann erhöhen wir den korrespondierenden Wert im Stencil-Buffer um 1.

- ➔ Wir rendern die Backfaces des Volumes. Wenn der Depth-Test eines Pixels erfolgreich verläuft, dann verringern wir den korrespondierenden Wert im Stencil-Buffer um 1.

Zuerst rendern wir also das Volume mit dem normalen Backface-Culling, also die Polygone, die mit ihrem Normalenvektor zum Betrachter hin zeigen. Dann rendern wir das Volume noch einmal, drehen aber vorher das Backface-Culling um, um die Rückseiten der Polygone zu rendern. Diese Variante heißt übrigens Depth-Pass, weil man nur dann eine Aktion unternimmt, wenn der Depth-Test erfolgreich verlaufen ist. Nun haben wir im Stencil-Buffer eine Maske aus Nullen und Einsen stehen. Allerdings sind diese sehr pfiffig angeordnet.

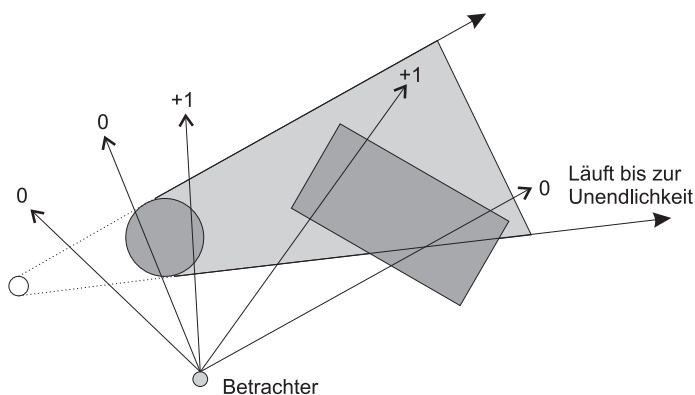


Abbildung 15.3:

Wenn das Shadow-Volume in den Stencil-Buffer gerendert wird, dann entscheidet sich der Stencil-Buffer-Wert per Pixel. Ist von der aktuellen Position des Betrachters aus ein Frontface-Pixel zu sehen, so wird der Wert um 1 erhöht. Ist ein Backface-Pixel zu sehen, so wird der Wert um 1 verkleinert.

Abbildung 15.3 zeigt uns, welcher tieferer Sinn dahinter steht. Wir sehen dort den Betrachter der Szene und sind gerade dabei, das Shadow-Volume in den Stencil-Buffer zu rendern. Die dunklen geometrischen Objekte wurden bereits in den Depth-Buffer geschrieben. Nun schauen wir uns exemplarisch fünf Pixel an, die wir rendern. Jeder wird durch einen Strahl repräsentiert, der vom Betrachter ausgeht. Der erste und der zweite Strahl von links betreffen Pixel, die gar nicht zum Shadow-Volume gehören. Daher bleibt der Stencil-Wert auf 0. Pixel der Geometrie, die später an diesen Stellen gerendert werden, liegen nicht im Schatten. Der dritte Strahl ist schon interessanter. Wir rendern zunächst einen Pixel des Frontfaces, was erfolgreich verläuft. Also erhöhen wir den Stencil-Wert auf 1. Dann rendern wir den Pixel für das Backface des Volumes an dieser Stelle. Doch hier schmettert der Depth-Buffer uns knallhart ab, denn die Kugel, die ja selbst das Volume erzeugt hat, blockiert die Sicht des Betrachters auf das Backface des Volumes. Daher können wir den Stencil-Wert nicht wieder senken, und er bleibt auf 1. Damit haben wir diesen Pixel für spätere Render-Aufrufe maskiert, denn alles, was dort gerendert werden muss, liegt eindeutig im Schatten. Bei dem vierten Strahl verhält es sich ganz genauso. Der ganz rechte Strahl hin-

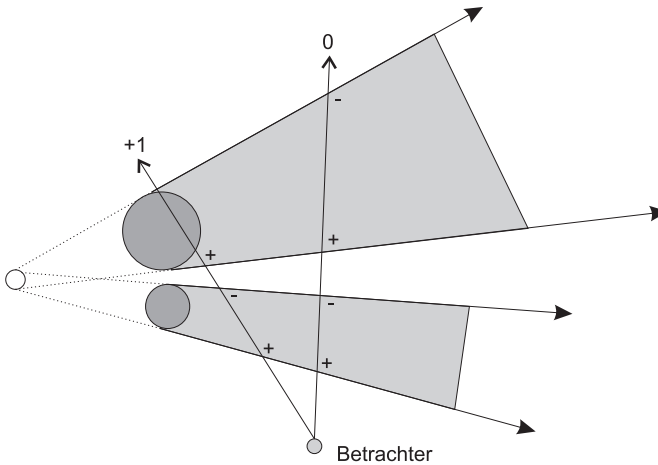
Sinn des Depth-Pass

gegen wird bereits für das Rendern des Frontface-Pixels des Volumes vom Depth-Buffer abgeschmettert, denn das viereckige Objekt liegt von Betrachter aus gesehen an dieser Stelle noch vor dem Frontface-Pixel des Volumes. Daher bleibt der Stencil-Wert auch hier auf 0, und der Pixel wird korrekterweise nicht vom Stencil-Buffer maskiert.

Nett, oder ... ?

Das war der ganze Trick dabei. Auf diese clevere Weise kann man durch Shadow-Volumes ganz einfach Schatten rendern. Letzten Endes macht man nichts anderes, als zu zählen, wie viele Frontfaces und wie viele Backfaces der Shadow-Volume ein Strahl vom Betrachter aus zu einem Pixel durchlaufen hat, bevor er vom Depth-Buffer gestoppt wird – sprich: auf ein solides Objekt der Szene trifft. Wenn beide Werte gleich sind, liegt der Pixel außerhalb eines Volumes, sind beide Werte unterschiedlich, dann liegt der Pixel innerhalb eines Volumes. Bei dieser Methode können Objekte natürlich auch Schatten auf sich selbst werfen, was ja auch ein Anspruch an ein robustes Verfahren ist. Und natürlich funktioniert das Ganze auch, wenn mehrere Volumes hintereinander liegen oder sich überlappen. Das zeigt Abbildung 15.4.

Abbildung 15.4:
Die Depth-Pass-Variante funktioniert auch, wenn mehrere Shadow-Volumes sich überlappen oder hintereinander liegen.



Ja. Aber ...!

So weit ist das ja total simpel. Das Problem ist natürlich mal wieder, dass der Teufel im Detail steckt. Oder besser gesagt in den Spezialfällen. Schauen wir uns Abbildung 15.5 an.

Betrachter im Volume? Au weia!

Wie ihr seht, besteht das Problem darin, dass die Werte im Stencil-Buffer nun fehlerhaft sind bzw. nicht mehr dem Schema entsprechen, dass Pixel, deren Stencil-Wert auf 1 steht, im Schatten liegen. In diesem Fall ist die Depth-Pass-Technik unbrauchbar. Nun gibt es eine scheinbar elegante Lösung für dieses Problem. Sobald sich der Betrachter im Shadow-Volume befindet, invertiert man den Algorithmus, oder man initialisiert den Stencil-Buffer mit 1 statt mit 0. Doch das ist keine robuste Lösung. Hier bekommt man Probleme mit der *Near Clipping Plane*, da es Fälle gibt, in denen der Betrachter nicht eindeutig

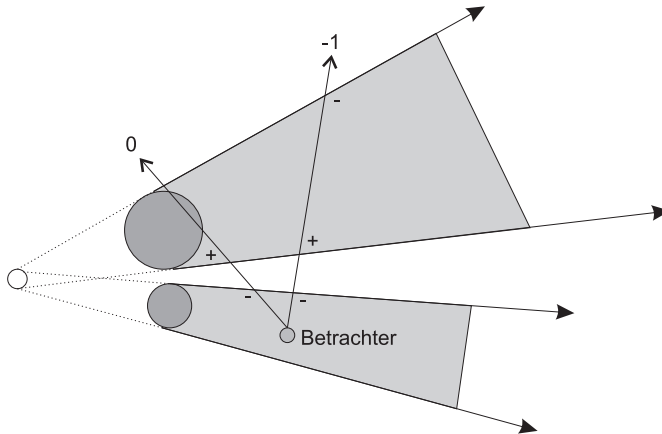


Abbildung 15.5:
Wenn der Betrachter im Shadow-Volume steht, dann versagt die Depth-Pass-Variante, und die Werte im Stencil-Buffer verlieren ihre Aussagekraft.

vor oder hinter der Near Clipping Plane ist. Wenn die Near Clipping Plane einen Teil des Shadow-Volumes clippt, dann ist das so, als befände sich der Betrachter im Shadow-Volume, weil die Frontface-Pixel des Volumes zum Teil fehlen. Das zweite Problem ist, dass man bei komplexeren Shadow-Volumes oftmals gar nicht so einfach bestimmen kann, ob sich der Betrachter tatsächlich im Volume befindet oder nicht. Ein solcher Trick als Notbehelf wird also in vielen Situationen funktionieren, aber eben nicht in allen, die während eines Spiels wirklich auftreten können.

Depth-Fail-Variante (Carmack's Reverse)

All diese Probleme haben den bereits des Öfteren erwähnten John Carmack dazu veranlasst, nach einer robusteren Lösung zu suchen. Dabei ist die zweite Variante des Stencil-Shadow-Volume-Renderings herausgekommen, die man Depth-Fail-Variante oder manchmal auch Z-Fail nennt und die den schmunken Beinamen *Carmack's Reverse* trägt.³ Diese Variante funktioniert wie folgt:

Schon wieder Carmack

- ➔ Rendere die Backfaces des Volumes. Wenn der Depth-Test eines Pixels negativ ausfällt, dann erhöhe den korrespondierenden Wert im Stencil-Buffer um 1.
- ➔ Rendere die Frontfaces des Volumes. Wenn der Depth-Test eines Pixels negativ ausfällt, dann verringere den korrespondierenden Wert im Stencil-Buffer um 1.

Vorteile

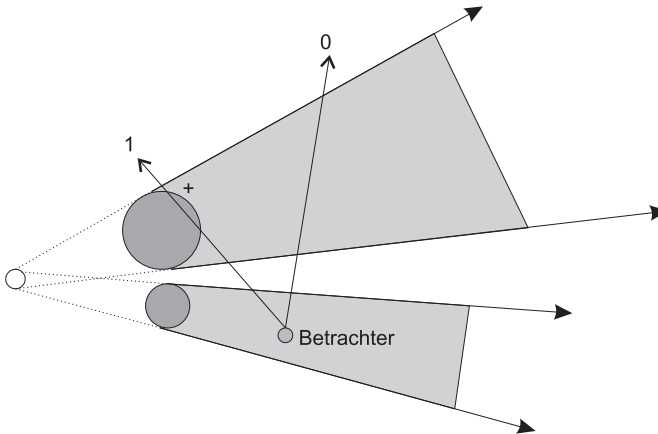
³ Auf der Webseite von nVidia könnt ihr ein Dokument finden, in dem John Carmack in einer E-Mail beschreibt, wie er die Z-Fail-Variante erstellt hat.

Quelle: http://developer.nvidia.com/object/robust_shadow_volumes.html

Hier sieht man auch, warum diese Variante »Depth-Fail« heißt: Wir ändern den Wert im Stencil-Buffer nur, wenn der Depth-Test fehlgeschlagen ist. Wenn man nun die Variante des Shadow-Volumes verwendet, dann kann der Betrachter sowohl außerhalb der Shadow-Volumes als auch innerhalb eines Volumes stehen. Die Ergebnisse sind in beiden Fällen korrekt. Dies zeigt die Abbildung 15.6.

Abbildung 15.6:

Die Depth-Fail-Methode funktioniert auch, wenn der Betrachter in einem Shadow-Volume steht. Der Stencil-Wert wird nur erhöht, wenn das Rendern eines Backface-Pixels am Depth-Buffer scheitert. Scheitert das Rendern eines Frontface-Pixels, wird der Wert verringert.



Nachteile

Natürlich bekommen wir nie etwas geschenkt. Wo es Vorteile gibt, da gibt es auch Nachteile. Der erste Nachteil ist, dass wir kein offenes Volume verwenden können. Der Algorithmus beruht darauf, den Wert im Stencil-Buffer zu setzen, wenn das Rendern eines Backface-Pixels des Volumes am Depth-Buffer scheitert. Wenn der Betrachter aber nun genau im Volume steht und durch das offene Ende des Volumes blickt, dann kann dieser Fall nicht eintreten, und Objekte, die im Schatten liegen, werden nicht schattiert. Das Volume muss also auf alle Fälle geschlossen sein. Dazu muss man eine Front-Cap und eine Back-Cap für das Shadow-Volume berechnen. Das ist jedoch das kleinste Problem.



Für die Berechnung der Front-Cap gibt es mehrere Möglichkeiten. Mark Kilgard hat eine komplexe Methode beschrieben, mit der man die vom Licht abgewandten Polygone des Occluders auf die Near Clipping Plane projiziert und diese dann als Front-Cap verwendet. Aber es geht auch einfacher. Man kann direkt die Triangles des Occluders für die Front-Cap verwenden, die dem Licht zugewandt sind. Die Back-Cap verwendet dann dieselben Triangles, jedoch mit vertauschter Blickrichtung, damit diese nach außen zeigen. Wir werden das noch später in unserer Implementierung sehen.

Far Plane Clipping

Das größere Problem ist, dass die Depth-Fail-Variante ein Problem mit der Far Clipping Plane hat, im Gegensatz zu dem Near-Clipping-Problem der Depth-Pass-Variante. Das Problem ist dasselbe wie bei dem offenen Shadow-

Volume mit dieser Variante. Das ist ja auch logisch, denn wenn die Far Clipping Plane ein Stück aus dem Shadow-Volume wegclippt, dann ist das Shadow-Volume ja wieder offen. Dagegen gibt es nun zwei Rezepte: Zum einen kann man sein Shadow-Volume einfach nicht bis alle Unendlichkeit ausziehen und dafür Sorge tragen, dass das Shadow-Volume vollkommen innerhalb der Far Clipping Plane liegt. Die andere, etwas robustere Lösung ist, dass man eine so genannte **unendliche Projektionsmatrix** verwendet. Damit liegt die Far Clipping Plane dann mathematisch gesehen in der Unendlichkeit und kann auch dort kein Teil des unendlich ausgezogenen Shadow-Volumes abschneiden.

Damit sind wir am Ende unseres kleinen Exkurses über die Funktionsweise von Shadow-Volumes. Nun ist es an der Zeit, dass wir uns an die Praxis machen.

Genug Theorie

Implementierung von Shadow-Volumes

Wir werden im Folgenden eine Klasse für Shadow-Volumes konstruieren, die unabhängig von einem bestimmten Vertex-Format funktioniert. Das Erstellen eines Shadow-Volumes ist eine total triviale Angelegenheit, wie wir gleich sehen werden. Im DirectX SDK gibt es ein gutes Sample als Vorlage, das jedoch die Depth-Pass-Methode verwendet. Allerdings ist anzumerken, dass sich die verwendete Variante nicht sehr stark auf das Volume auswirkt. Für die Depth-Fail-Methode müssen wir lediglich die Front-Cap und die Back-Cap für das Volume zusätzlich erstellen. Alle anderen Unterschiede zwischen den beiden Varianten beziehen sich ja auf das Rendern des Volumes bzw. auf die dabei zu verwendenden Einstellungen des Stencil-Buffer. Aber eines nach dem anderen.

*Depth-Fail
natürlich*

Klassendeklaration

Bei der Konstruktion des Shadow-Volumes benötigen wir eine Hilfsstruktur für ein Dreieck. Diese beinhaltet drei Pointer auf Vertices vom Typ `VERTEX` und einen Normalenvektor für das Dreieck.

Hilfsstruktur

```
typedef struct TRIANGLE_TYPE {  
    VERTEX *pV0;  
    VERTEX *pV1;  
    VERTEX *pV2;  
    ZFXVector vcN;  
} TRIANGLE;
```



Das in dieser Struktur verwendete Vertex-Format bedeutet aber nicht, dass unser Shadow-Volume auf ein bestimmtes Vertex-Format festgelegt ist. Unsere animierten Characters verwenden ja beispielsweise ein anderes Format als die Level-Geometrie. Beide sollen aber natürlich dieselbe Klasse für Shadow-Volumes nutzen können. In der folgenden Klasse `CShadowVolume` sind

*Vertex-format-
unabhängige
Klasse*

die Vertices daher in einem `BYTE`-Pointer gespeichert, und die Byte-Größe eines Vertex wird im Attribut `m_nSize` vermerkt. Alles in allem kommt die Klasse mit gerade mal sechs Methoden neben dem Konstruktor und dem Destruktor aus.



```
class CShadowVolume {
public:
    CShadowVolume(void);
    virtual ~CShadowVolume(void);

    HRESULT Init(const void*, UINT,
                const WORD*, UINT,
                UINT, ZFXAabb,
                ZFXRenderDevice*);

    HRESULT UpdateGeometry(const void*,
                          const WORD*,
                          ZFXAabb);

    HRESULT Render( CGameLight*, ZFXMatrix* );

private:
    ZFXRenderDevice *m_pDevice;
    ZFXAabb          m_Aabb;

    // originales Mesh
    TRIANGLE      *m_pTris;
    BYTE          *m_pVerts;
    WORD          *m_pIndis;
    WORD          *m_pEdges;
    bool          *m_pIsCap;
    UINT          m_NumV;
    UINT          m_NumI;
    UINT          m_NumT;
    UINT          m_NumE;
    UINT          m_nSize;

    // während der Initialisierung
    PVERTEX      *m_pVolume;
    UINT          m_NumVV;

    void BuildShadowVolume(ZFXVector);
    void AddEdge(WORD v0, WORD v1);
    inline PVERTEX Extrude(const VERTEX*,
                          const ZFXVector&);
}; // class
typedef class CShadowVolume *LPShadowVolume;
```


Die folgenden beiden öffentlichen Methoden `CShadowVolume::Init` und `CShadowVolume::UpdateGeometry` dienen mehr oder weniger demselben Zweck: Mit ihnen wird die Geometrie bestimmt, für die ein Shadow-Volume berechnet werden soll. Die erstgenannte Methode muss dabei ein paar Informationen mehr erhalten, wie beispielsweise die Größe eines Vertex, die Anzahl an Vertices und Indices usw. Die zweite Methode setzt dann voraus, dass sich an der Triangulation des Meshs nichts geändert hat, sondern dass sich lediglich die Positionen der Vertices verändert haben. Die neuen Vertex- und Indexdaten werden einfach über die alten Daten im Shadow-Volume geschrieben. Die Größe eines Vertex muss dabei auch identisch mit der Geometrie aus dem `Init()`-Aufruf sein. Das Update brauchen wir natürlich insbesondere für unsere animierten Character-Objekte.

Methoden

Die Bedeutung der weiteren Methoden und Attribute wird sich uns im Folgenden erschließen, wenn wir uns an die Implementierung der hier deklarierten Methoden machen.

Attribute**Eine Instanz initialisieren**

Beim Initialisieren eines Objekts der Klasse `CShadowVolume` müssen wir neben den trivialen Aufgaben, wie beispielsweise dem Kopieren der Vertex- und Indexdaten in die Attribute des Objekts, noch einen anderen Job erledigen. Wir müssen eine Liste von `TRIANGLE`-Objekten erzeugen, die unsere Geometrie repräsentieren. Dabei erhält ein `TRIANGLE`-Objekt drei Pointer auf die Vertices, die zu ihm gehören, sowie einen Normalenvektor.

**Konstruktion der
Triangles**

```
HRESULT CShadowVolume::Init(const void *pVerts,
                           UINT NumV,
                           const WORD *pIndis,
                           UINT NumI,
                           UINT nSize, ZFXAabb aabb,
                           ZFXRenderDevice *pDevice) {
    m_pDevice = pDevice;
    m_Aabb     = aabb;
    m_nSize    = nSize;

    m_pVerts = new BYTE[NumV*nSize];
    m_pIndis = new WORD[NumI];
    m_pTris  = new TRIANGLE[NumI/3];
    m_pIsCap = new bool[NumI/3];
    m_NumV   = NumV;
    m_NumI   = NumI;
    m_NumT   = NumI/3;

    m_pEdges = new WORD[m_NumT*6];

    // im schlimmsten Fall hat der Volume ein
    // Rechteck für jede Kante aller Dreiecke: 3*4
    m_pVolume = new PVERTEX[m_NumT*12];
```



```

// kopiere Daten des Original-Meshs
memcpy(m_pVerts, pVerts, m_nSize*NumV);
memcpy(m_pIndis, pIndis, sizeof(WORD) * NumI);

int nFact = m_nSize/sizeof(BYTE);

// erzeuge Dreiecksdaten
for (UINT i=0; i<m_NumT; i++) {
    m_pTris[i].pV0 = (VERTEX*)&m_pVerts[
        m_pIndis[i*3+0] * nFact ];
    m_pTris[i].pV1 = (VERTEX*)&m_pVerts[
        m_pIndis[i*3+1] * nFact ];
    m_pTris[i].pV2 = (VERTEX*)&m_pVerts[
        m_pIndis[i*3+2] * nFact ];

    ZFXVector vc0( m_pTris[i].pV0->x,
        m_pTris[i].pV0->y,
        m_pTris[i].pV0->z);

    ZFXVector vc1( m_pTris[i].pV1->x,
        m_pTris[i].pV1->y,
        m_pTris[i].pV1->z);

    ZFXVector vc2( m_pTris[i].pV2->x,
        m_pTris[i].pV2->y,
        m_pTris[i].pV2->z);

    // Normalenvektor vorberechnen
    m_pTris[i].vcN.Cross( (vc2 - vc1),
        (vc1 - vc0));
    m_pTris[i].vcN.Normalize();
}
return ZFX_OK;
} // Init

```

PVERTEX Hier sehen wir auch eine neue Vertex-Struktur namens `PVERTEX`. Diese ist in der *ZFXEngine* definiert und enthält nur eine Position, also eine x-, eine y- und eine z-Koordinate. Wenn wir das Shadow-Volume an die Grafikkarte zum Rendern schicken, dann wissen wir ja, dass wir das Volume nur in den Stencil-Buffer rendern werden. Dazu brauchen wir unseren Bus nicht mit unnötigen Normalenvektor- oder Texturkoordinaten-Informationen zu belasten. Und noch etwas ist dabei interessant: Wir allokieren hier im Attribut `m_pVolume`, das nachher das Shadow-Volume enthalten wird, Platz für wesentlich mehr Vertices, als in der Geometrie vorhanden sind. Das hat den Grund, dass wir einige Dreiecke erzeugen müssen, die in der eigentlichen Geometrie gar nicht vorkommen. Im schlimmsten Fall muss man an jede Kante eines Dreiecks ein Rechteck anlegen und dies in Laufrichtung des Lichts ausziehen. Daher kann ein Dreieck im schlimmsten Fall drei zusätzliche Rechtecke erfor-

dern. Das macht zwölf zusätzliche Vertices pro Dreieck der Geometrie, und für die müssen wir genügend Platz allokalieren.

Erstellen des Shadow-Volumes

Bevor wir mit dem Erstellen eines Shadow-Volumes beginnen, müssen wir ein paar Hilfsfunktionen schreiben, die uns die Arbeit erleichtern und den Code wesentlich lesbarer machen. Die folgende kleine Inline-Methode dient uns dazu, eine angegebene Position um einen angegebenen Vektor zu verschieben und aus der neuen Position ein PVERTEX-Objekt zu erstellen und zurückzugeben:

*Ausziehen eines
Vertex*

```
inline PVERTEX CShadowVolume::Extrude(const VERTEX *pv,
                                       const ZFXVector &vc)
{
    PVERTEX v;
    v.x = pv->x - vc.x;
    v.y = pv->y - vc.y;
    v.z = pv->z - vc.z;
    return v;
} // extrude
```



Die zweite Hilfsfunktion benötigen wir, um eine Liste mit Kanten (engl. Edge) der Geometrie zu erstellen. Unsere Klasse CShadowVolume hat ein Attribut m_pEdges. Dieses wird beim Erstellen des Shadow-Volumes eine Liste mit Kanten der Geometrie enthalten, die die Silhouette der Geometrie aus Sicht der Lichtquelle bilden. Eine Kante besteht in unserem Fall aus zwei Indices auf die Vertex-Liste der Geometrie. Die folgende Methode fügt der Kantenliste eine Kante aus zwei Indices hinzu. Dabei muss die Bedingung erfüllt werden, dass die Kante in der Liste noch nicht auftaucht. Genau diese Bedingung bestimmt nämlich, welche Kanten die Silhouette bilden. Nur Kanten der Silhouette werden genau ein einziges Mal zu dieser Liste hinzugefügt. Finden wir hingegen eine hinzuzufügende Kante bereits in der Liste, so dürfen wir die Kanten nicht hinzufügen und müssen diese Kante aus der Liste löschen.

*Edge-Liste
ergänzen*

```
void CShadowVolume::AddEdge(WORD v0, WORD v1) {
    // Entferne innere Kanten (Die tauchen
    // in der Liste genau zweimal auf.)
    for( UINT i=0; i < m_NumE; i++ ) {
        if( (m_pEdges[2*i+0]==v0 && m_pEdges[2*i+1]==v1)
            || (m_pEdges[2*i+0]==v1 && m_pEdges[2*i+1]==v0))
        {
            if( m_NumE > 1 ) {
                m_pEdges[2*(m_NumE-1)+0] = m_pEdges[2*i+0];
                m_pEdges[2*(m_NumE-1)+1] = m_pEdges[2*i+1];
            }
        }
    }
}
```



```

        m_NumE--;
        return;
    }
}

m_pEdges[2*m_NumE+0] = v0;
m_pEdges[2*m_NumE+1] = v1;
m_NumE++;
} // AddEdge

```

Erstellung des Volumes

Nun können wir endlich das eigentliche Volume erstellen. Dabei sei noch flugs erwähnt, dass wir das Shadow-Volume immer dann neu berechnen müssen, wenn sich das schattenwerfende Objekt oder die Lichtquelle bewegt hat. Das Erstellen des Volumes verläuft in zwei Passes. In einem ersten Pass laufen wir durch alle TRIANGLE-Objekte der Geometrie und prüfen ihre Ausrichtung zur Lichtquelle. Dazu verwenden wir den Normalenvektor eines TRIANGLE-Objekts und den als Parameter angegebenen Vektor vom Objekt zur Lichtquelle hin. Ist der Winkel zwischen den beiden kleiner als 90 Grad (also das Skalarprodukt ≥ 0), dann schaut das betrachtete Dreieck zur Lichtquelle hin. Seine Kanten bilden dabei potenziell einen Teil der Silhouette und werden zu der Kantenliste hinzugefügt. Zusätzlich bilden diese Dreiecke die Front-Cap des Volumes und werden daher direkt zur Vertex-Liste `m_pVolume` hinzugefügt. Für die Back-Cap verwenden wir dieselben Dreiecke wie für die Front-Cap. Allerdings ziehen wir diese in Laufrichtung des Lichts aus und drehen natürlich die Reihenfolge der Vertices um, damit die Dreiecke auch in die richtige Richtung schauen.

Einfügen von Rechtecken

Nach dem ersten Pass haben wir also bereits die Front-Cap und die Back-Cap des Volumes. Alles, was dazwischen liegt, fehlt uns aber noch. Allerdings haben wir ja auch eine Liste mit den Kanten, die die Silhouette bilden. Diese Silhouette ist quasi der Umriss der Front-Cap aus Sicht der Lichtquelle. Um die Front-Cap und die Back-Cap zu einem geschlossenen Volume verbinden zu können, müssen wir nun lediglich für jede dieser Kanten ein Rechteck zu dem Volume hinzufügen, dessen hinteres Ende ebenfalls in Laufrichtung des Lichtes ausgezogen wird und damit bis zur Back-Cap läuft und das Volume schließt.



```

void CShadowVolume::BuildShadowVolume(ZFXVector vcLight) {
    ZFXVector vc;
    PVERTEX v1, v2, v3, v4;

    vc = vcLight * 5.0f;

    // Zähler zurückstellen
    m_NumVV = 0;
    m_NumE = 0;

    // 1st Pass: Bestimme, welche Triangle die
    // Front- und Back-Cap des Volumes bilden

```

```

for (UINT i=0; i<m_NumT; i++) {

    WORD wFace0 = m_pIndis[3*i+0];
    WORD wFace1 = m_pIndis[3*i+1];
    WORD wFace2 = m_pIndis[3*i+2];

    if ( (m_pTris[i].vcN * vcLight) >= 0.0f ) {

        AddEdge(wFace0, wFace1);
        AddEdge(wFace1, wFace2);
        AddEdge(wFace2, wFace0);

        memcpy(&v1, m_pTris[i].pV0, sizeof(PVERTEX));
        memcpy(&v2, m_pTris[i].pV2, sizeof(PVERTEX));
        memcpy(&v3, m_pTris[i].pV1, sizeof(PVERTEX));

        // Frontfacing Tris formen die Front-Cap
        m_pVolume[m_NumVV++] = v1;
        m_pVolume[m_NumVV++] = v2;
        m_pVolume[m_NumVV++] = v3;

        // Back-Cap wird ausgezogen
        m_pVolume[m_NumVV++] = Extrude(m_pTris[i].pV0,vc);
        m_pVolume[m_NumVV++] = Extrude(m_pTris[i].pV1,vc);
        m_pVolume[m_NumVV++] = Extrude(m_pTris[i].pV2,vc);

        m_pIsCap[i] = true;
    }
    else m_pIsCap[i] = false;
} // for

int nFact = m_nSize/sizeof(BYTE);

// 2. PASS: loope durch die Liste, und ziehe
//         alle Kanten zu echten Rechtecken aus
for (UINT j=0; j<m_NumE; j++) {
    v3 = Extrude( (VERTEX*)&m_pVerts[ m_pEdges[2*j+0]
                * nFact ], vc);
    v4 = Extrude( (VERTEX*)&m_pVerts[ m_pEdges[2*j+1]
                * nFact ], vc);

    memcpy(&v1, (VERTEX*)&m_pVerts[ m_pEdges[2*j+0]
                * nFact ], sizeof(PVERTEX));
    memcpy(&v2, (VERTEX*)&m_pVerts[ m_pEdges[2*j+1]
                * nFact ], sizeof(PVERTEX));

    // füge ein Rechteck zur Vertexliste hinzu
    m_pVolume[m_NumVV++] = v1;
    m_pVolume[m_NumVV++] = v2;
    m_pVolume[m_NumVV++] = v3;

```

```

        m_pVolume[m_NumVV++] = v2;
        m_pVolume[m_NumVV++] = v4;
        m_pVolume[m_NumVV++] = v3;
    }
} // BuildShadowVolume

```

Das Shadow-Volume rendern

Einfache Implementierung

Das Rendern des Shadow-Volumes werden wir ganz einfach implementieren. Wir werden insbesondere nicht prüfen, ob sich das Objekt oder die Lichtquelle bewegt hat. Wenn ein Shadow-Volume zum Rendern aufgerufen wird, dann muss man ihm eine Lichtquelle in Weltkoordinaten angeben und die Weltmatrix der Geometrie, zu der das Volume gehört. Wird die Matrix nicht angegeben, dann muss diese Geometrie bereits ebenfalls in Weltkoordinaten vorliegen.

Falls sowohl die Lichtquelle als auch die Geometrie in Weltkoordinaten vorliegen, müssen wir einfach den Vektor vom Mittelpunkt der Bounding-Box der Geometrie zur Lichtquelle hin berechnen. Wenn die Geometrie aber in lokalen Koordinaten vorliegt, dann müssen wir zuerst die Position der Lichtquelle mit der inversen Weltmatrix der Geometrie in den lokalen Raum der Geometrie transformieren. Da der Mittelpunkt der Geometrie im lokalen Raum natürlich der Ursprung ist, stellt die Position der Lichtquelle im lokalen Raum gleichzeitig selbst den Vektor vom Objekt zur Lichtquelle hin dar.

Objekt im Radius des Lichts?

Mit dem Vektor zur Lichtquelle berechnen wir dann die Distanz, die das Objekt zur Lichtquelle hat. Wenn diese größer ist als der Radius der Lichtquelle, so brechen wir die Funktion gleich ab, denn die Lichtquelle wirft gar kein Licht auf das betrachtete Objekt. Dieser Test ist natürlich nicht vollkommen korrekt, da wir eigentlich auch den Radius des Objekts prüfen müssten und nicht nur von seinem Mittelpunkt aus.

Rendern

Ist diese Bedingung erfüllt, dann berechnen wir das Shadow-Volume mit der eben gezeigten Funktion neu. Dann rendern wir die Backfaces des Volumes und in einem zweiten Pass die Frontfaces des Volumes. Die Methode sieht wie folgt aus:



```

HRESULT CShadowVolume::Render( CGameLight *pLight,
                               ZFXMatrix *pMat ) {
    HRESULT hr=ZFX_OK;
    ZFXVector vcLookAtLight;

    ZFXVector vcLightPos = pLight->GetPosition();

    // transformiere Licht zu Modell, falls nötig
    if (pMat) {

```

```

    ZFXMatrix mat; mat.InverseOf( *pMat );
    vcLightPos = mat * vcLightPos;
    vcLookAtLight = vcLightPos;
}
else {
    vcLookAtLight = vcLightPos
                  - m_Aabb.vcCenter;
}

// Fällt überhaupt Licht auf das Objekt?
float fDist_2 = vcLookAtLight.GetSqrLength();
float fRadi_2 = pLight->GetSqrRadius()*2;

if ( fDist_2 > fRadi_2 ) return ZFX_OK;

// erzeuge das Shadow-Volume
BuildShadowVolume(vcLookAtLight);

if ( pMat ) m_pDevice->SetWorldTransform( pMat );

// FIRST PASS: RENDER DER BACKFACES
m_pDevice->SetStencilBufferMode(
    RS_STENCIL_ZFAIL_INCR, 0);

m_pDevice->SetBackfaceCulling(RS_CULL_CW);

hr = m_pDevice->GetVertexManager()->RenderNaked(
    m_NumVV, m_pVolume, false);

// SECOND PASS: RENDER DER FRONTFACES
m_pDevice->SetStencilBufferMode(
    RS_STENCIL_ZFAIL_DECR, 0);

m_pDevice->SetBackfaceCulling(RS_CULL_CCW);

hr = m_pDevice->GetVertexManager()->RenderNaked(
    m_NumVV, m_pVolume, false);

if ( pMat ) m_pDevice->SetWorldTransform( NULL );

return hr;
} // Render

```

Zwei Dinge, die unser *ZFXRenderDevice* nun anbietet, sollten uns neu sein. Zunächst haben wir da die neue Methode des Vertex-Cache-Managers *ZFXVertexCacheManager::RenderNaked*. Diese Methode dient dazu, nackte Vertices, die nur eine Position enthalten, direkt zu rendern. Das ist genau das Richtige für unser *Shadow-Volume* mit seinen *PVERTEX*-Vertices. Die zweite Neuerung sind die beiden Einstellungen *RS_STENCIL_ZFAIL_INCR* und *RS_STENCIL_ZFAIL_DECR* für den Stencil-Buffer. Diese inkrementieren bzw. dekrementieren den Sten-

*Neuheiten im
Render-Device*

cil-Wert im Falle eines Depth-Fails. In Direct3D-Code würde das wie folgt aussehen, und es ist auch so intern in der Engine umgesetzt:

```
pD3DDevice->SetRenderState(D3DRS_STENCILZFAIL,
                           D3DSTENCILOP_DECR);
pD3DDevice->SetRenderState(D3DRS_STENCILZFAIL,
                           D3DSTENCILOP_INCR);
```

Aktivieren des Stencil-Buffers

Damit haben wir auch schon die Depth-Fail-Variante umgesetzt. Wir rendern zuerst die Backfaces und dann die Frontfaces und setzen dabei die Stencil-Werte entsprechend bei einer negativen Rückmeldung des Depth-Buffers. Man beachte, dass wir jedoch den Stencil-Buffer in dieser Methode nicht aktivieren. Das hat den Sinn, dass wir vermutlich viele Schatten nacheinander mit denselben Einstellungen rendern werden. Daher lagern wir das Setup des Stencil-Buffers in übergeordnete Methoden aus. Die *ZFXEngine* stellt dazu im Render-Device die folgende Methode zur Verfügung:



```
void ZFXD3D::UseStencilShadowSettings(bool b) {
    ZFXMatrix matProj;

    m_pVertexMan->InvalidateStates();

    if (b) {
        m_pDevice->SetRenderState(D3DRS_ZWRITEENABLE,
                                  FALSE);
        m_pDevice->SetRenderState(D3DRS_STENCILENABLE,
                                  TRUE);
        m_pDevice->SetRenderState(D3DRS_SHADEMODE,
                                  D3DSHADE_FLAT);
        m_pDevice->SetRenderState(D3DRS_STENCILFUNC,
                                  D3DCMP_ALWAYS);
        m_pDevice->SetRenderState(D3DRS_STENCILPASS,
                                  D3DSTENCILOP_KEEP);
        m_pDevice->SetRenderState(D3DRS_STENCILFAIL,
                                  D3DSTENCILOP_KEEP);
        m_pDevice->SetRenderState(D3DRS_STENCILREF,
                                  0x1);
        m_pDevice->SetRenderState(D3DRS_STENCILMASK,
                                  0xffffffff);
        m_pDevice->SetRenderState(D3DRS_STENCILWRITEMASK,
                                  0xffffffff);
        m_pDevice->SetRenderState(D3DRS_STENCILZFAIL,
                                  D3DSTENCILOP_INCR);
        UseColorBuffer(false);
    }
    else {
        m_pDevice->SetRenderState(D3DRS_SHADEMODE,
                                  D3DSHADE_GOURAUD);
    }
}
```



```

m_pDevice->SetRenderState(D3DRS_CULLMODE,
                          D3DCULL_CCW);
m_pDevice->SetRenderState(D3DRS_ZWRITEENABLE,
                          TRUE);
m_pDevice->SetRenderState(D3DRS_STENCILENABLE,
                          FALSE);
m_pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,
                          FALSE);

UseColorBuffer(true);
SetMode(m_Mode, m_nStage);
}
} // UseStencilShadowSettings

```

Diese Renderstates sorgen dafür, dass wir den Stencil-Buffer korrekt betanken können. Wer sich im Umgang mit den Renderstates noch ein wenig unsicher fühlt, der sollte die DirectX SDK-Dokumentation öffnen und dort nachlesen, was die einzelnen Einstellungen bewirken. Man beachte hier insbesondere auch den Aufruf der Methode `ZFXRenderDevice::UseColorBuffer`: Dadurch werden im Modus für das Rendern des Shadow-Volumes in den Stencil-Buffer keine Pixel in den Color-Buffer (Back-Buffer) geschrieben.

Nachholbedarf?

Damit haben wir alles zusammen, was wir für das Shadow-Volume benötigen. Aber wie das Ganze im Gesamtkontext aussieht, das haben wir noch nicht gesehen. Daher zeige ich hier ein paar Codezeilen, mit denen man letzten Endes einen Schatten auf den Schirm bekommt. Im Folgenden nehmen wir an, dass wir eine Klasse `CModel1` und eine Instanz `pMesh` für ein Mesh von eben dieser Klasse haben, die ein Attribut `m_pShadowVolume` als Instanz der Klasse `CShadowVolume` hat. Über die Methode `CShadowVolume::UpdateGeometry` hat das Modell seine aktuellen Vertices für diesen Frame bereits seinem Shadow-Volume mitgeteilt. Die Variable `pDev` im folgenden Beispiel ist ein Zeiger auf ein korrekt initialisiertes `ZFXRenderDevice`-Objekt. Das Objekt `L` ist eine gültige Instanz einer Lichtquelle der Klasse `CGameLight`.

*Komplettes
Rendern des
Shadow-Volumes*

```

pDev->BeginRendering(true, true, true);

/* rendere jetzt alle Geometrie 'normal' */

/* nun das Shadow-Volume */
pDev->SetWorldTransform( pMesh->m_matWorld );
pDev->UseStencilShadowSettings(true);
pMesh->m_pShadowVolume->Render(L, pMesh->m_matWorld);
pDev->UseStencilShadowSettings(false);

/* und jetzt ein Rechteck für die Schattierung */
pDev->SetStencilBufferMode(RS_STENCIL_ENABLE, 0);
pDev->SetStencilBufferMode(RS_STENCIL_REF, 1);
pDev->SetStencilBufferMode(RS_STENCIL_FUNC_LESSEQUAL, 0);
pDev->SetStencilBufferMode(RS_STENCIL_PASS_KEEP, 0);

```



```
pDev->FadeScreen(0.0f, 0.0f, 0.0f, 0.5f);  
pDev->SetStencilBufferMode(RS_STENCIL_DISABLE, 0);  
  
pDev->EndRendering();
```

Im Detail Nun sollte alles klar sein. Wir rendern zuerst die Szene, wie wir es sonst auch tun würden. Danach werfen wir den Stencil-Buffer in den Einstellungen zum Rendern des Shadow-Volumes an und rendern dann auch gleich das Shadow-Volume in den Stencil-Buffer. Danach deaktivieren wir diese Einstellungen für den Stencil-Buffer wieder und stellen ihn so ein, dass er das Rendern nur noch an Stellen zulässt, an denen ein Schatten liegen soll. Dann nutzen wir unsere Methode zum Ausblenden des Bildes und rendern damit ein schwarzes Rechteck mit einer 50%igen Transparenz auf den gesamten Bildschirm. Dabei lässt der Stencil-Buffer das Rendern nur dort zu, wo ein Schatten liegen soll, und somit dunkeln wir die Pixel im Schatten effektiv durch das schwarze Rechteck ab.

Ein Wort zu Shadow-Volumes über Shader

Shader sind schnell ... Shader sind schnell – oder? An allen Ecken und Enden hört man die Leute darüber reden, und im Internet pfeifen es die Spatzen erst recht von den Servern: *Shader sind schnell. Shader sollte man immer einsetzen, wo es nur geht.* Doch gilt das auch für Shadow-Volumes? Na ja, wenn ich schon so frage, sollte die Antwort zumindest lauten, dass wir das mal genauer betrachten sollten. Es ist sehr einfach, einen Vertex-Shader zu schreiben, der alle Vertices, deren Normalenvektor von der Lichtquelle abgewandt ist, in Laufrichtung des Lichts verschiebt. Dieses Verfahren, das natürlich all die kleinen superduper Samples im Internet verwenden, funktioniert allerdings nur bei Modellen mit sehr weichen Rundungen (Stichwort Kugel oder Torus) und auch nur, wenn das Modell ohne redundante Vertices an derselben Position im Raum auskommt. Anderenfalls würde man unter Umständen das Modell auseinander reißen. In der praktischen Anwendung sind solche Show-Off-Samples daher total sinnlos und nicht anwendbar.

Robuste Vertex-Shader-Shadow-Volumes Aber natürlich kann man auch in einem Vertex-Shader ein Shadow-Volume generieren, ebenso wie man es im normalen Quellcode macht. Doch da hat man ein Problem: Beim Generieren des Volumes müssen wir ja an den Kanten, die die Silhouette bilden, jeweils ein Rechteck einfügen, um das Shadow-Volume zwischen der Front-Cap und der Back-Cap zu schließen. Aber natürlich kann man in einem Vertex-Shader keine Vertices erzeugen, geschweige denn ganze Rechtecke. Die Lösung zu dem Problem besteht darin, dass man diese potenziell benötigten Rechtecke in einem Vorbereitungsschritt erzeugen muss. Dazu verwendet man so genannte *degenerierte Quads*. Dabei handelt es sich um Rechtecke, bei denen jeweils zwei Vertices auf derselben Position liegen, wodurch das Rechteck damit zu einer Linie bzw. Kante degeneriert ist. Nun fügt man ein solches degeneriertes Quad in

sein Modell ein, und zwar an einer Kante, die zur Silhouette gehört. Jeweils zwei der vier Vertices des Rechtecks verwenden dabei die Normale jeweils eines der beiden Dreiecke, die sich diese Kante im Modell teilen. Im Vertex-Shader wird nun das Backfacing Dreieck an dieser Kante ausgezogen und mit ihm eine Kante des Quads. das damit von einem degenerierten zu einem vollwertigen Rechteck wird, das zwischen Front-Cap und Back-Cap das Shadow-Volume schließt.

Das klingt unheimlich gut, oder? Aber natürlich hat das auch Nachteile. Der größte Nachteil dabei ist der benötigte Platz. Da man die Silhouette nicht vorher bestimmen kann – es sei denn, man hat statische Objekte und statische Lichtquellen –, muss man **alle** potenziell benötigten degenerierten Quads in seine Modelle einfügen. Das bedeutet, dass man je Kante eines Modells, an der sich zwei Dreiecke des Modells treffen, ein degeneriertes Quad mit vier Vertices einfügen muss. Dabei kann man natürlich über Indices gehen, wodurch man immerhin den Overhead des redundanten Speicherns von Vertex-Daten vermeidet. Aber 64 Bit je Quad benötigt man trotzdem für die Indices, und bei großen Modellen hat man eine Menge Kanten. Ein einfacher Würfel mit sechs Flächen hat beispielsweise zwölf Kanten, und man sieht hier, dass die Anzahl der degenerierten Quads die Anzahl an Flächen im eigentlichen Modell immer übersteigt.

Nachteile

Nun können moderne Grafikkarten degenerierte *Dreiecke* erkennen und versuchen gar nicht erst, diese zu rendern. Man hat also keinen Impact auf die Fillrate zu befürchten. Trotzdem müssen die Daten erst einmal durch das Triangle-Setup auf der Grafikkarte, und das wird Performance fressen: Ein weiterer Nachteil der Shader-Variante für Shadow-Volumes.

Allerdings sollte man die Shader für diesen Job nicht pauschal verdammen. Man muss immer schauen, ob man in seiner bisherigen Applikation eher CPU-limited oder eher GPU-limited ist. Wenn man sowieso kaum Arbeit auf der Grafikkarte erledigt, dann wird die Shader-Variante sicherlich besser sein. Aber wenn die Grafikkarte ohnehin schon eine Menge Workload hat und die CPU eher unterbeschäftigt ist, dann fährt man mit der hier gezeigten CPU-Version wesentlich besser.

Vorteile

Letzten Endes entsteht der Bottleneck – also der Performance-Impact beim Einsetzen des Shadow-Volumes – oftmals durch die Fillrate des Stencil-Buffers. Auch wenn man keine Pixel in den Color-Buffer, sondern nur in den Stencil-Buffer rendert, so hat man doch eine Menge Pixel durch die Pipeline zu schicken, vor allem bei sehr vielen Shadow-Castern und sehr vielen Occludern, für die Schatten berechnet werden sollen. Letzten Endes fährt man natürlich am besten, wenn man beide Varianten zur Auswahl hat und einen echten Performance-Test in der finalen Applikation macht.

*Die Moral von der
Geschicht': Ohne
Benchmarking
geht es nicht*

Nun verlassen wir die dunkle Welt der Schatten und widmen uns wieder der Aufgabe, die uns eigentlich interessiert. Wir wollen ja immer noch unsere Level laden und als interaktive Spielumgebung anbieten. Und das machen wir nun wirklich.



Den gesamten Code dieses Projekts Klasse für Klasse und Methode für Methode hier abzdrukken ist aufgrund des Mangels an Platz unrealistisch. Von daher beschränke ich mich im Folgenden auf die wirklich relevanten Aspekte, die euch helfen werden, das Projekt als Ganzes zu durchschauen. Solche Kleinigkeiten wie z.B. das Initialisieren unserer Engine oder das Auslesen von float-Werten aus einer Binärdatei erspare ich mir hier. Natürlich findet ihr den gesamten Code auf der CD-ROM zu diesem Buch.

15.3 Einen Level laden

Performance-Kappe

Nun setzen wir uns die Performance-Kappe auf und sehen zu, dass wir unseren Level in einer Art und Weise laden, die für die Echtzeitdarstellung gut geeignet ist. Das Level-Format ist ja eher eine Art *natives* Format für den Editor. Wir haben dort alles brav Polygon für Polygon gespeichert. Grafikkarten haben aber lieber große Batzen an Vertices und Indices. Sie scheuen sich nicht im Geringsten darum, ob wir nun einzelne Polygone komplett haben oder nicht. Wir müssen beim Laden des Levels also einiges an Umsortierungen vornehmen. Das heißt, wir müssen die Vertex- und Index-Listen, die in der Datei pro Polygon gespeichert sind, zu je einer großen Liste umstrukturieren.

Big Picture

Das wäre jedenfalls der Idealfall. Wir werden aber etwas pragmatischer vorgehen und jeweils eine große Vertex-Liste je Sektor erzeugen. Grundsätzlich arbeiten wir hier also beim Rendern nun auch je Sektor. Ein Sektor speichert diejenigen Daten, die er zum Rendern benötigt. Zudem unterscheiden wir noch nach verwendeten Skins in einem Sektor. Das bedeutet: Wenn in einem Sektor vier verschiedene Skins von den Polygonen benutzt werden, dann legen wir vier separate Index-Listen für die Vertex-Liste des Sektors an. Dies verdeutlicht die Abbildung 15.7 noch einmal grafisch.

Umsortierungen

Wie ihr seht, müssen wir dabei einiges umsortieren, da wir nicht einfach Polygon für Polygon laden und in einer großen Polygonliste ablegen können. Daher sind die Methoden zum Laden eines Levels auch ein wenig komplex, denn die Daten müssen ja auch in das richtige Format gebracht werden. Zusätzlich werden wir auch jedem Sektor einen eigenen Octree verpassen, der für Kollisionsabfragen benutzt wird. Dazu können wir jedoch nicht die Vertex-Strukturen verwenden, die in der Level-Datei enthalten sind. Stattdessen müssen wir die Positionsdaten der Vertices aller Polygone nochmals aufbereiten und zusätzlich zu den Vertex-Listen in Vektor-Listen bzw. `ZFXPolygon`-Objekten speichern. Denn nur aus solchen Objekten kön-

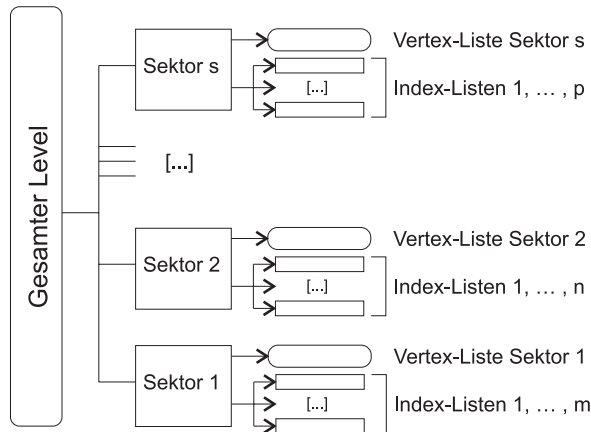


Abbildung 15.7:
Anordnung unserer Daten für einen Level. Die obere Gliederungsebene sind die Sektoren des Levels. Jeder Sektor erhält einen einzelnen statischen Vertex-Buffer für alle Vertices, die zu seinen Polygonen gehören. Zusätzlich hat jeder Sektor genau einen statischen Index-Buffer pro unterschiedlicher Skin.

nen wir unabhängig vom tatsächlichen Vertex-Format unseren Octree oder auch einen eventuellen BSP-Baum erzeugen.

Wir haben also eine Menge zu tun. Und damit fangen wir auch gleich an, bevor die Seiten hier zu Ende gehen und wir immer noch keine Umsetzung unseres Plans haben. Aber keine Panik, das Setup der Daten für eine gute Performance im Spielverlauf ist das Komplexeste, was uns im Game-Code begegnen wird.

Benötigte Hilfsdatenstrukturen

Während unserer Sortierungs- und Umräumarbeiten benötigen wir eine kleine Unterstützung bei der Gruppierung bestimmter Daten. Dazu haben wir eine Struktur `SECBUF`, wobei diese Abkürzung für *Sector's Buffer* steht. In dieser Struktur speichern wir die SkinID, die Anzahl an Dreiecken und die ID des statischen Buffers in unserer Render-Device-DLL für die Index-Buffer, die wir für jede unterschiedliche Skin in jedem Sektor anlegen. Als Zweites benötigen wir eine Struktur für einzelne Polygone. Wir haben zwar die Klasse `ZFXPolygon`, aber diese arbeitet ja auf der Basis von Vektoren. Diese Struktur benötigen wir, um auf der Basis von Vertices die Polygone aufzunehmen. Das ist nötig, weil wir zwar die Vertices und Indices später in eigenen großen Buffern in jedem Sektor abspeichern, aber zuvor müssen wir die Polygone doch noch einzeln haben, um sie nach ihrer SkinID sortieren zu können.

*Schon wieder
Polygon-
Strukturen*

```
typedef struct SECBUF_TYPE {
    UINT SkinID;
    UINT NumTris;
    UINT IBID;
} SECBUF;
```



```
typedef struct POLYGON_TYPE {
    VERTEX *pVerts;
    WORD    *pIndis;
    UINT     NumV;
    UINT     NumI;
    UINT     SkinID;
} POLYGON;
```

Lade-Methode in CGameLevel

Der leichte Part

Die vor uns liegende Aufgabe lösen wir getreu dem alten Prinzip »Teile und herrsche« in kleinen Teilen. Zuerst beginnen wir auf der Ebene des Levels, die Lade-Methode auseinander zu nehmen. In der Klasse `CGameLevel` findet sich die folgende Lade-Methode:



```
bool CGameLevel::Load(const char *pchName) {
    UINT     NumSkins    = 0;
    UINT     DummyUINT   = 0;

    FILE *pFile = fopen(pchName, "rb");

    if (!pFile) return false;

    fread(&m_NumSectors, sizeof(UINT), 1, pFile);
    fread(&DummyUINT,    sizeof(UINT), 1, pFile);
    fread(&NumSkins,     sizeof(UINT), 1, pFile);

    m_pSectors = new LPGAMESECTOR[m_NumSectors];

    // lade alle Sektoren aus der Datei
    for (UINT s=0; s<m_NumSectors; s++) {
        m_pSectors[s] = new CGameSector(m_pDevice, s,
                                         this);
        m_pSectors[s]->Load(pFile);
        m_NumSpawns += m_pSectors[s]->GetNumSpawns();
    }

    // lade alle Skins aus der Datei
    for (UINT sn=0; sn<NumSkins; sn++) {
        LoadSkin( m_pDevice->GetSkinManager(), pFile );
    }

    fclose(pFile);

    BuildAllPortalRelationships();
    return true;
} // Load
```

Hier begegnet uns nichts wirklich Neues. Wir ziehen einfach die Informationen Stück für Stück aus der Datei heraus. Man vergleiche hierzu die Lade-Methoden unseres Editors. Die Sektoren haben ihre eigenen Lade-Methoden, und wie wir dann die Skins laden, haben wir auch schon im Level-Editor gesehen. Interessant ist hier noch der Aufruf der Methode `CGameLevel::BuildAllPortalRelationships`. Diese sehen wir uns später noch an. Zuerst wollen wir den Level erst mal fertig laden. Denn wie wir schon vermuten konnten, spielt sich der Hauptteil der Arbeit in der Klasse `CGameSector` ab.

*Vergleiche
Level-Editor*

Lade-Methode in CGameSector

Das weitere Laden aus der Datei gestaltet sich auf den ersten Blick sehr einfach. Die folgende Methode in der Klasse `CGameSector` hat noch keine der angesprochenen Haken und Ösen bezüglich des Umschichtens und Neuansordnens der Daten. Ganz sauber und einfach lesen wir hier zunächst den Header eines Sektors aus und laden dann nach und nach die Meshs, Polygone, Portale usw. mit jeweils einer eigenen Methode.

*Ein Kinderspiel –
auf den ersten
Blick*

```
bool CGameSector::Load(FILE *pFile) {
    WORD      *pDummyWORD = NULL;
    UINT       NumPolys=0, NumPolysM=0, nSkin=0;
    UINT       NumV=0, NumI=0;
    int        nName      = 0;
    char        chName[2048];

    fread(&nName, sizeof(int), 1, pFile);
    fread(chName, nName, 1, pFile);
    chName[nName] = '\0';

    fread(&m_NumSkins,  sizeof(UINT),  1, pFile);
    fread(&m_NumVerts,  sizeof(UINT),  1, pFile);
    fread(&m_NumIndis,  sizeof(UINT),  1, pFile);
    fread(&NumPolys,    sizeof(UINT),  1, pFile);
    fread(&NumPolysM,   sizeof(UINT),  1, pFile);
    fread(&m_NumMeshs,  sizeof(UINT),  1, pFile);
    fread(&m_NumPorts,  sizeof(UINT),  1, pFile);
    fread(&m_NumLights, sizeof(UINT),  1, pFile);
    fread(&m_NumEntys,  sizeof(UINT),  1, pFile);
    fread(&m_NumSpawns, sizeof(UINT),  1, pFile);
    fread(&m_Aabb,      sizeof(ZFxAabb), 1, pFile);
    fread(&m_Ambient,   sizeof(ZFXCOLOR),1, pFile);

    // allokiere Speicher für alle Listen
    m_pVerts = new VERTEX[m_NumVerts];
    m_pIndis = new WORD[m_NumIndis];
    m_pBuffer = new SECBUF[m_NumSkins];
}
```



```

m_NumZFXPolys = NumPolysM + NumPolys;
m_pZFXPolys = new ZFXPolygon[m_NumZFXPolys];

if ( !LoadMeshs(pFile) ) return false;

for (UINT p=0; p<NumPolys; p++) {
    if ( !LoadSinglePolygon(pFile) )
        return false;
    } // for

if ( !LoadPortals(pFile) )    return false;
if ( !LoadLights(pFile) )    return false;
if ( !LoadEntities(pFile) )  return false;
if ( !LoadSpawns(pFile) )    return false;
if ( FAILED( CreateOctree() ) ) return false;

if ( FAILED( CompilePolygonList() ) )
    return false;

SAFE_DELETE_A(m_pZFXPolys);
return true;
} // Load

```

Octrees und Kompilare

Das Einzige, was uns hier neu vorkommt, ist die Erstellung eines Octrees und das Kompilieren der Polygonliste am Ende dieser Methode. Alles andere kennen wir in der Form schon aus dem Level-Editor. Die Polygonliste schauen wir uns am Ende dieses Abschnitts an. Was den Octree angeht, so bringen wir das gleich hinter uns, und danach geht es an die wichtigsten der einzelnen Lade-Methoden.



```

HRESULT CGameSector::CreateOctree(void) {
    if (!m_pZFXPolys) return ZFX_FAIL;
    if (m_Count < 1) return ZFX_OK;

    m_pOctree = new ZFXOctree();

    m_pOctree->BuildTree(m_pZFXPolys, m_Count);

    // diese temporäre Liste brauchen wir nicht länger
    SAFE_DELETE_A(m_pZFXPolys);
    return ZFX_OK;
} // CreateOctree

```

Polygonliste, die Erste

Den Octree erstellen wir für den Sektor also über die Liste von ZFXPolygon-Objekten im Attribut `m_pZFXPolys` unserer Klasse `CGameSector`. Beim Laden müssen wir aus allen Polygonen des Sektors, und zwar sowohl aus den freien Polygonen im Sektor als auch aus denen, die Bestandteil eines Meshs sind, eine solche Liste von ZFXPolygon-Objekten erstellen. Das merken wir uns zunächst und schauen mal, an welcher Stelle das passiert.

Polygone aus Meshs laden

Nun wird es ein wenig haarig. Jetzt schauen wir uns an, wie wir die einzelnen Meshs aus der Datei heraus laden. Das Problem dabei ist, dass wir ein Mesh im eigentlichen Spiel gar nicht als Mesh identifizieren werden. Uns ist es an dieser Stelle total egal, ob ein Polygon Teil eines Meshs ist oder nicht. Das heißt, doch nicht ganz: Für das Rendern des Levels behandeln wir alle Polygone, als wären sie freie Polygone. Daher müssen wir die einzelnen Polygone in den Meshs auch in die globale Liste für Polygone eines Sektors einsortieren. Allerdings haben Meshs doch eine Besonderheit im Game-Code: Jedes Mesh wirft im Spiel nachher automatisch einen Schatten. Das bedeutet, dass alle Meshs, die wir im Editor definiert haben, wenn wir den Level fertig gestellt haben, Objekte im Level sind, die einen Schatten werfen. Die freien Polygone ohne Bindung an ein Mesh tun dies nicht.

*Meshs werfen
Schatten*

Die folgende Methode läuft nun durch alle Meshs, die in der Datei gespeichert sind. Jedes Mesh wird dabei einmal als Menge von POLYGON-Objekten gespeichert und in der entsprechenden Liste `m_pPolys` des Sektors gespeichert. Das erledigt allerdings die separate Methode `CGameSector::LoadSinglePolygon` für uns, denn diese Aufgabe ist ein Akt für sich. Danach kehren wir zu dem ersten Polygon eines Meshs in dieser Liste zurück und beginnen den ersten Part unserer kleinen Nachbearbeitungsaktion. Die Daten der Polygone eines Meshs speichern wir dann um zu einer Liste von Vertices und Indices je Mesh. Und mit diesen rein temporären Listen erstellen wir anschließend eine Instanz der `CShadowVolume`-Klasse. So haben wir zu jedem Mesh im Level ein eigenes Shadow-Volume. Das war auch schon alles, was wir beim Laden der Meshs aus der Level-Datei berücksichtigen müssen.

*Ablauf der
Methode*

```
bool CGameSector::LoadMeshs(FILE *pFile) {
    UINT NumPolys=0, StartPoly=0, VertexBase=0;
    UINT NumV=0, NumI=0;
    ZFXAabb Aabb;

    VERTEX *pV=NULL;
    WORD *pI=NULL;

    if (m_NumMeshs < 1) return true;

    m_pMeshShadows = new CShadowVolume[m_NumMeshs];

    for (UINT m=0; m<m_NumMeshs; m++) {
        // markiere das erste Polygon, das zu diesem Mesh
        // gehört, weil alle Polygone des Sektors in
        // einer großen Liste stehen werden
        StartPoly = m_NumPolys;

        // 1. Lade den Mesh-Header
        fread(&NumPolys, sizeof(UINT), 1, pFile);
```



```

fread(&Aabb, sizeof(ZFXAabb), 1, pFile);

// 2. Lade die Polygone des Meshs
for (UINT pm=0; pm<NumPolys; pm++)
    if ( !LoadSinglePolygon(pFile) )
        return false;

// laufe durch alle extrahierten Polygone, und
// erstelle eine temporäre Liste mit den
// kompletten Vertex-/Index-Daten dieses Meshs
// nur allein zur Erzeugung des Shadow-Volumes

for (UINT i=0, NumV=0, NumI=0; i<NumPolys; i++) {
    POLYGON *pPoly = &m_pPolys[StartPoly];

    pV = (VERTEX*)realloc(pV, sizeof(VERTEX) *
                          (NumV + pPoly[i].NumV));

    memcpy(&pV[NumV], pPoly[i].pVerts,
           sizeof(VERTEX) * pPoly[i].NumV);

    NumV += pPoly[i].NumV;

    pI = (WORD*)realloc(pI, sizeof(WORD) *
                       (NumI + pPoly[i].NumI));

    for (UINT j=0; j<pPoly[i].NumI; j++)
        pI[NumI+j] = pPoly[i].pIndis[j]+VertexBase;

    NumI += pPoly[i].NumI;
    VertexBase += pPoly[i].NumV;
}

// 3. Erzeuge das Shadow-Volume aus der temporären
// Liste mit den Daten dieses Meshs
m_pMeshShadows[m].Init(pV, NumV, pI, NumI,
                       sizeof(VERTEX), Aabb,
                       m_pDevice);

NumV = NumI = 0;
VertexBase = 0;
SAFE_FREE(pV);
SAFE_FREE(pI);
} // meshs

return true;
} // LoadMeshs

```

Indices

Hierbei ist zu beachten, dass die Indices je Mesh unbedingt neu berechnet werden müssen. In der Datei sind die Meshs so gespeichert, dass ein Polygon jeweils wieder mit dem Index 0 auf seine eigene Vertexliste zu zählen

beginnt. Für das Shadow-Volume muss die eine Indexliste für das gesamte Mesh dann aber entsprechend so gestaltet sein, dass die Indices durchgehend nummeriert sind und nicht zwischendurch immer wieder bei 0 anfangen, weil die Vertices nun ja auch hintereinanderweg in einer Liste für ein ganzes Mesh gespeichert sind.

Freie Polygone laden

Die wohl wichtigste Lade-Methode ist die zum Laden einzelner Polygone. Wie wir gerade gesehen haben, brauchen wir diese sowohl zum Laden der freien Polygone aus der Level-Datei als auch zum Laden der einzelnen Polygone aus einem Mesh-Objekt. In unserem Game-Code ist beim Rendern natürlich jedes Polygon ein Polygon wie jedes andere, daher brauchen wir nicht zwischen Polygonen aus einem Mesh und freien Polygonen zu unterscheiden.

*Ein Polygon ist ein
Polygon ist ein
Polygon*

Das Laden eines Polygons ist dabei auch ganz einfach. Wir laden zuerst den Header des Polygons aus der Datei. Dann allokieren wir Speicher für die Vertices und Indices und lesen auch diese Daten aus der Datei in ein `POLYGON`-Objekt aus. Wenn wir diese Daten haben, dann generieren wir daraus eine temporäre Liste mit Vektoren für die Positionen der Vertices und eine temporäre Liste für die Indices im `UINT`-Format und nicht mehr als `WORD`-Werte. Diese beiden Listen brauchen wir, um aus dem Polygon eine Instanz der Klasse `ZFXPolygon` zu erzeugen, die wiederum in der entsprechenden Liste `m_pZFXPolys` der Klasse `CGameSector` abgespeichert wird.

*Laden eines
Polygons*

```
bool CGameSector::LoadSinglePolygon(FILE *pFile) {
    bool      DummyBOOL = false;
    float      DummyFLOAT[2];
    ZFXAabb     DummyAABB;
    WORD       *pDummyWORD = NULL;

    // Daten-Container
    ZFXVector   *pVC      = NULL;
    unsigned int *puI      = NULL;

    if (!pFile) return false;

    // Speicher allokieren
    if ( ( m_NumPolys%100) == 0) {
        m_pPolys = (POLYGON*)realloc(m_pPolys,
                                     sizeof(POLYGON)*(m_NumPolys+100));
        if (!m_pPolys) return false;
    }

    POLYGON *pPoly = &m_pPolys[m_NumPolys];

    pPoly->pVerts = NULL;
    pPoly->pIndis = NULL;
```



```

// lade den Polygon-Header
fread(&pPoly->NumV, sizeof(UINT), 1, pFile);
fread(&pPoly->NumI, sizeof(UINT), 1, pFile);
fread(&pPoly->SkinID, sizeof(UINT), 1, pFile);
fread(&DummyAABB, sizeof(ZFXAabb), 1, pFile);
fread(&DummyBOOL, sizeof(bool), 1, pFile);
fread(DummyFLOAT, sizeof(float), 2, pFile);
fread(DummyFLOAT, sizeof(float), 2, pFile);

// Speicher allokieren
pVC = new ZFXVector[pPoly->NumV];
puI = new unsigned int[pPoly->NumI];
pDummyWORD = new WORD[pPoly->NumV*2];

pPoly->pVerts = new VERTEX[pPoly->NumV];
pPoly->pIndis = new WORD[pPoly->NumI];

// lade Vertex- und Index-Daten
fread(pPoly->pVerts, sizeof(VERTEX),
      pPoly->NumV, pFile);
fread(pPoly->pIndis, sizeof(WORD),
      pPoly->NumI, pFile);
fread(pDummyWORD, sizeof(WORD),
      pPoly->NumV*2, pFile);

// erstelle Vektoren aus den Vertices
for (UINT i=0; i<pPoly->NumV; i++) {
    pVC[i].Set(pPoly->pVerts[i].x,
              pPoly->pVerts[i].y,
              pPoly->pVerts[i].z);
}

// extrahiere Indices zu einer Liste
for (UINT j=0; j<pPoly->NumI; j++) {
    puI[j] = pPoly->pIndis[j];
}

// füge das Polygon als ZFXPolygon der globalen
// Liste aller Polygone des Sektors hinzu
m_pZFXPolys[m_Count++].Set(pVC, pPoly->NumV,
                           puI, pPoly->NumI);
m_NumPolys++;

// aufräumen
SAFE_DELETE_A(pVC);
SAFE_DELETE_A(puI);
SAFE_DELETE_A(pDummyWORD);
return true;
} // LoadSinglePolygon

```

Wenn diese Methode für alle Polygone in der Level-Datei einmal durchgelaufen ist, dann haben wir als Ergebnis alle Polygone des Levels – ob nun freie oder mesh-gebundene – einmal als `ZFXPolygon`-Objekte in der Liste des Sektors gespeichert. Diese `ZFXPolygon`-Objekte benötigen wir einzig und allein zur Erstellung des Octrees. Zum anderen haben wir die Polygone aber auch als `POLYGON`-Objekte in der Liste `m_pPolys` der Klasse `CGameSector`. Um mit diesen Daten etwas anfangen zu können, müssen wir diese Polygonliste kompilieren.

Ergebnis des Vorgangs

Polygonliste kompilieren

Nachdem wir unsere Lade-Methoden quer durch die Level-Datei gehetzt haben, sind nun alle Polygon-Daten zusammen in einer großen Liste `m_pPolys` als `POLYGON`-Objekte. Von dieser Ausgangssituation aus können wir nun geeignete Mittel verwenden, um die Polygone später effizient rendern zu können. Dabei ist es wichtig, dass wir eine gewisse Menge von Polygonen in einem Aufruf rendern können. Je nach Grafikkarte und Witterung sollte man zwischen 500 und etwa 15.000 Vertices in einem Vertex-Buffer haben und alle Dreiecke, die diese Vertices verwenden, auf einen Rutsch rendern.

Aufarbeitung der Daten – Akt 2

Als Erstes sortieren wir unsere Liste mit `POLYGON`-Objekten aber nach der `SkinID`. Schließlich können wir immer nur alle Dreiecke gleichzeitig rendern, die dieselbe Textur verwenden. Dann haben wir also alle Polygone nach Skins geordnet. Nun laufen wir durch alle Polygone und speichern alle Vertices hintereinander weg in die große Vertex-Liste für den Sektor. Bei den Indices müssen wir uns aber noch etwas mehr Arbeit machen.

Sortieren nach der Skin

Wenn die Anzahl der verschiedenen Texturen für alle Dreiecke aus einem Vertex-Buffer das restriktive Element beim Batching ist, dann kann man seine Daten mit einem etwas aufwändigeren System auch so anordnen, dass man die verschiedenen Texturen in verschiedene Stages der Grafikkarte packt. Dann verwenden alle Dreiecke mit der Textur A die Texel aus der 0. Stage, die Dreiecke mit der Textur B die Texel aus der 1. Stage usw. Je nachdem, ob man Multitexturing verwendet oder nicht, kann man so bis zu vier Texturen gleichzeitig aktiv haben und alle Dreiecke auf einen Rutsch rendern, die diese vier unterschiedlichen Texturen verwenden.

**:-)
TIPP**

Ein solches System setzt aber voraus, dass wir im Vorfeld wissen, welche Textur in welcher Stage sein wird, damit die Vertices auf die entsprechenden Texturdaten zugreifen können.

Die Indices speichern wir auch in die große Liste des Sektors ab. Davon gibt es nämlich auch zunächst einmal nur eine. Dabei merken wir uns aber immer dann, wenn eine neue `SkinID` von dem folgenden Polygon verwendet wird, bei welchem Index diese `SkinID` als Erstes auftaucht und für wie viele Indices diese Skin verwendet wird. Sobald wir feststellen, dass eine neue

Eigene Index-Buffer je Skin

Skin verwendet wird, legen wir mit den gemerkten Daten einen eigenen statischen Index-Buffer im Sektor an. Damit haben wir nach Abschluss der Prozedur für jede unterschiedliche Skin im Sektor einen eigenen Index-Buffer. Fluffig, oder? Nun können wir den ganzen Sektor, sortiert nach Skins, über seinen Vertex-Buffer und die Index-Buffer rendern. Im Attribut `m_pBuffer` haben wir dann eine Liste von `SECBUF`-Objekten gespeichert, in denen die relevanten Daten eines Index-Buffers zu finden sind: die SkinID, die Anzahl der Dreiecke in diesem Buffer und die ID des Index-Buffers in der Render-DLL.

Über Lichter und Caches

Eine Sache bliebe noch zu klären. Wir haben ja den festen Willen, unseren Level mit voll dynamischer Beleuchtung von Omni-Lights anzuzeigen. Bis jetzt können wir einen ganzen Sektor mit Texturen korrekt und performant rendern. Und wir wissen auch aus dem Kapitel über Shader, wie man die Omni-Lights auf eine beliebige Geometrie loslässt. Es wäre jedoch unklug, einen ganzen Sektor für jedes Licht ein komplettes Mal zu transformieren und zu rendern. Denn jedes Licht beleuchtet ja nur einen vergleichsweise sehr kleinen Teil des Sektors. Daher machen wir einfach Folgendes: Jede Lichtquelle erhält einen eigenen Index-Buffer. In diesem Index-Buffer, den man auch als Light-Cache bezeichnet, sind einfach nur diejenigen Dreiecke gespeichert, die innerhalb des Radius einer Lichtquelle liegen.



An dieser Stelle möchte ich mich bei Marco bedanken, der mich ganz beiläufig zu den Light-Caches inspirierte. Zudem hat er mir viele wertvolle Tipps zur performanten Umsetzung der dynamischen Omni-Lights gegeben.

Dreiecke im Licht oder nicht?

Während wir also die Indices eines Polygons bearbeiten, fassen wir immer drei Indices zu einem Dreieck zusammen. Dann geben wir jeder Lichtquelle im Sektor dieses Dreieck an und fragen die Lichtquelle, ob dieses Dreieck innerhalb ihres Radius liegt. Ist das der Fall, dann geben wir die Indices an die Lichtquelle, damit diese sie zu ihrem Cache hinzufügen kann. Dabei geben wir dem Dreieck auch die SkinID mit, denn selbst ein effizienter Light-Cache muss die Dreiecke später nach Skins geordnet rendern. Und nach all dem Gerede überlasse ich euch jetzt erst einmal den Quellcode der Methode:



```
HRESULT CGameSector::CompilePolygonList(void) {
    HRESULT      hr=ZFX_OK;
    POLYGON      *pPoly = NULL;
    ZFXVector     Tri[3];
    WORD          I[3];

    UINT          CountV=0,
                  CountI=0,
                  SkinIndis=0,
                  Start=0,
```

```

        nSkin=0;

    if (m_NumPolys < 1) return ZFX_OK;

    // Polygone nach Skin sortieren
    qsort(m_pPolys, m_NumPolys, sizeof(POLYGON),
        SortBySkin);

    for (UINT i=0; i<m_NumPolys; i++) {
        pPoly = &m_pPolys[i];

        // Vertices in eine eigene Liste kopieren
        memcpy(&m_pVerts[CountV], pPoly->pVerts,
            sizeof(VERTEX)*pPoly->NumV);

        // Indices in eine eigene Liste kopieren
        for (UINT j=0; j<pPoly->NumI; j+=3) {
            I[0] = CountV + pPoly->pIndis[j];
            I[1] = CountV + pPoly->pIndis[j+1];
            I[2] = CountV + pPoly->pIndis[j+2];

            memcpy(&m_pIndis[CountI+j], I, sizeof(WORD)*3);

            Tri[0].Set(m_pVerts[I[0]].x,
                m_pVerts[I[0]].y,
                m_pVerts[I[0]].z);

            Tri[1].Set(m_pVerts[I[1]].x,
                m_pVerts[I[1]].y,
                m_pVerts[I[1]].z);

            Tri[2].Set(m_pVerts[I[2]].x,
                m_pVerts[I[2]].y,
                m_pVerts[I[2]].z);

            // laufe durch alle Lichter
            for (UINT k=0; k<m_NumLights; k++) {
                if ( m_pLights[k].InfluencesTri(Tri) ) {
                    m_pLights[k].AddTriIndices(I[0], I[1],
                        I[2], Tri,
                        pPoly->SkinID);
                }
            } // for
        }

        // Zähler neu setzen
        CountV += pPoly->NumV;
        CountI += pPoly->NumI;
        SkinIndis += pPoly->NumI;
    }

```

```

// ändert sich die Skin für das nächste Polygon?
if ( (i == (m_NumPolys-1)) ||
      (pPoly->SkinID != m_pPolys[i+1].SkinID) ) {

    hr = m_pDevice->GetVertexManager()->
        CreateIndexBuffer(SkinIndis,
                          &m_pIndis[Start],
                          &m_pBuffer[nSkin].IBID);
    if (FAILED(hr)) return hr;

    m_pBuffer[nSkin].SkinID = pPoly->SkinID;
    m_pBuffer[nSkin].NumTris = (UINT)
        (((float)SkinIndis)/3);
    SkinIndis = 0;
    Start = CountI;
    nSkin++;
}

// erzeuge einen einzelnen statischen Buffer
hr = m_pDevice->GetVertexManager()
    ->CreateStaticBuffer(
        VID_UU, 2, CountV,
        0, m_pVerts,
        NULL, &m_BufID);

// schließe die Caches für die Lichter
for (UINT k=0; k<m_NumLights; k++) {
    m_pLights[k].CloseCaches(m_pDevice);
}

return hr;
} // CompilePolygonList

```

Mehr über Lichter

Die Klasse für die Lichter werden wir uns nicht mehr anschauen, aber die Implementierungen der Methoden sind jeweils nur schicke Vierzeiler. Die Methode `CGameLight::CloseCaches` teilt einer Instanz schließlich mit, dass keine weiteren Daten mehr zum Cache hinzukommen. Daraufhin kann die Lichtquelle vom Render-Device je einen Index-Buffer pro SkinID beantragen und merkt sich deren IDs für das spätere Rendern. Aber das werden wir in diesem Kapitel auch noch sehen.

Geladen!

Nun sind wir an der Stelle angelangt, wo wir den Level komplett geladen haben. Solche Dinge wie das Laden von Portalen und Spawn-Points erkläre ich hier nicht, denn dabei gibt es keine Haken oder Fallstricke zu beachten. Wir müssen die notwendigen Daten einfach aus der Datei ziehen. Jetzt sehen wir uns an, wie wir die Portale miteinander verbinden können, um unserer Portal-Engine Leben einzuhauchen.

Verbindungen zwischen den Portalen herstellen

Bevor wir die Portale in einem Level verbinden, möchte noch einmal rekapitulieren, auf welchem Stand unsere Portale bisher sind und was wir mit ihnen erreichen wollen. Bisher haben wir die Portale mit folgenden Daten in unserer Level-Datei:

Aktueller Stand

- ➔ Vier Vertices bilden ein Viereck für die Fläche des Portals.
- ➔ Jedes Portal hat eine level-weit einheitliche ID.
- ➔ Jedes Portal hat die ID maximal eines anderen Portals gespeichert, mit dem es sich überlappt.
- ➔ Jede Door-Entity hat die ID von maximal zwei Portalen, mit denen sie sich überlappt.
- ➔ Portale werden je Sektor in einer Liste gespeichert.

So weit, so gut; das haben wir alle im Editor schon erledigt. Dann schauen wir mal, was wir nun von unseren Portalen verlangen, damit sie im Game-Code von Nutzen sind:

- ➔ Jedes Portal sollte zu einem anderen Portal in einem beliebigen Sektor (auch demselben) führen.
- ➔ Jede Door-Entity braucht Pointer auf die sie schneidenden Portale, falls es diese gibt.
- ➔ Sollte eine Door-Entity Portale enthalten, so sind diese als inaktiv zu markieren, wenn die Tür geschlossen ist.

Diese Liste klingt eigentlich ganz vernünftig, und in der Tat ist das alles gar nicht schwierig umzusetzen. Über die Bounding-Box eines Portals kann man sehr leicht feststellen, ob es sich mit einem anderen Portal und eventuell sogar einer Door-Entity überlappt. In diesem Fall müssen einfach die entsprechenden Verbindungen hergestellt werden. Jedes Portal hat einen Zeiger auf seinen eigenen Sektor und einen Zeiger auf sein so genanntes *Bruder-Portal* (engl. brother portal). Wenn der Spieler beispielsweise ein Portal durchschreitet, dann holen wir uns einfach von dessen Bruder den Zeiger auf den Sektor und wissen damit, in welchem Sektor der Spieler sich nun befindet. Die folgende Methode der `CGameLevel`-Klasse sorgt für die Verbindung der Portale untereinander und mit den Door-Entities:

*Level als
Manager-Ebene*

```
void CGameLevel::BuildAllPortalRelationships(void) {
    for (UINT s=0; s<m_NumSectors; s++) {
        m_pSectors[s]->BuildAllPortalRelationships();
    }
} // BuildAllPortalRelationships
```



Delegieren

Wieder delegiert der Level die Arbeit nur an alle seine Sektoren. Die eigentliche Arbeit geschieht also dort in einer namensgleichen Methode, die wie folgt aussieht:



```
void CGameSector::BuildAllPortalRelationships(void) {

    // 1. Verbinde die Türen mit den Portalen
    for (UINT e=0; e<m_NumEntys; e++) {
        // falls Tür dann verbinde zu Portalen
        if ( m_ppEntities[e]->IsOfType(ENT_DOOR) ) {
            ((CGameDoor**)m_ppEntities)[e]->
                ConnectToPortals(m_pLevel);
        }
    }

    // 2. Verbinde die Portale untereinander
    for (UINT p=0; p<m_NumPorts; p++) {
        m_pLevel->FindBrother( &m_pPorts[p] );
    }

} // BuildAllPortalRelationships
```

Türen anbinden

Die Door-Entities haben ja bereits aus dem Level-Editor das Wissen darüber, ob sie sich mit einem Portal überlappen oder nicht. Falls das der Fall ist, dann kennen sie die ID des Portals. Nun macht die Methode `CGameEntity::ConnectToPortals` nichts anderes, als in dem Level nach dem Portal mit dieser ID zu suchen. Der Level sucht wiederum in seinen Sektoren, und wenn das Portal mit der gesuchten ID gefunden wurde, gibt der Level der Door-Entity einen Zeiger auf dieses Portal zurück. Die Door-Entity speichert diesen Zeiger für später, um auf das Portal zugreifen zu können. Das ist dann nötig, wenn sich die Tür öffnet bzw. schließt und das Portal aktiviert oder deaktiviert werden muss. Natürlich muss eine Door-Entity entweder gar kein Portal überlappen oder genau zwei Portale, die zusammengehören.

Portale verbinden

Das Anbinden der Türen ist also trivial und muss hier nicht gezeigt werden. Interessanter ist schon die Verbindung der Portale. Dazu läuft ein Sektor durch seine Liste mit Portalen und ruft für jedes Portal über den Level die folgende Methode auf, die seinen Bruder im Level finden soll:



```
bool CGameLevel::FindBrother(LPGAMEPORTAL pPortal) {
    if (pPortal->GetBrotherID() == 0) return true;

    for (UINT s=0; s<m_NumSectors; s++) {
        if (m_pSectors[s]->FindBrother(pPortal))
            return true;
    }
    return false;
} // FindBrother
```

Der Level sieht aber gar nicht ein, dass er die Arbeit wieder vom Sektor zurückerhält, und delegiert sie wiederum an alle seine Sektoren – es sei denn, die ID des Burder-Portals steht in einem Portal auf 0, was bedeutet, dass es keinen Bruder gibt.

Wenn ein Portal für seinen Bruder die ID mit dem Wert 0 hat, dann bedeutet dies, dass im Level-Editor kein Portal gefunden wurde, das dieses überlappt. In unserem Game-Code werten wir das nicht als Fehler, da dieser Zustand beim Testen eines halb fertigen Levels schon einmal auftreten kann. In einem echten Release müssen natürlich alle Portale verbunden sein und damit auch einen Bruder haben.

Nun sind wir wiederum in einem Sektor-Objekt gelandet und suchen dort nach dem Bruder des angegebenen Portals. Finden wir in der Liste der Portale des Sektors das Portal mit der gesuchten ID, dann rufen wir die Methode `CGamePortal::SetBrotherRelationship` auf, um die beiden Portale miteinander zu verbinden:

```
bool CGameSector::FindBrother(LPGAMEPORTAL pPortal) {
    for (UINT i=0; i<m_NumPorts; i++) {
        if ( m_pPorts[i].GetID() ==
            pPortal->GetBrotherID() )
        {
            m_pPorts[i].SetBrotherRelationship(pPortal);
            return true;
        }
    }
    return false;
} // FindBrother
```

Dabei wird einfach überprüft, ob es sich bei den beiden Portalen nicht um dasselbe handelt. Dieser Fall sollte eigentlich nicht auftreten können, aber wir gehen lieber auf Nummer Sicher. Wenn wir das evaluiert haben, dann speichern die beiden Portale einfach einen Zeiger auf das jeweils andere und sind damit offiziell als Blutsbrüder miteinander verbunden. Das war schon der gesamte Automatismus zum Verbinden der Portale für den direkten Zugriff von einem Portal auf seinen Bruder bzw. von einer Door-Entity auf die potenziell in ihr liegenden Portale.

15.4 Einen Level rendern

Nach diesem kleinen Ausflug in die Welt der Portale kehren wir zu den Wurzeln des Geschehens zurück. Beim Laden des Levels haben wir uns viel Arbeit damit gemacht, die Daten möglichst effizient in möglichst wenigen Buffern anzuordnen, damit wir beim Rendern performant bleiben können.

Pingpong



Zurück im Sektor



Zeiger um Zeiger

Back to the roots

Und genau darum geht es in diesem Abschnitt. Wir werden hier auch sehen, welche Rolle unsere Portale dabei spielen.

Rendern der Geometrie

*Erst die
Geometrie, dann
die Schatten*

Als Erstes muss unsere Klasse `CGameLevel` natürlich eine Methode bieten, mit der man den Level rendern kann. Als Parameter erwartet diese Methode die ID des Sektors, in dem sich die Kamera befindet, die Position der Kamera sowie zwei optionale Pointer, in denen die Anzahl der gerenderten Triangles und die Anzahl der gerenderten Lights zurückgegeben werden kann.



Die ID eines Sektors ist einfach sein Index im Array aller Sektoren in der Klasse `CGameLevel`. Basierend auf der Position der Kamera, kann man mit Hilfe des Octrees oder eines BSP-Baums auch berechnen, in welchem Sektor sich die Kamera befindet. Aber das ist ein zeitaufwändigeres Verfahren, als diese Information einfach zu wissen. Das Wissen kommt daher, dass der Level jedem Spieler aus der Menge der Spawn-Points einen zufälligen initialen Spawn-Point zuweist. Und mit diesem Spawn-Point erhält ein Spieler auch die ID des Sektors, in der er das Spiel beginnt. Danach muss er sich einfach merken, wenn er ein Portal durchschreitet, in welchem Sektor er dann ist.

*Mit oder ohne
Shader?*

Wir werden jetzt die dynamischen Omni-Lights nutzen, die wir als Echtzeit Per-Pixel-Lighting implementiert haben. Allerdings brauchen wir auch eine Fallback-Option, falls das Programm auf einer Grafikkarte läuft, die keine Shader unterstützt. Auf diesen Path werde ich hier nicht eingehen, denn er ist analog zum Rendern der Szene mit ambientem Licht. Die folgende Methode auf Ebene der Klasse `CGameLevel` bietet diese beiden Render-Varianten an. Die Variante mit Shadern werden wir gleich ausführlich besprechen.



```
HRESULT CGameLevel::Render(UINT nSector, UINT *pNumT,
                           UINT *pNumL, const ZFXVector &vcPos) {
    ZFXPlane Frustum[6];
    ZFXAabb aabb;
    HRESULT hr=ZFX_OK;

    m_pDevice->GetFrustum(Frustum);
    m_pDevice->SetWorldTransform(NULL);

    if (nSector >= m_NumSectors)
        return ZFX_INVALIDPARAM;

    if ( !m_pDevice->CanDoShaders() ) {
        m_pDevice->UseShaders(false);
        hr = m_pSectors[nSector]->
            RenderAmbient(Frustum, vcPos);
    }
```

```

else {
    m_pDevice->UseShaders(true);
    m_pDevice->ActivateVShader(m_VSH, VID_UU);
    m_pDevice->ActivatePShader(m_PSH);

    m_pDevice->UseTextures(false);
    m_pDevice->UseColorBuffer(false);
    hr = m_pSectors[nSector]->RenderDepth(Frustum,
                                           vcPos);

    m_pDevice->UseTextures(true);
    m_pDevice->UseColorBuffer(true);
    m_pDevice->SetDepthBufferMode(
        RS_DEPTH_READONLY);
    hr = m_pSectors[nSector]->RenderLit(
        Frustum, vcPos,
        pNumT, pNumL);

    m_pDevice->SetDepthBufferMode(
        RS_DEPTH_READWRITE);
    m_pDevice->UseShaders(false);
}
return hr;
} // Render

```

Wie ihr seht, delegieren wir hier die Arbeit nicht gleich an den entsprechenden Sektor, in dem sich die Kamera befindet. Interessanterweise stellen wir zuerst den Color-Buffer und die Texturen aus und rufen dann die Methode `CGameSector::RenderDepth` auf. Danach stellen wir die Texturen und den Color-Buffer wieder an und setzen den Depth-Buffer auf `READONLY`. Nun rendern wir den Sektor über die Methode `CGameSector::RenderLit`. Unsere Vorgehensweise hat natürlich einen handfesten Grund: *Performance*. Da wir mit mehreren Lichtquellen arbeiten, müssen wir ja zunächst einen Pass mit ambientem Licht rendern und dann je einen Pass für jede Lichtquelle. Das haben wir in Kapitel 7, *3D-Pipeline und Shader*, bereits so gemacht. Dabei ist es natürlich vollkommen unnötig, bei jedem Pass Werte in den Depth-Buffer zu schreiben. Wir rendern also zunächst die gesamte Szene einmal in den Depth-Buffer. Der ist damit komplett, und braucht nicht weiter beschrieben zu werden. Dann rendern wir den ambienten Pass und danach die einzelnen Passes pro Lichtquelle.

Depth-First

Vor allem, wenn wir Szenen mit viel Overdraw haben, wird uns das den einen oder anderen Frame pro Sekunde mehr ermöglichen.

Den Depth-Pass rendern

Nun kommen wir wirklich zu den Teilen des Game-Codes, die auch wirklich etwas rendern. Die Methode zum Rendern des Depth-Pass entspricht

Endlich rendern

quasi dem Rendern des ambienten Pass. Es ist ja nur der Color-Buffer ausgeschaltet. Das Rendern in unserer Engine erfolgt nun in drei Phasen je Sektor:

1. Rendern der Geometrie des Sektors
2. Rendern der Door-Entities des Sektors
3. Rendern der benachbarten Sektoren

In der ersten Phase rendern wir die Geometrie des Sektors, und zwar sind das alle Polygone, die im Level-Editor als freie oder mesh-gebundene Polygone erstellt wurden. Das Rendern erfolgt hier über die Index-Buffer, die je Skin angelegt wurden. Danach rendern wir die Door-Entities, die hier ein wenig stiefmütterlich behandelt werden. Wir rendern diese nämlich voll dynamisch einzeln Polygon für Polygon. Bei sehr vielen Door-Entities in einem Sektor ist das ist nicht sehr performant, und man sollte das durch eine etwas schauere Variante ersetzen. Hier brauchen wir aber diese Einfachheit, weil die Türen ja auch animiert sind. Daher muss jede Tür auch einzeln mit einer eigenen Weltmatrix gerendert werden können. Zuletzt rendern wir die benachbarten Sektoren, und dabei kommen endlich die Portale für unser Scene-Management zum Einsatz.

Portale im Einsatz

Welche anderen Sektoren in dem Level existieren, interessiert unseren Sektor gar nicht. Er kennt ja nur die Portale, die in ihm liegen. Wir laufen also durch alle Portale des Sektors. Für jedes Portal prüfen wir genau drei Bedingungen:

1. Ist das Portal innerhalb einer geschlossenen Tür?
2. Ist das Portal außerhalb des View-Frustums?
3. Ist das Portal mit seiner Vorderseite von uns abgewandt?

Sobald wir eine dieser drei Fragen mit »Ja« beantworten, verwerfen wir dieses Polygon für diesen Frame und schauen uns das nächste an. Können wir alle drei Fragen mit »Nein« beantworten, dann befindet sich das Portal aller Wahrscheinlichkeit nach im Blickfeld des Betrachters. Folglich kann er auch den Sektor sehen, zu dem dieses Portal führt. Also lassen wir uns von dem Portal einen Zeiger auf den Sektor geben, zu dem es führt, und rufen unsere Render-Methode rekursiv für diesen Sektor auf. Danach durchlaufen wir die anderen Portale unseres Sektors weiter, bis wir alle durch haben.

Herzlichen Glückwunsch: Ihr habt soeben das Prinzip einer Portal-Engine implementiert. Wenn das mal kein Pfannkuchen war.



```

HRESULT CGameSector::RenderDepth(ZFXPlane *pFrustum,
                                const ZFXVector &vcPos)
{
    static CGameSector *pSector=NULL;
    HRESULT hr=ZFX_OK;
    ZFXAabb aabb;

    // dieser Sektor wurde in diesem Frame schon
    // einmal im Depth-Pass gerendert
    if (m_bDepthRendered) return ZFX_OK;
    else m_bDepthRendered = true;

    if (m_NumZFXPolys == 0) return ZFX_OK;

    // 1. RENDERE DIE GEOMETRIE DES SEKTORS
    for (UINT i=0; i<this->m_NumSkins; i++) {

        hr = m_pDevice->GetVertexManager()->Render(
            m_BufID, m_pBuffer[i].IBID,
            m_pBuffer[i].SkinID);
        if (FAILED(hr)) return ZFX_FAIL;
    }

    // 2. RENDERE DIE DOOR-ENTITIES DES SEKTORS
    for (UINT k=0; k<m_NumEntys; k++) {
        if (!m_ppEntities[k]->IsOfType(ENT_DOOR))
            continue;
        aabb = m_ppEntities[k]->GetAabb();
        if (aabb.Cull(pFrustum, 6) == ZFXCULLED )
            continue;
        m_ppEntities[k]->Render( m_pDevice );
    }

    // 3. RENDERE BENACHBARTE SEKTOREN
    for (UINT j=0; j<m_NumPorts; j++) {
        // Portal innerhalb einer geschlossenen Tür
        if ( !m_pPorts[j].IsOpen() ) continue;

        // Das Portal ist außerhalb des View-Frustums
        if ( m_pPorts[j].GetAabb().Cull(pFrustum, 6)
            == ZFXCULLED ) continue;

        // Das Portal zeigt uns nur die Rückseite
        if ( m_pPorts[j].IsBackfacingTo(vcPos) ) continue;

        pSector = m_pPorts[j].GetTargetSector();
        if (pSector) pSector->RenderDepth(pFrustum,vcPos);
    }
    return ZFX_OK;
} // RenderDepth

```

m_bRendered

Unsere Klasse `CGameSector` hat nun einige `bool`-Attribute wie beispielsweise `m_bDepthRendered`. Wir müssen natürlich Endlos-Schleifen beim rekursiven Durchlaufen der Portale und Sektoren vermeiden. Daher hat jeder Sektor diese Flags, die auf `true` gesetzt werden, sobald der Sektor in einem Frame betrachtet wurde. Das verhindert die Endlos-Schleife. Natürlich müssen alle diese Attribute zu Beginn eines jeden Frames wieder auf `false` gesetzt werden.

Den Ambient-Pass rendern

The same again

Das Rendern des ambienten Pass entspricht beinahe dem Rendern des Depth-Pass. Hier kommt lediglich hinzu, dass wir entsprechende Vertex-Shader und Pixel-Shader aktivieren und dann dasselbe Schema wie oben durchkauen. Innerhalb dieser Methode rufen wir dann auch schon für jeden Sektor die Methode `CGameSector::RenderLitPass` auf. Dies sind die Passes für die Lichter des Sektors, zu denen wir gleich kommen werden. Die hier verwendeten Shader sind dieselben, die wir auch für das Beispiel der Omni-Lights in Kapitel 7 benutzt haben.

Das Rendern der Door-Entities ist in eine separate Methode ausgelagert worden, denn hier müssen wir natürlich auch für jede Tür analog einen ambienten Pass und je einen Pass pro Lichtquelle rendern. Diese Methode findet ihr auf der CD-ROM, aber sie gleicht dem Rendern der Light-Passes für die Geometrie des Levels.



```
HRESULT CGameSector::RenderLit(ZFXPlane *pFrustum,
                                const ZFXVector &vcPos,
                                UINT *pNumT, UINT *pNumL)
{
    static CGameSector *pSector=NULL;

    static bool bTemp=true;
    static DWORD dwTick=0;

    HRESULT hr=ZFX_OK;

    if (m_bLitRendered) return ZFX_OK;
    else m_bLitRendered = true;

    // ambierter Licht-Level für diesen Sektor
    m_pDevice->SetAmbientLight(m_Ambient.fR,
                               m_Ambient.fG,
                               m_Ambient.fB);

    // Flackern der Lichter per Zufall berechnen
    if ( (dwTick==0) && ((rand()%8)==0))
        { bTemp=false; dwTick = GetTickCount(); }
    else if ((GetTickCount() - dwTick) > 100)
        { bTemp=true; dwTick = 0; }
```



```

// 1. RENDERE GEOMETRIE DES SEKTORS
for (UINT i=0; i<this->m_NumSkins; i++) {

    m_pDevice->ActivatePShader(m_BasePSH);
    m_pDevice->ActivateVShader(m_BaseVSH, VID_UU);

    hr = m_pDevice->GetVertexManager()->Render(
        m_BufID, m_pBuffer[i].IBID,
        m_pBuffer[i].SkinID);

    if (FAILED(hr)) return ZFX_FAIL;

    if (pNumT) (*pNumT) += m_pBuffer[i].NumTris;

    m_pDevice->ActivatePShader(m_OmniPSH);
    m_pDevice->ActivateVShader(m_OmniVSH, VID_UU);

    RenderLightPass(pFrustum, m_pBuffer[i].SkinID,
        bTemp);
}

// 2. RENDERE DOOR-ENTITIES DES SEKTORS
RenderDoors(pFrustum, m_BaseVSH, m_BasePSH,
    m_OmniVSH, m_OmniPSH, bTemp);

// 3. RENDERE BENACHBARTE SEKTOREN
for (UINT j=0; j<m_NumPorts; j++) {
    // Portal innerhalb einer geschlossenen Tür
    if ( !m_pPorts[j].IsOpen() ) continue;

    // Das Portal ist außerhalb des View-Frustums
    if ( m_pPorts[j].GetAabb().Cull(pFrustum, 6)
        == ZFXCULLED ) continue;

    // Das Portal zeigt uns nur die Rückseite
    if ( m_pPorts[j].IsBackfacingTo(vcPos) ) continue;

    m_pPorts[j].ChopFrustum(pFrustum);

    pSector = m_pPorts[j].GetTargetSector();
    if (pSector)
        pSector->RenderLit(pFrustum,vcPos,pNumT,pNumL);
}

if (pNumL) (*pNumL) += m_NumLights;
return ZFX_OK;
} // RenderLit

```

*Flackernde
Lichter*

Erinnert ihr euch noch daran, dass wir im Level-Editor jeder Lichtquelle auch ein Attribut geben können, dass sie flackern soll? In dieser Methode hier berechnen wir für jeden Sektor eine Zufallsvariable, die angibt, ob das Licht jetzt gerade an oder aus ist. Das bedeutet wiederum, dass natürlich alle flackernden Lichter eines Sektors zur gleichen Zeit entweder an oder aus sind – sie flackern also synchron. Das kann erwünscht sein oder nicht. Sinnvoll ist das bei einem Kurzschluss im Stromsystem eines Sektors, weniger sinnvoll wäre es bei Fackeln oder Kerzen im Wind. Hier solltet ihr bei Bedarf eine andere Berechnungsebene für die Entscheidung finden, ob ein Licht gerade an oder aus ist.

Die Light-Passes rendern*Das Beste zum
Schluss*

Nun kommen wir endlich zu den einzelnen Passes, die wir für jede Lichtquelle durchführen müssen. Jedes Licht enthält ja einen Cache in Form eines eigenen Index-Buffers je Skin, der die Geometrie des Levels angibt, auf die das Licht Einfluss hat. Die folgende Methode läuft für eine bestimmte Skin durch alle Lichter und prüft dabei eine Reihe von Dingen ab. Das Licht darf kein Shadow-Caster sein, denn diese spenden ja nur Schatten, aber kein Licht. Wenn es ein flackerndes Licht ist, dann muss das Flackern auch gerade in diesem Frame aktiv sein. Außerdem muss das Licht überhaupt auch einen Cache für die gerade gewünschte Skin besitzen, und zu guter Letzt muss der Cache des Lichts auch im View-Frustum liegen. Sind alle diese Bedingungen für ein Licht erfüllt, dann berechnen wir die Attenuation-Matrix und rendern den Index-Buffer dieses Lichtobjekts mit dem entsprechenden Shader für ein Omni-Light.



```
void CGameSector::RenderLightPass(
    const ZFXPlane *pFrustum,
    UINT nSkin, bool bFlicker) {
    ZFXMatrix mat;
    UINT IBID=0;

    if (!m_pDevice->CanDoShaders()) return;

    m_pDevice->UseAdditiveBlending(true);

    // RENDERE EINEN ADDITIVEN PASS JE LIGHT
    for (UINT i=0; i<m_NumLights; i++) {

        // das ist nur ein Shadow-Caster
        if (m_pLights[i].IsShadowCasting())
            continue;

        // Licht ist gerade inaktiv wegen Flackern
        if (m_pLights[i].IsFlickering() && bFlicker)
            continue;
```

```

// Scheint das Licht auf diese Skin?
if ( !m_pLights[i].InfluencesSkin(nSkin, &IBID) )
    continue;

// Ist beleuchtete Geometrie im View-Frustum?
if ( m_pLights[i].GetAabb().Cull(pFrustum, 6)
    == ZFXCULLED ) {
    continue;
}

mat = CalcTransAttenNoRot(
    m_pLights[i].GetPosition(),
    m_pLights[i].GetRadius());

m_pDevice->SetShaderConstant(SHT_VERTEX,
    DAT_FLOAT, 20,
    4, (void*)&mat);

m_pDevice->SetShaderConstant(
    SHT_PIXEL, DAT_FLOAT, 0, 1,
    (void*)&m_pLights[i].GetColor());

m_pDevice->GetVertexManager()->Render(m_BufID,
    IBID, nSkin);

} // for [lights]

m_pDevice->UseAdditiveBlending(false);
} // RenderLights

```

Die Schatten im Level rendern

Das Rendern der Schatten in der Klasse `CGameLevel` sieht nun fast genauso aus wie das kleine Codebeispiel von Anfang dieses Kapitels. Es gibt also kaum noch etwas dazu anzumerken. Interessant ist aber noch, dass diese Methode zwei Pointer auf Arrays mit Instanzen der Klasse `CGameCharacter` übernimmt. Dazu muss hier gesagt werden, dass die Spieler und Bots, die jeweils von dieser Klasse erben, nicht im Level selbst gespeichert sind, sondern in der übergeordneten Klasse `CGame`. Allerdings müssen die Schatten für die Characters in Relation zu den Shadow-Castern gerendert werden, und die sind nur im Level gespeichert. Daher braucht diese Methode die Liste mit Spielern und die Liste mit Bots (siehe nächstes Kapitel), um für diese den Schatten rendern zu können.

Wie besprochen

```

HRESULT CGameLevel::RenderShadows(UINT nSector,
    const ZFXVector &vcPos,
    CGameCharacter **pPlayers, UINT NumP,
    CGameCharacter **pBots, UINT NumB) {
    ZFXPlane Frustum[6];

```



```

ZFXAabb aabb;
HRESULT hr=ZFX_OK;

m_pDevice->GetFrustum(Frustum);
m_pDevice->SetWorldTransform(NULL);

if (nSector >= m_NumSectors)
    return ZFX_INVALIDPARAM;

m_pDevice->UseStencilShadowSettings(true);

hr = m_pSectors[nSector]->RenderShadows(
                                Frustum, vcPos,
                                pPlayers, NumP,
                                pBots,    NumB);

m_pDevice->UseStencilShadowSettings(false);

m_pDevice->SetStencilBufferMode(RS_STENCIL_ENABLE,0);
m_pDevice->SetStencilBufferMode(RS_STENCIL_REF, 1);
m_pDevice->
    SetStencilBufferMode(RS_STENCIL_FUNC_LESSEQUAL, 0);
m_pDevice->
    SetStencilBufferMode(RS_STENCIL_PASS_KEEP,0);

m_pDevice->FadeScreen(0.0f, 0.0f , 0.0f, 0.5f);

m_pDevice->SetStencilBufferMode(RS_STENCIL_DISABLE,0);
return hr;
} // Render

```

Delegation an die Sektoren

Das eigentliche Rendern des Shadow-Volumes erfolgt hier wiederum in der Klasse `CGameSector`. Dort laufen wir einfach durch alle Lichter des Sektors und prüfen, welche davon Shadow-Caster sind. Haben wir einen gefunden, dann rendern wir die Shadow-Volumes der Meshs, die der Spieler-Charakter und die der Bot-Charakter. Danach besuchen wir wie gewohnt die benachbarten Sektoren, die wir durch Portale erreichen können, und rendern dort alle notwendigen Schatten rekursiv.



```

HRESULT CGameSector::RenderShadows(ZFXPlane *pFrustum,
                                   const ZFXVector &vcPos,
                                   CGameCharacter **pPlayers, UINT NumP,
                                   CGameCharacter **pBots,  UINT NumB) {
    static CGameSector *pSector=NULL;
    HRESULT hr=ZFX_OK;

    if (m_bShadowsRendered) return ZFX_OK;
    else m_bShadowsRendered = true;

```

```

// suche Shadow-Caster unter den Lights
for (UINT l=0; l<m_NumLights; l++) {
    if ( !m_pLights[l].IsShadowCasting() ) continue;

    for (UINT m=0; m<m_NumMeshs; m++)
        m_pMeshShadows[m].Render(&m_pLights[l],NULL);

    for (UINT p=0; p<NumP; p++)
        pPlayers[p]->RenderShadow( &m_pLights[l] );

    for (UINT b=0; b<NumB; b++)
        pBots[b]->RenderShadow( &m_pLights[l] );
}

// rendere benachbarte Sektoren
for (UINT j=0; j<m_NumPorts; j++) {
    // Portal innerhalb einer geschlossenen Tür
    if ( !m_pPorts[j].IsOpen() ) continue;

    // Portal außerhalb des View-Frustums
    if ( m_pPorts[j].GetAabb().Cull(pFrustum, 6)
        == ZFXCULLED ) continue;

    // Das Portal zeigt uns nur die Rückseite
    if ( m_pPorts[j].IsBackfacingTo(vcPos) ) continue;

    pSector = m_pPorts[j].GetTargetSector();
    if (pSector) pSector->RenderShadows(pFrustum,
                                         vcPos, pPlayers,
                                         NumP, pBots, NumB);
}
return ZFX_OK;
} // RenderShadows

```

Damit haben wir einen Level geladen und können ihn effizient rendern. Nun sollten wir uns aber noch schnell ein paar andere Dinge anschauen, um das Big Picture von *Pandoras Legacy* zu vervollständigen. Allerdings denke ich, das ist die richtige Stelle, um ein Bild aus dem Programm in Abbildung 15.8 zu zeigen.

*Fertig mit dem
Rendern*

15.5 Integration von Characters

In einem Computerspiel darf natürlich eines nicht fehlen: die Lebewesen, hier allgemein als *Character* bezeichnet, die in der virtuellen Spielwelt leben und agieren. In diesem Abschnitt werde ich kurz zeigen, wie wir die Characters in *Pandoras Legacy* implementieren werden.

Ich, wir und ihr

Abbildung 15.8:
Screenshot des
Spiels Pandoras
Legacy mit dynami-
scher Echtzeitbe-
leuchtung und
Realtime-Shadows



CGameCharacter, die Basisklasse für Spieler und NPCs

Basis

Für unser Spiel brauchen wir zwei Arten von Characters, nämlich Avatare für die menschlichen Spieler in unserem Netzwerk-Spiel und Avatare für die Bots – also die computergesteuerten virtuellen Lebewesen, die wir im nächsten Kapitel implementieren. Dazu werden wir die beiden Klassen `CGamePlayer` und `CGameBot` definieren. Diese beiden Klassen sind jeweils von einer Basis-klasse `CGameCharacter` abgeleitet, die im Folgenden dargestellt ist:



```
class CGameCharacter {
public:
    CGameCharacter(void);
    virtual ~CGameCharacter(void);

    virtual void Update(CGameLevel*, bool,
                       CGameCharacter**, UINT,
                       CGameCharacter**, UINT,
                       float)=0;

    HRESULT Render(void);
    HRESULT RenderShadow(CGameLight*);

    virtual void ApplyDamage(UINT, UINT OwnerID);
    bool TestCollision(const ZFXRay&, float, float*);
    HRESULT FireBullet(void);
};
```

```

HRESULT TouchAndUse(void);

// Netzwerk-Kram
UPDATEPKG GetUpdate(void);
void      SetUpdate(UPDATEPKG*);

// verschiedene Accessor-Methoden
void      SetID(UINT n) { m_ID = n; }
UINT      GetID(void)   { return m_ID; }
UINT      GetCS(void)   { return m_CS; }
ZFXAabb   GetAabb(void);
ZFXMatrix GetWorldMatrix(void);
ZFXVector GetPos(void);
ZFXVector GetRight(void);
ZFXVector GetUp(void);
ZFXVector GetDir(void);
UINT      GetEnergy(void){ return m_nEnergy; }
UINT      GetAnimation(void);
bool      IsFrozen(void){return(m_fFrozen>0.0f);}
bool      IsBot(void)   {return m_bIsBot;}
bool      IsIdle(void)  {return m_bIdle; }
void      SetRotaSpeedY(float f);
void      SetRotaSpeedX(float f);
void      SetSpeed(float f);
void      SetSlideSpeed(float f);

```

protected:

```

LPZFXRENDERDEVICE m_pRDevice;
LPZFXNETWORKDEVICE m_pNDevice;
ZFXMCEgo          *m_pMCEgo;
CZFXModel         *m_pModel;
CShadowVolume     *m_pShadow;
UINT              m_VSH;
UINT              m_PSH;
UINT              m_ID;
UINT              m_CS;
UINT              m_nEnergy;
float             m_fFrozen;
bool              m_bMoved;
bool              m_bIsFalling;
bool              m_bIdle;
bool              m_bIsBot;

void InitShadowVolume(void);
void UpdateModel(float fElapsed);
bool Intersects(CGameCharacter**, UINT);

// Netzwerk-Nachrichten
HRESULT SendMsg_TouchAndUse(void);
HRESULT SendMsg_FireBullet(void);

```

```

        HRESULT SendMsg_PlayerFrozen(UINT);
    }; // class
typedef class CGameCharacter *LPGAMECHARACTER;

```

Attribute Nicht ganz selbsterklärend sind sicherlich die beiden Attribute `m_ID` und `m_CS`. Das Attribut `m_CS` ist die Variable, in der ein Character den aktuellen Sektor (engl. current sector) speichert, also den Sektor des Levels, in dem er sich gerade befindet. Diese Information brauchen wir ja für den Spieler, um beispielsweise den Level aus seiner Perspektive rendern zu können. Das Attribut `m_ID` ist die ID des Characters, die er vom Netzwerk als Client zugewiesen bekommen hat, wenn es sich um einen menschlichen Spieler handelt. Ansonsten vergibt die Klasse `CGame` einheitliche IDs für die Bots.

Alles nur Verweise Die meisten der hier gezeigten Methoden verweisen nur auf die entsprechenden Methoden der Attribute der Funktion. So delegieren alle Steuerkommandos (wie beispielsweise das Setzen der Geschwindigkeit, die Rotation usw.) die gewünschte Aktion an den Movement-Controller. Das Rendern erfolgt über die Klasse `CZFXModel11`. Damit verbleiben nur ganz wenige interessante Sachen in dieser Klasse, die wirklich eine eigene Implementierung erfordern. Wir werden uns hier nur den Netzwerk-Teil anschauen, da dieser mit Sicherheit der interessanteste Part ist.

Netzwerknachrichten von und für Characters

UPDATEPKG Der Netzwerkcode in unserem Spiel ist sehr simpel gehalten, da wir nur wenige Nachrichten über das Netzwerk schieben werden. In einem simplen First-Person-Shooter fallen auch gar nicht so viele Informationen an, die man versenden muss. Was die Characters angeht, so schnüren wir für diese ein einfaches Update-Paket, das alle relevanten Daten enthält, die die anderen Clients erfahren müssen. Die folgende Struktur `UPDATEPKG` wird zu diesem Zweck verwendet.



```

typedef struct UPDATEPKG_TYPE {
    UINT      NID;
    ZFXVector vcPos;
    float     fRotY;
    char      nAnim;
    float     fASpd;
} UPDATEPKG;

```

Paket-Dienst Ein Character muss dort seine Netzwerk-ID eintragen, seine aktuelle Position, seine Rotation auf der Y-Achse, die Animationssequenz, die er gerade durchläuft, und die Geschwindigkeit, mit der die Animation abläuft. Allein auf Basis dieser Daten, die jeder Client von seiner Spielfigur an den Server sendet und die dieser an die Clients verteilt, können die Clients ihre Avatare, die die anderen Spieler im Netzwerk repräsentieren, korrekt anzeigen lassen. Das geschieht alles in der Klasse `CGame`. Die beiden Funktionen `GetUpdate()`

und `SetUpdate()` der `CGameCharacter`-Klasse dienen dazu, ein solches Update-Paket von einem Character zu erfragen oder den Character auf die Daten eines Update-Pakets zu setzen.

Nun kommen wir zu den anderen Nachrichten, die ein Netzwerk noch verarbeiten muss. Das regelmäßige Update aller Clients durch ein entsprechendes Paket ist ja nur die Grundaufgabe. Wenn das implementiert ist, dann kann jeder Spieler im Netzwerk alle anderen Clients auf seinem Bildschirm mit der korrekten Animation an der korrekten Position in der korrekten Ausrichtung sehen, die diese Spieler gerade haben. Aber es gibt ja auch noch andere Events, über die das gesamte Netzwerk auf dem Laufenden bleiben muss. Unsere Netzwerk-DLL definiert ja bereits drei Nachrichten: für die Neuanmeldung eines Clients, für die Abmeldung eines Clients und für die Zusendung einer ID an einen Client. Dazu definieren wir nun zusätzlich die folgenden Nachrichten:

```
const UINT MSG_UPDATE      = 4;
const UINT MSG_PLAYERLIST  = 5;
const UINT MSG_TOUCHANDUSE = 6;
const UINT MSG_FIREBULLET  = 7;
const UINT MSG_PLAYERFROZEN = 8;
const UINT MSG_BOTLIST     = 9;
const UINT MSG_BOTUPDATE   = 10;
```

Die Klasse `CGameCharacter` kann nun über das Netzwerk drei verschiedene Nachrichten absenden. Dazu hat sie drei entsprechende Methoden mit dem Präfix `SendMsg`, von denen wir uns eine hier exemplarisch ansehen. Man beachte, dass eine Instanz dieser Klasse im Attribut `m_pNDevice` einen Pointer auf ein gültiges Device-Objekt unserer Netzwerk-DLL gerichtet hat.

```
HRESULT CGameCharacter::SendMsg_TouchAndUse(void) {
    ZFXPACKAGE msgPkg;

    ZFXVector vcPos = m_pMCEgo->GetPos();

    msgPkg.nLength   = sizeof(ZFXVector);
    msgPkg.nType     = MSG_TOUCHANDUSE;
    msgPkg.nSender   = m_ID;
    msgPkg.pData     = &vcPos;

    return m_pNDevice->SendToServer(&msgPkg);
} // SendMsg_TouchAndUse
```

Die Nachricht `MSG_TOUCHANDUSE` dient dem Zweck, dass der Spieler dadurch einen Gegenstand im Level aktivieren kann. Dazu drückt er während des Spiels eine bestimmte Taste, beispielsweise  oder die rechte Maustaste. Daraufhin wird diese Nachricht generiert, und die Klasse `CGame` muss entsprechend auf diese Nachricht reagieren. Über das Netzwerk wird zusam-

*Nachrichten im
Netzwerk*



*Aktivieren einer
Tür*



*Nach Ablauf-
Schema F*

men mit der Nachricht auch die Position im Level übertragen, an der dieser Event aufgetreten ist. Nun muss man im Level diese Position prüfen und schauen, ob sich in der Nähe ein aktivierbarer Gegenstand befindet. In *Pandoras Legacy* sind lediglich die Türen solche Gegenstände. Ist eine Tür in der Nähe dieser Position, wird sie durch diese Nachricht aktiviert. So einfach ist das Programmieren von Netzwerk-Spielen. :-)

15.6 CGame, die Klasse für das Spiel

Steuerung des Ablaufs

Nun haben wir viele kleine Details gesehen, die sich auf dieses und jenes beziehen. Die Steuerung aller dieser Details auf einer übergeordneten Ebene erfolgt durch die Klasse `CGame`. Diese initialisiert die benötigten Device-Objekte unserer Engine, verwaltet eine Instanz der Klasse `CGameLevel`, kümmert sich um das Update im Netzwerk usw. Die gesamte Klasse passt leider nicht mehr in dieses Kapitel, aber ich werde die wichtigsten Dinge hier zeigen. Es ist hilfreich, wenn ihr die CD-ROM zu diesem Buch in das Laufwerk legt, falls sie dort nicht schon längst ist, und den Projektbereich zu diesem Kapitel öffnet. Werft nun einen kurzen Blick auf die Klassendeklaration von `CGame`, und lest dann im nächsten Absatz weiter.

Update des Spiels

Alle wieder da?

Nun solltet ihr einen groben Überblick darüber haben, welche Aufgabe diese Klasse hat. Als Erstes schauen wir uns die Update-Methode an, die in jedem Frame auszuführen ist. Zuerst werden die Update-Methoden der Attribute ausgeführt, sofern es diese gibt. Das Update für das Netzwerk wird dabei nur alle paar Frames einmal ausgeführt, um den Netzwerk-Traffic zu minimieren. Beim Update des Netzwerks sendet der Spieler sein Update-Paket an den Server. Der Server verteilt die Update-Pakete an alle Clients und sendet eine Liste aller Bots und deren Update-Pakete gleichsam an die Clients.



Pandoras Legacy ist als Game-Server konzipiert und nicht als dezidierter Server. Das bedeutet, dass der Server selbst auch ein Spieler ist. Nur der Server führt Bewegungen für die Bots durch (im nächsten Kapitel) und sendet Update-Pakete für die Bots. Die Clients haben natürlich auch eine eigene Liste für die Bots, empfangen aber nur Updates für sie und senden nicht selbst welche. Ansonsten ist ein Client aber voll verantwortlich für sich selbst. Der Server leitet die Updates der Clients direkt an die anderen Clients weiter und berechnet nicht etwa die Bewegung der Clients anhand von deren Input-Daten. Letzteres wäre zwar sicherer, aber auch zeitaufwändiger, da die gesamte Arbeit auf einem einzigen Rechner ausgeführt werden müsste.

Hier ist aber erst einmal die Update-Methode für die Klasse CGame:

```
void CGame::Update(void) {
    float fET = m_pTimer->GetElapsed();

    m_pTimer->Update();
    m_pLevel->Update( fET );

    m_pBullets->Update( m_pLevel,
                        (LPGAMECHARACTER*)
                        m_pPlayers,
                        m_NumP,
                        (LPGAMECHARACTER*)
                        m_pBots,
                        m_NumB,
                        fET );

    m_fDT += fET;

    ProcessNetwork();
    ProcessInput();

    // Netzwerk-Updates
    if ( m_fDT >= 0.01f ) {
        UpdateNetwork();
        m_fDT = 0.0f;
    }

    // Player updaten
    for (UINT i=0; i<m_NumP; i++) {
        m_pPlayers[i]->Update( m_pLevel,
                              m_pPlayers[i] == m_pMe,
                              (LPGAMECHARACTER*)
                              m_pPlayers, m_NumP,
                              (LPGAMECHARACTER*)
                              m_pBots, m_NumB, fET );
    }
}
```



Die `Process...()`-Methoden sind hier die interessantesten. In der Methode `CGame::ProcessInput` wird die Eingabe des Spielers an Tastatur, Maus und Joystick abgefragt und in Eingabekommandos umgewandelt. Der Avatar des Spielers wird bewegt, entsprechend wird seine Kollisionsabfrage mit dem Level durchgeführt usw. Dazu nutzen wir natürlich die Methoden des Octrees, die wir ja in Kapitel 13, *Scene-Management*, schon gesehen haben. Daher konzentrieren wir uns nun ganz auf die Methode `CGame::ProcessNetwork`.

Process...()

Aufgaben für das Netzwerk

Netzwerk-Nachrichten verdauen

In dieser Methode müssen wir nämlich in jedem Frame unsere Inbox prüfen. Solange dort Nachrichten für uns gelagert sind, fragen wir diese ab und verarbeiten sie entsprechend. Als Client müssen wir nur auf bestimmte Nachrichten reagieren; der Server hingegen muss alle Nachrichten einmal selbst wie ein Client verarbeiten, weil er ja auch ein Spieler ist, und dann muss er zusätzlich jede Nachricht, die er empfangen hat, an alle Clients weiterleiten. Sonst würden diese die Nachricht, dass sich zum Beispiel ein neuer Client angemeldet hat, ja gar nicht erhalten. Die Funktion ist also nicht viel mehr als eine riesige if-Abfrage nach der zu verarbeitenden Nachricht.



```
HRESULT CGame::ProcessNetwork(void) {
    HRESULT    hr=ZFX_OK;
    ZFXPACKAGE msgPkg = { 0, 0, 0, 0 };
    UINT       nNID=0;

    BYTE pBuffer[MAX_PKG_SIZE];
    memset(pBuffer, 0, MAX_PKG_SIZE);
    msgPkg.pData = pBuffer;

    // frage alle Nachrichten aus der Inbox ab
    while (m_pNDevice->IsPkgWaiting()) {
        if (FAILED( m_pNDevice->GetNextPkg(&msgPkg) )) {
            hr = ZFX_FAIL;
        }
        else {
            // WIR ERHALTEN DIE ID VOM SERVER
            if (msgPkg.nType == 0) {
                m_NID = *((UINT*)msgPkg.pData);
                AddPlayer(m_NID);
                m_pMe = m_pPlayers[m_NumP-1];
            }
            // EIN NEUER CLIENT IST INS NETZ GEKOMMEN
            else if (msgPkg.nType == 1) {
                nNID = *((UINT*)msgPkg.pData);
                if (m_NID!=nNID) AddPlayer(nNID);

                // SERVER INFORMIERT DEN NEUEN ÜBER DIE
                // ANDEREN CLIENTS
                if (m_nmd==NMD_SERVER) {
                    SendMsg_PlayerListTo(nNID);
                }
            }
            // EIN CLIENT HAT SICH ABGEMELDET
            else if (msgPkg.nType == 2) {
                nNID = *((UINT*)msgPkg.pData);
                RemovePlayer(nNID);
            }
        }
    }
}
```

```

// WIR ERHALTEN EIN UPDATE-PAKET VON
// EINEM CLIENT
else if (msgPkg.nType == MSG_UPDATE) {
    nNID = msgPkg.nSender;
    UpdatePlayer(nNID,
        (UPDATEPKG*)msgPkg.pData);

    if (m_nmd==NMD_SERVER) {
        m_pNDevice->SendToClients(&msgPkg);
    }
}

// WIR ERHALTEN DIE PLAYERLISTE VOM SERVER
else if (msgPkg.nType == MSG_PLAYERLIST) {
    InitPlayerListOnClient( (CHARACTERLIST*)
        msgPkg.pData);
}

// TOUCHANDUSE-NACHRICHT
else if (msgPkg.nType == MSG_TOUCHANDUSE) {
    m_pLevel->TouchAndUse(
        *((ZFXVector*)msgPkg.pData) );
    if (m_nmd==NMD_SERVER) {
        m_pNDevice->SendToClients(&msgPkg);
    }
}

// FIREBULLET-NACHRICHT VON EINEM PLAYER
else if (msgPkg.nType == MSG_FIREBULLET) {
    m_pBullets->AddBullet(
        *((BULLET*)msgPkg.pData) );
    if (m_nmd==NMD_SERVER) {
        m_pNDevice->SendToClients(&msgPkg);
    }
}

// FROZEN-NACHRICHT VON EINEM PLAYER
else if (msgPkg.nType == MSG_PLAYERFROZEN) {
    UINT nBy = *((UINT*)msgPkg.pData);
    if (m_nmd==NMD_SERVER) {
        m_pNDevice->SendToClients(&msgPkg);
    }
}
} // if GetNextPkg() succeeded
} // while [msg empfangen]
return hr;
} // ProcessNetwork

```

Ein Beispiel

Auch hier möchte ich exemplarisch eine Methode herausgreifen, die eine komplexe Nachricht verarbeitet. Einem Client ein Update-Paket einzuspielen ist über die Klasse `CGameCharacter` ja keine Herausforderung mehr. Daher schauen wir uns einmal die Methode an, mit der der Server seine Master-Liste von Clients an einen neu angemeldeten Client schickt, damit dieser bei eine Kopie der Liste für die Darstellung der Clients auf seinem Rechner erzeugen kann.



```
HRESULT CGame::SendMsg_PlayerListTo(UINT CID) {
    // nur Server dürfen das tun
    if (m_nmd != NMD_SERVER) return ZFX_OK;

    ZFXPACKAGE    msgPkg;
    CHARACTERLIST List;
    HRESULT        hr=ZFX_OK;

    List.Num = m_NumP;

    for (UINT i=0; i<m_NumP; i++)
        List.upkg[i] = m_pPlayers[i]->GetUpdate();

    msgPkg.nLength    = sizeof(CCHARACTERLIST);
    msgPkg.nType      = MSG_PLAYERLIST;
    msgPkg.nSender    = m_NID;
    msgPkg.pData      = &List;

    return m_pNDevice->SendToClient(&msgPkg, CID);
} // SendMsg_PlayerListTo
```

Ein Job für Server

Natürlich darf nur der Server eine Liste aller Spieler und ihrer Daten verschicken. Die Clients haben ja jeweils nur Kopien dieser Liste und keine Berechtigung, Daten direkt an andere Clients zu versenden. Wir laufen einfach in einer Schleife über alle Player, die dem Server gemeldet sind, und holen uns für jeden ein Update-Paket. Diese Update-Pakete werden zusammen in eine Netzwerk-Nachricht verpackt und dann an den Adressaten, einen neu angemeldeten Client, verschickt. So ist ein Client, der neu in das Spiel eintritt, sofort darüber auf dem Laufenden, wie viele andere Spieler noch im Spiel sind, wie deren Status und Position ist usw. Auch auf die Gefahr hin, mich zu wiederholen – so einfach ist nun mal das Programmieren von Netzwerk-Spielen. :-)

15.7 Ein Blick zurück, zwei Schritt nach vorn

Deathmatch-Shooter

Wenn wir nun aus vollem Galopp zurückblicken, so sehen wir drei Dinge, die wir in diesem Kapitel bewältigt haben. Zuerst haben wir gelernt, wie man Schatten in einem Computerspiel darstellen kann, und diese auch gleich implementiert. Das effektvolle Einsetzen von Licht und Schatten

gehört heute zum A und O eines guten Spiels. Als Zweites haben wir dann gesehen, wie man eine große Menge an Daten für einen Level sinnvoll verwalten und im Speicher gruppieren kann, so dass ein performantes Rendern möglich ist. Natürlich ist das hier gezeigte Beispiel nur ein sehr einfaches System, und mir fallen spontan etliche Optimierungspotenziale ein. Die bessere Verwaltung der Door-Entites hatte ich schon angesprochen. Das Occlusion-Culling von Portalen wäre eine weitere Idee. Aber für die Kürze dieses Buches haben wir schon ein erstaunliches Spiel geschaffen. Das Dritte, was wir hier gelernt haben, ist die Programmierung eines voll funktionsfähigen Netzwerk-Spiels.

Ich hätte in diesem Kapitel gern noch ein wenig mehr Game-Code gezeigt, um die Betrachtung der besprochenen Themen hier abzurunden. Aber es liegt noch eine andere Aufgabe vor uns, die wir zu meistern haben. Außerdem sind alle nun noch anfallenden Aufgaben (beispielsweise das Aufrufen des Renderers eines Levels, die Abfrage des Inputs und die Kollisionsdetektion über den Octree) nichts anderes als das Weiterleiten der Aufrufe an die Klasse X oder Y. Solche Banalitäten könnt ihr besser anhand der CD-ROM nachvollziehen.

Selbststudium

In diesem Kapitel habe ich auch schon des Öfteren angesprochen, was als Nächstes kommt. Das folgende letzte Kapitel dieses Buches widmet sich ausführlich der Implementierung von Bots allgemein und ergänzt im Speziellen dann das hier angefangene Spiel *Pandoras Legacy* um einfache Bots. Dabei werden wir auf das Hardcoding von KI im Game-Code verzichten und stattdessen den eleganteren Weg über das Scripting gehen. Was genau das ist und warum man das bei einem kommerziellen Spieleprojekt braucht, lernt ihr im folgenden Kapitel, sobald ihr endlich umblättert.

Bots

16 Scripting und Definition von NPC-Verhalten

»Eine Maschine kann die Arbeit von fünfzig gewöhnlichen Menschen leisten, aber nicht einen einzigen außergewöhnlichen ersetzen.« (Elbert Hubbard)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- ➔ Bedeutung von NPCs und Definition ihres Verhaltens
- ➔ Einführung in Virtual Machines und Compilerbau
- ➔ Sinn des Scriptings in Computerspielen
- ➔ Vorstellung der Scriptsprache ZBL/0
- ➔ Programmierung von Bots für *Pandoras Legacy*

16.1 Kontrolle von NPCs

Eines der Features, das man heutzutage standardmäßig bei einem FPS (First Person Shooter) erwartet, ist die Möglichkeit, NPCs (Non-Player Characters) als computergesteuerte Gegner über ein Scriptsystem zu kontrollieren. Ein NPC ist ein virtueller Spieler, dessen Wahrnehmung und Verhalten im Spiel durch ein Computerprogramm gesteuert wird. In der Fachliteratur wird so eine künstlich intelligente Entität auch als *autonomer Agent* (engl. *autonomous agent*) bezeichnet.

Was sind NPCs?

*Ein Agent ist ein Programm, das über Wahrnehmung und die Fähigkeit zu (re-)agieren verfügt. Ein **autonomer Agent** (**Autonomous Agent**) ist ein Programm, das sich selbst steuert. Das heißt, seine Aktionen entstehen aus einer Verrechnung des Wissens bzw. der Erfahrung des Agenten mit einer Analyse der wahrgenommenen Umgebung des Agenten und der aktuellen Situation.*



Ein Scriptsystem, das die Kontrolle eines solchen autonomen Agenten in einer virtuellen Spielwelt erlaubt, existiert normalerweise auf zwei Ebenen. Die obere Ebene ist gemeinhin eine Scriptsprache, die in den meisten Fällen an eine klassische Programmiersprache angelehnt ist. Die zweite, untere

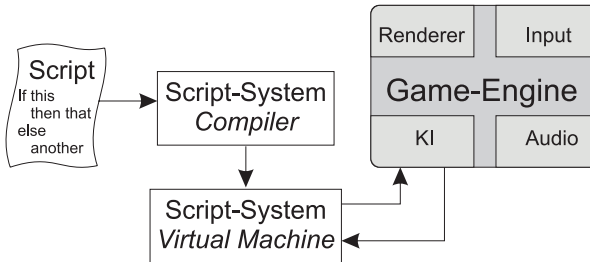
*Scriptsprache und
Scripting-Engine*

Ebene ist die dazugehörige Scripting-Engine, die in das FPS-Spiel integriert ist. Eine Scripting-Engine selbst kann ein Interpreter sein, der Scripts zur Laufzeit direkt umsetzt, oder aber eine so genannte *Virtual Machine*, die von einem Compiler in eine Zwischensprache übersetzte Scripts ausführt.



Eine *Virtual Machine* ist ein Programm, das einen Computer emuliert und somit eine Abstraktionsebene für Anwendungen über der Hardware (und dem Betriebssystem) eines Computers bereitstellt. Da das emulierte, virtuelle System nicht real existieren muss, können mit Hilfe der Virtual Machine Anwendungen für rein virtuell existierende Systeme entwickelt und ausgeführt werden. Die wohl bekannteste Virtual Machine ist die JVM (Java Virtual Machine) von Sun Microsystems, die die Runtime-Umgebung für in der Programmiersprache Java entwickelte Anwendungen bereitstellt.

Abbildung 16.1:
Schematische Darstellung der Integration eines Scripts über ein Scriptsystem in die Game-Engine bzw. deren KI-Komponente



ZBL/0 Egal, welche dieser beiden Formen das Scriptsystem eines Spiels benutzt, der Vorteil ist der, dass das Verhalten der NPCs durch simple Modifikation des Scripts verändert werden kann und es somit nicht mehr nötig ist, das Spiel selbst erneut zu kompilieren. Das Spiel *Pandoras Legacy* hat ebenso wie die meisten FPS-Spiele ein integriertes Scriptsystem für die Entwicklung von computergesteuerten Gegnern, das ZBL/0-Scriptsystem (sprich: Sybil – Zero). Das ZBL/0-Scriptsystem besteht aus einem Compiler für Bot-Programme, die mit der Scriptsprache ZBL/0 programmiert wurden, und einer Virtual Machine, die, integriert in eine Game-Engine, dort existierende NPCs steuert.

16.2 ZBL/0 ZFX Bot Language – die Scriptsprache für ZFX-Bots

Die Scriptsprache ZBL/0 lehnt an prozedurale/funktionale Hochsprachen wie C und PASCAL an.¹ Das Design von ZBL/0 ist jedoch besonders für die Definition von deterministischem (d.h. durch Regeln vorbestimmtem) Verhalten von NPCs in FPS-Spielen (Game-Bots) ausgelegt. Dabei ist das intelligent erscheinende Verhalten eines NPCs nicht aus der Sprache selbst

¹ Vgl. u.a. Kernighan, B. & Ritchie, D.: *The C Programming Language*, Prentice Hall bzw. Wirth, N. & Jensen, K., *PASCAL – User Manual and Report*, Springer-Verlag

ableitbar: Die in ZBL/0 definierten Funktionen sind nicht in der ZBL/0-Scripting-Engine selbst implementiert, sondern müssen in der Host-Anwendung (also in dem FPS-Spiel) implementiert werden. Die ZBL/0-Virtual Machine selbst kommuniziert über ein Interface mit der Host-Anwendung und kann in ihr die dort implementierten ZBL/0-Funktionen ausführen. Das als intelligent empfundene Verhalten der Game-Bots im Spiel entsteht also aus der Kombination der in den ZBL/0-Script-Programmen verwendeten ZBL/0-Befehlssequenzen mit der Implementierung der ZBL/0-Funktionen in der Host-Anwendung. Das Interface zwischen Host-Anwendung und ZBL/0-Virtual Machine ist die ZBL-API, die als C++-Interface die Integration von ZBL/0 in eigene Anwendungen erleichtert.

16.3 ZBL-API – Integrieren der ZBL/0-Virtual Machine in eigene Projekte

Die ZBL-API besteht aus zwei Komponenten: zum einen aus dem Game-Bot-Interface für die Definition der Funktionalität der Game-Bots in der virtuellen Spielwelt, und zum anderen aus dem Virtual Machine-Interface für die Ausführung von ZBL/0-Bot-Programmen in der Game-Engine.

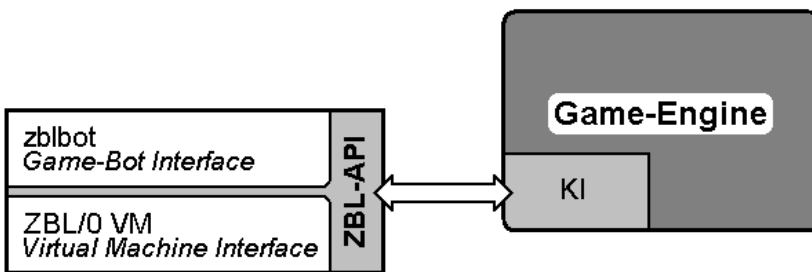


Abbildung 16.2: Die ZBL-API sitzt als Schnittstelle zwischen der Virtual Machine bzw. dem Game-Bot-Interface auf der einen Seite und der Game-Engine auf der anderen Seite.

Mehr über das ZBL-API-Interface

Das Herzstück der ZBL-API selbst ist Teil des Game-Bot-Interfaces und ist dort als abstrakte Basisklasse `zblbot` für die Definition von Game-Bots zu finden, die in der C++-Header-Datei `zblbot.h` definiert ist. Die Verwendung einer abstrakten Klasse erleichtert die Implementierung von Game-Bots in einer Game-Engine enorm. Da ein NPC-Game-Bot zu einem großen Teil dieselbe Funktionalität wie ein Spieler-Objekt hat, liegt es nahe, Game-Bots selbst von einer in der Engine implementierten Spielerklasse abzuleiten. Wenn eine solche eigene Bot-Klasse durch Mehrfachvererbung von der abstrakten Bot-Basisklasse und der Spielerklasse abgeleitet wird, lässt sich das Game-Bot-Interface der ZBL-API mit geringem Aufwand in eine existierende Game-Engine integrieren. Ein weiterer Vorteil dieser Technik ist es,

Abstrakte Basisklasse `zblbot`

dass es keine Probleme bei der Funktionsbindung zwischen Scriptfunktionen, die von der Virtual Machine ausgeführt werden, und Bot-Funktionen, die in der Game-Engine implementiert sind, gibt. Wir müssen lediglich die rein virtuellen Funktionen der abstrakten Bot-Basisklasse auch tatsächlich in der abgeleiteten Game-Bot-Klasse implementieren. Angenommen, wir haben eine Spielerklasse `CGameCharacter` und wollen eine Game-Bot-Klasse `CGameBot` erstellen, die sowohl von der Spielerklasse als auch von der Bot-Basisklasse abgeleitet ist, so wird die Klasse folgendermaßen deklariert:

```
class CGameBot : public CGameCharacter,  
                public zblbot { ... };
```

Scope Dabei müssen wir natürlich beachten, dass die Member-Attribute der Klasse `CGameCharacter`, die von Methoden der abgeleiteten `CGameBot`-Klasse direkt, d. h. nicht über den Umweg einer Interface-Methode, verwendet werden sollen, als `protected` und nicht als `private` deklariert werden müssen.

Präfixe und Marker Zur besseren Unterscheidung zwischen Methoden, die von der Spielerklasse der Game-Engine stammen, und Methoden, die von der Bot-Basisklasse abgeleitet wurden, besitzen Methoden der Bot-Basisklasse `zblbot` eine unverwechselbare Syntax. Zu diesem Zweck wird eine eindeutige, strenge Namenskonvention (engl. naming convention) in der `zblbot`-Klasse angewendet, nach der die Bezeichner von Methoden der Klasse zusammengesetzt werden. Alle Methoden und Attribute der Klasse beginnen mit dem Präfix `zb_`, das ihre Herkunft eindeutig markiert. Bei Attributen der Klasse folgt der Name des Attributs (er beginnt mit einem Kleinbuchstaben) diesem Präfix und schließt den Bezeichner ab. Bei Methoden folgt ein Marker auf dieses Präfix, der die Aufgabe der Methode beschreibt:

- ➔ `check` für Sensorabfragen
- ➔ `do` für Aktionen
- ➔ `mdf` für Modifikatoren

Zusätzlich gibt es die Marker `get`, `set` und `unSet` für Methoden, die zwar von der `zblbot`-Klasse abstammen, aber nicht direkt zum eigentlichen Bot-Interface gehören.

Abgeschlossen wird jeder Methoden-Bezeichner mit dem Namen der Methode (entspricht dem dazugehörigen ZBL/0-Standardfunktionsnamen, wenn es eine Bot-Interface-Methode ist), wobei der erste Buchstabe jeweils ein Großbuchstabe ist. Die Namen der Methoden sind weitgehend selbsterklärend, was ihre Anwendung erleichtert.

Bot-Interface-Methode	Bot-Interface-Methode
int zb_checkAlive();	double zb_checkArmour();
int zb_mdfBack();	void zb_doBackstep();
int zb_checkBlocked(int);	void zb_doCrawl();
int zb_checkDanger();	void zb_doDie();
void zb_doDuck();	int zb_doFace(int);
void zb_doFind(int);	void zb_doFire();
int zb_mdfFront();	double zb_checkHealth();
int zb_checkIdle();	int zb_checkUsing();
void zb_doJump();	void zb_doJumpBack();
void zb_doJumpLeft();	void zb_doJumpRight();
void zb_doJumpUp();	int zb_mdfLeft();
void zb_doMemorize(int);	int zb_mdfObject();
int zb_checkObjectAhead();	int zb_checkObstacle();
int zb_checkOwns(int);	void zb_doRespawn();
int zb_mdfRight();	void zb_doSpawn();
int zb_checkSpawned();	void zb_doStep();
void zb_doStrafeLeft();	void zb_doStrafeRight();
int zb_mdfTarget();	int zb_checkTargetAhead();
int zb_checkTargetAlive();	double zb_checkTargetArmour();
double zb_checkTargetHealth();	void zb_doTurn(double);
void zb_doTurnLeft();	void zb_doTurnRight();
void zb_doUse(int);	void zb_initialize(double, double, double);

Tabelle 16.1:
Bot-Interface-
Methoden der ZBL-
API (Klasse zblbot)

Tabelle 16.1 listet die Prototypen der 44 Methoden des Bot-Interfaces der ZBL-API auf. Diese Methoden entsprechen den Standardfunktionen von ZBL/0, die im Anhang genauer erklärt werden. Die exakte Implementierung jeder dieser Methoden ist von der Host-Anwendung abhängig, in die Game-Bots implementiert werden sollen, und ist somit von Anwendung zu Anwendung verschieden. Unabhängig davon soll aber einmal am Beispiel zweier einfacher Bot-Interface-Methoden verdeutlicht werden, wie eine Implemen-

tierung aussehen könnte. Angenommen, eine Spielerklasse `CGameCharacter`, von der eine Game-Bot-Klasse abgeleitet werden soll, hätte ein Attribut `m_dHealth` vom Typ `double`, das den aktuellen Gesundheitszustand des Spielers oder Game-Bots enthält:



```
class CGameCharacter
{
    protected:
        double m_dHealth;
    ...
};
```

Dieses Attribut wird bei der Ableitung der `CGameBot`-Klasse von der Spielerklasse und der ZBL-API-`zblbot`-Klasse vererbt und kann von der abgeleiteten Klasse verwendet werden.



```
#include <zblbot.h>
...
class CGameBot : public CGameCharacter, public zblbot
{
    public:
        int  zb_checkAlive(void);
        double zb_checkHealth(void);
    ...
};
```

Rein virtuelle Methoden der abstrakten Basisklasse `zblbot` müssen in der abgeleiteten Klasse deklariert werden, so auch die Funktionen `zb_checkAlive()` und `zb_checkHealth()`. Das Beispiel zeigt, wie diese beiden Funktionen implementiert werden können:



```
int CGameBot::zb_checkAlive(void)
{
    if (m_dHealth > 0.0)
        return 1;
    else
        return 0;
}

double CGameBot::zb_checkHealth(void)
{
    return m_dHealth;
}
```

Die Klasse `zblbot` enthält noch eine Reihe weiterer Attribute und Methoden, die selbst zwar nicht Teil des Bot-Interfaces, aber dennoch wichtig für die korrekte Ausführung der Game-Bots sind:

- ➔ `void zb_setProcess(int p)`: Diese Methode wird von der Virtual Machine implizit ausgeführt, wenn ein Bot-Prozess über das Virtual Machine-Interface zur Liste der Bot-Prozesse hinzugefügt wird. Sie speichert die Prozess-ID, die dem Game-Bot von der Virtual Machine zugewiesen wurde im `int`-Attribut `zb_process` der Klasse ab. Die `zb_setProcess()`-Methode sollte niemals explizit aufgerufen werden, da sonst die Prozessausführung der Virtual Machine gestört werden kann.
- ➔ `int zb_getProcess(void)`: Diese Methode liefert die Prozess-ID des Bots zurück. Falls die Methode den Wert `-1` zurückgibt, bedeutet dies, dass der Prozess suspendiert oder beendet wurde. Damit kann für jeden Game-Bot getestet werden, ob er noch aktiv ist und gerendert werden muss oder ob er einfach ignoriert werden kann. Das Attribut `char zb_busy` der Klasse `zblbot` enthält entweder den Wert `0`, wenn ein Bot-Programm aktiv ist, oder den Wert `1`, wenn ein Game-Bot in der Virtual Machine blockiert ist, weil auf die Beendigung einer früher gestarteten Aktion gewartet wird. Die Klasse `zblbot` stellt drei Methoden bereit, die Zugriff auf dieses Attribut bieten:
- ➔ `void zb_setBusy(void)`: Die `zb_setBusy()`-Methode blockiert einen aktiven Game-Bot. Diese Methode sollte aufgerufen werden, wenn der Game-Bot eine Aktion beginnt, die einen längeren Zeitraum in Anspruch nimmt und während der die Ausführung von folgenden Instruktionen des Bot-Programms gestoppt werden soll.
- ➔ `void zb_unSetBusy(void)`: Um einen blockierten Bot wieder zu aktivieren, muss man die `zb_unSetBusy()`-Methode aufrufen. Dies ist normalerweise dann der Fall, wenn eine Aktion beendet ist, die zur Blockierung des Bots geführt hatte. Es ist unbedingt notwendig, einen blockierten Bot mit `zb_unSetBusy()` zu befreien, wenn die Blockierung nicht mehr notwendig ist, da die Ausführung des Bot-Prozesses für die Dauer der Blockierung ausgesetzt wird, was in der Virtual Machine wegen unnötigen Prozesswechseln zu einem zeitverschwendenden Overhead führt.
- ➔ `char zb_getBusy(void)`: Die Methode `zb_getBusy()` gibt den aktuellen Blockierungsstatus des Game-Bots zurück. Das heißt, wenn die Methode den Wert `1` zurückerliefert, kann man davon ausgehen, dass der Bot blockiert ist.

Mehr über die Virtual Machine

Die Bot-Programme der Game-Bots selbst werden auf der ZBL/0 Virtual Machine ausgeführt. Es gibt eine Reihe verschiedener Definitionen dafür, was eine Virtual Machine ist. Die gängigste Definition (geprägt durch die Java Virtual Machine von Sun Microsystems) ist, dass eine Virtual Machine ein Programm oder ein Programmmodul ist, das als reine Softwarelösung einen Prozessor – oder in einigen Fällen auch ein komplettes Computersys-

Virtual Machines

tem – emuliert und genauso funktioniert, als ob es ein Prozessor bzw. ein Computer wäre.²

*Real Machine
versus Virtual
Machine*

Die emulierten Systeme bzw. Prozessoren können real als Hardware existieren oder aber auch imaginäre Systeme sein, für die es keine existierende Hardware gibt. Im ersten Fall dient eine Virtual Machine dazu, Software, die für ein bestimmtes System entwickelt wurde, auf einem anderen System zu benutzen, oder aber auch dazu, um auf einem System Software für ein anderes System zu entwickeln. Im Fall von imaginären Systemen kann man mit einer Virtual Machine lauffähige Programme für nicht existente Hardware entwickeln. Da es relativ unwahrscheinlich ist, dass in absehbarer Zukunft ein Coprozessor für Game-Bots entwickelt werden wird, ist eine Implementierung über den Umweg einer Virtual Machine die einzige Möglichkeit, um Programme, die in einer Sprache wie ZBL/0 geschrieben wurden, als Teil eines Computerspiels zu integrieren.

Im Fall von ZBL/0 ist die Virtual Machine kein eigenständiges Programm, sondern ein Programmmodul, das zur existierenden Game-Engine hinzugefügt wird. Die ZBL/0-Virtual Machine ist dabei als statische Bibliothek `zblvm32.lib` implementiert. Um die ZBL/0-Virtual Machine in eine Anwendung zu integrieren, muss man nur die entsprechenden Funktionen der ZBL-API aufrufen und die Bibliothek `zblvm32.lib` zur Anwendung dazulinken. Danach kann man dann Bot-Programme, die in ZBL/0 geschrieben wurden, über die Virtual Machine in der Anwendung ausführen. Wie leicht ZBL/0 in eigene Anwendungen integriert werden kann, hängt dabei zu einem großen Teil von der Architektur der ZBL/0-Virtual Machine ab.

Stack-Automaten

Stack-Automaten

Der Kern der ZBL/0-Virtual Machine ist eine Art paralleler Stack-Automat (engl. stack machine), d.h. jeder einzelne Bot-Prozess hat einen eigenen Stack. Bei Stack-Automaten werden Daten nicht in Register geladen und dort bearbeitet, sondern sie werden stattdessen auf einem Stack abgelegt, auf dem auch die Datenverarbeitung stattfindet: Die Eingangsdaten für Berechnungen, d.h. die Operanden, werden zuerst auf den Stack geschoben, und nachdem die Daten verarbeitet worden sind (nach der Operation), werden sie auf dem Stack durch das Ergebnis der Berechnung ersetzt. Dies kann als Nachteil von Stack-Automaten angesehen werden, da hier – anders als bei registerbasierten Prozessoren, bei denen die Operanden oft noch nach der Operation vorhanden sind – die Operanden während einer Operation immer verloren gehen und nur das Operationsergebnis auf dem Stack übrig bleibt. Aber Stack-Automaten haben auch einige Vorteile: Da hier keine Register vorhanden sind, gibt es auch keine komplizierte Adressierung, was die Entwicklung und Optimierung von Programmen auf der Ebene des Interpreter-Codes erleichtert.

² Vgl. Blunden, B.: *Virtual Machine Design and Implementation in C/C++*, Wordware



Eine Virtual Machine verarbeitet so genannte Bytecode-Programme, die ihre Instruktionen bereits codiert enthalten. Diese Bytecode-Programme sind Programme in der Zwischensprache, in die der entsprechende Compiler den ursprünglichen Quellcode für die Virtual Machine übersetzt hat. Wenn die Instruktionen der Zwischensprache zusätzlich in einer leserlichen, Assembler-ähnlichen Form ausgedrückt werden können, so nennt man diesen Assembler-Quellcode für die Virtual Machine **Interpreter-Code**.

Weitere Vorteile sind, dass Stack-Automaten schnellere Funktionsaufrufe erlauben als registerbasierte Prozessoren und dass Prozesswechsel beim Multitasking schnell und mit geringem Speicheraufwand vollzogen werden können. Ein gutes Beispiel zur Demonstration der Funktionalität eines Stack-Automaten ist die Addition zweier Werte A und B auf dem Stack:

```
PUSH A
PUSH B
ADD
```

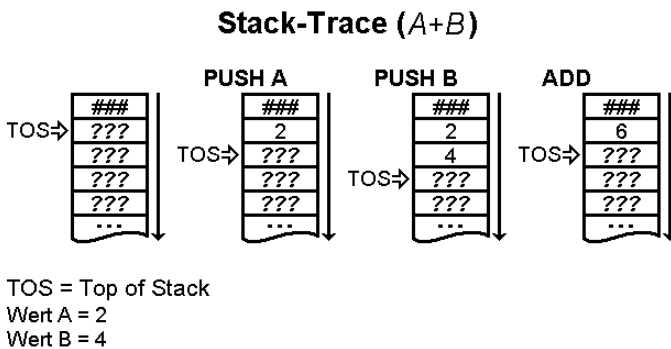


Abbildung 16.3:
Schematische Darstellung des Stack-Trace für die Operation A+B

Zuerst wird der Wert A auf den Stack geschoben, dann wird der Wert B auf den Stack geschoben, und zu guter Letzt werden die beiden Werte auf dem Stack addiert. Das bedeutet, sie werden auf dem Stack durch die Summe ihrer Addition ersetzt. Die Rechnung $A+B$ ist somit zur Rechnung $AB+$ umgeformt worden. Die Abbildung eines (arithmetischen) Ausdrucks in der sprachlich korrekten Reihenfolge $A+B$ nennt sich **Infix-Notation**. Die für den Stack-Automaten umgeformte Art der mathematischen Notation, die diesen Ausdruck als $AB+$ abbildet, nennt sich **Postfix-Notation**, wird aber manchmal auch nach der Nationalität ihres Erfinders als *umgekehrte polnische Notation* (reverse Polish Notation) bezeichnet. Als Anwendungsbeispiel für die umgekehrte polnische Notation kann die Programmiersprache *FORTH* betrachtet werden, denn sie benutzt eine Syntax, die auf der Postfix-Notation aufbaut. Ein großer Vorteil dieser Notation ist es, dass sie keine Klammerungen um arithmetische Ausdrücke für Operationspräcedenzen benötigt, da sie diese bereits implizit korrekt auflöst. Dies wird durch folgendes Beispiel verdeutlicht:

Addition auf dem Stack

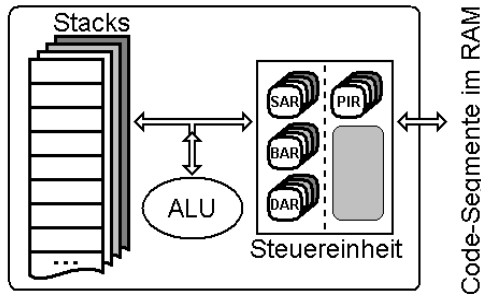
Aus dem Infix-Ausdruck $(A+B)*(C+D) \dots$

$\dots$ wird der Postfix-Ausdruck $AB+CD+*$.

Ein Stack-Automat hat eine Prozessorarchitektur, die anders funktioniert als die meisten heutzutage verwendeten CPUs, die auf Registern basieren. Stack-Automaten sind dafür aber um einiges einfacher aufgebaut und lassen sich somit auch wesentlich einfacher als eine Virtual Machine emulieren.

Abbildung 16.4:
Darstellung eines
parallelen virtuellen
Stack-Automaten

Paralleler Virtueller Stack-Prozessor



ALU - Rechenwerk (*Arithmetical Logical Unit*)
PIR - Program Instruction Register
SAR - Stack Address Register
BAR - Base Address Register
DAR - Displacement Address Register

Der Ablauf eines Programms auf einem Stack-Automaten ist um einiges einfacher organisiert als der eines registerbasierten Prozessors. Die meisten Instruktionen eines Stack-Automaten fallen dabei in eine der folgenden vier Kategorien:

- ➔ Ladeoperationen (PUSH), die einen Wert auf dem Stack ablegen
- ➔ Speicheroperationen (POP), die einen Wert vom Prozessorstack entfernen
- ➔ Ausführungsbefehle für die verschiedenen Operationen des Stack-Automaten
- ➔ Sprungbefehle für die Navigation der Programminstruktionen

Ablaufschema eines Programms auf einem Stack- Automaten

Prinzipiell würde ein Programm auf einem Stack-Automaten in etwa wie folgt ablaufen:

1. Auf dem Stack wird zunächst Speicherplatz für die globalen Programmvariablen angelegt.
2. Operanden werden auf dem Stack abgelegt.

3. Operationen werden auf die Operanden angewendet, die auf dem Stack liegen.
4. Resultate werden entweder vom Stack entfernt oder selbst als Operanden weiterverwendet.
5. Wenn ein Funktionsaufruf gefunden wird, wird auf dem Stack Speicherplatz für die lokalen Variablen der Funktion angelegt und die Instruktionen der Funktion werden abgearbeitet.
6. Wenn ein Rücksprung aus einer Funktion oder das Programmende gefunden wird, wird der vorher für Variablen allokierte Speicher wieder freigegeben.

Ein Compiler, der Code für einen Stack-Automaten generiert, muss nur dafür sorgen, dass die Befehle des ursprünglichen Programm-Codes aus der Infix-Notation der Quellsprache in die Postfix-Notation des Codes der Virtual Machine umgeformt wird, die der Stack-Automat verwendet. Dies wird von einem Compiler in einer Anzahl von aufeinander folgenden Operationen durchgeführt:³

- ➔ Die *lexikalische Analyse* (Scanner) liefert das nächste Symbol bzw. Token aus dem Quelltext des Programms – dieses Symbol kann dabei jedes Element des Wortschatzes der Sprache sein.
- ➔ Die *syntaktische Analyse* (Parser) stellt fest, ob die grammatische Reihenfolge, in der die Symbole von der lexikalischen Analyse geliefert wurden, korrekt ist, d.h., ob die Syntax vom Quelltext des Programms korrekt ist.
- ➔ Zum Schluss erfolgt die *Code-Generierung*, in der Anweisungen des Quelltextes zu Anweisungen für den Stack-Automaten aufgelöst werden und in der die Umformung von der Infix- zur Postfix-Notation der Instruktionen für den Stack-Automaten durchgeführt wird.

Die Virtual Machine als paralleler Stack-Automat

Zurück zur ZBL/0-Virtual Machine: Sie ist, wie bereits erwähnt wurde, ein paralleler Stack-Automat, d.h., mehrere Bot-Prozesse können zur gleichen Zeit auf ihr ablaufen. Aber die Daten der einzelnen Prozesse sind voneinander getrennt und unabhängig, da jeder Prozess einen eigenen Stack (sowohl ein eigenes Programmsegment als auch ein eigenes Datensegment) besitzt. Eine Scheduling-Funktion, der Kernel der Virtual Machine, sorgt dabei dafür, dass jeder Bot-Prozess entsprechend des **Round-Robin**-Prinzips einen gewissen Anteil der Prozessorzeit des Stack-Automaten zugeteilt bekommt.

³ Vgl. Bäumle, P. & Alenfelder, H.: *Compilerbau*, S+W Verlag



Round-Robin ist einer der ältesten und einfachsten Scheduling-Algorithmen, der unter Systemen mit Multitasking weit verbreitet ist. Alle laufenden Prozesse werden dabei in einer kreisförmigen Queue gehalten (das Ende der Queue ist mit ihrem Anfang verbunden), die vom Prozessor der Reihe nach abgearbeitet wird. Jedem Prozess wird vom Prozessor ein bestimmter Anteil der Zeit vom Ausführungszyklus des Prozessors zugeteilt. Nach Ablauf dieser Zeit wird ihm der Prozessor entzogen und dem nächsten Prozess in der Queue zugeteilt.

**Preemptives
Multitasking**

Der Kernel der ZBL/0-Virtual Machine hat dabei die volle Kontrolle über den aktuell ausgeführten Prozess und entscheidet auch, wann ein Prozess ausgesetzt wird und der nächste Prozess weitergeführt wird. Diese Methode der Prozessverwaltung nennt sich *preemptives Multitasking*. Bot-Prozesse werden sequenziell, d.h. einer nach dem anderen, ausgeführt. Von außen jedoch sieht es jedoch so aus, als ob in der Virtual Machine mehrere parallele Prozesse gleichzeitig ausgeführt würden.

**Das Virtual
Machine-Inter-
face der ZBL-API**

Die zweite Komponente der ZBL-API, neben dem Game-Bot-Interface, ist das Virtual Machine-Interface, das in der C++-Header-Datei `zbl_vm.h` definiert ist. Über die dort definierte Klasse `zbl_vm`, die den Kernel der Virtual Machine beinhaltet, kann eine Anwendung eine Instanz der ZBL/0-Virtual Machine erzeugen, die die Ausführung von ZBL/0-Programmen innerhalb der Anwendung erlaubt. Die Methoden des Interfaces sind in Tabelle 16.2 aufgelistet.

Tabelle 16.2:
Virtual Machine-
Interface-Metho-
den der ZBL-API
(Interface-Klasse
`zbl_vm`)

Rückgabewert	Methode
double	<code>zbl_getVersion(void);</code>
char*	<code>zbl_getVersionString(void);</code>
int	<code>zbl_addProcess(char *filename,zblbot *bot);</code>
void	<code>zbl_removeProcess(int pID);</code>
int	<code>zbl_replaceProcess(int pID,char *filename);</code>
void	<code>zbl_resetProcess(int pID);</code>
int	<code>zbl_replaceBot(int pID,zblbot *bot);</code>
void	<code>zbl_setPriority(int pID,int pr);</code>
int	<code>zbl_getPriority(int pID);</code>
int	<code>zbl_getActiveProcess(void)</code>
int	<code>zbl_run(void);</code>
int	<code>zbl_getErrors(void);</code>
void	<code>zbl_resetOnError(int pID);</code>

Rückgabewert	Methode
zbl_error_t	zbl_nextError(void);
zbl_error_t	zbl_peekError(void);
int	zbl_getWarnings(void);
zbl_error_t	zbl_nextWarning(void);

Tabelle 16.2:
Virtual Machine-
Interface-Metho-
den der ZBL-API
(Interface-Klasse
zbl_vm)
(Forts.)

Ebenso wie die Methoden des Bot-Interfaces haben natürlich auch die Methoden des VM-Interfaces (Virtual Machine-Interface) der ZBL-API eine Namenskonvention, nach der ihre Bezeichner aufgebaut sind. Präprozessor-konstanten des VM-Interfaces beginnen mit dem Präfix `ZBL_`, gefolgt vom Namen der Konstante. Die Bezeichner aller Methoden der Klasse `zbl_vm` beginnen mit dem Präfix `zbl_`. Abgeschlossen wird jeder Methoden-Bezeichner mit einem selbsterklärenden Namen der Methode. Das VM-Interface der ZBL-API enthält eine Anzahl verschiedener Methoden, die die Integration von Game-Bots in Anwendungen mit der ZBL/0-Virtual Machine erlauben. Die folgenden Methoden sind die wichtigsten:

- ➔ `double zbl_getVersion(void)`
- ➔ `char *zbl_getVersionString(void)`: Aus klar ersichtlichen Kompatibilitätsgründen ist es oftmals wichtig zu wissen, welche Version von ZBL/0 von einer Anwendung unterstützt wird. Um festzustellen, welche Version der ZBL/0-Virtual Machine installiert ist, kann man entweder die Methode `zbl_getVersion()` benutzen, die die Versionsnummer als numerischen Wert zurückliefert, oder aber die Methode `zbl_getVersionString()`, die einen `char*`-String mit der Versionsnummer der ZBL/0-Virtual Machine zurückgibt.
- ➔ `int zbl_addProcess(char *filename, zblbot *bot)`: Die `zbl_addProcess()`-Methode fügt einen Bot-Prozess zur Queue der aktiven Prozesse der Virtual Machine hinzu. Die Methode erhält zwei Parameter: Das erste Argument ist der Dateiname des Bot-Programms, das ausgeführt werden soll, und das zweite Argument ist die Speicheradresse des Game-Bots (vom Typ der abgeleiteten Klasse des Game-Bot-Interfaces), für den das Bot-Programm geschrieben wurde. Die Methode gibt die Prozess-ID des hinzugefügten Prozesses zurück.
- ➔ `void zbl_removeProcess(int pID)`: Die Methode `zbl_removeProcess()` erhält die Prozess-ID eines Bot-Prozesses als Parameter und entfernt dieses Bot-Programm aus der Prozessliste der ZBL/0-Virtual Machine.
- ➔ `void zbl_resetProcess(int pID)`: Diese Methode setzt den Bot-Prozess zurück, der durch die Prozess-ID spezifiziert wird, die als Argument übergeben wurde. Dabei wird das Bot-Programm von Anfang an neu gestartet.

- ➔ `int zbl_run(void)`: Die wichtigste Methode des VM-Interfaces der ZBL-API ist die `zbl_run()`-Methode, die die Scheduling-Funktion der ZBL/0-Virtual Machine beinhaltet und die Ausführung der Bot-Programme regelt. Die Methode `zbl_run()` muss einmal für jeden Virtual-Machine-Laufzyklus aufgerufen werden, was in einer grafischen Anwendung normalerweise einmal pro berechnetem Frame geschehen wird.

Neben der `zbl_vm`-Klasse definiert die Header-Datei `zblvm.h` auch den Strukturdatentyp `zbl_error_t`, der natürlich für die Ausgabe von Fehlermeldungen der ZBL/0-Virtual Machine nützlich ist. Die öffentlichen `public`-Attribute dieser Struktur sind:

- ➔ `int error`: Die ID des Fehlers.
- ➔ `int process`: Die Prozess-ID des Bots, in dem der gemeldete Fehler im Programmablauf aufgetreten ist.
- ➔ `address_t instruction`: Der Index des Befehls, der den Fehler verursacht hat.
- ➔ `address_t stack`: Die Stack-Adresse, bei der der gemeldete Fehler im Programmablauf aufgetreten ist.
- ➔ `char description[128]`: Ein Fehlerstring, der den Fehler in Klartext beschreibt.

Wenn ein Fehler während eines Durchlaufs der Virtual Machine auftritt, so kann er mit Hilfe der Struktur `zbl_error_t` abgefragt und dem User über eine Fehlermeldung in der Anwendung mitgeteilt werden. Dafür stehen die folgenden Methoden zur Verfügung:

- ➔ `int zbl_getErrors(void)`: Diese Methode gibt die Anzahl der in der ZBL/0-Virtual Machine aufgetretenen Fehler zurück, die noch nicht aus der Fehlerliste entfernt wurden.
- ➔ `zbl_error_t zbl_nextError(void)`
- ➔ `zbl_error_t zbl_peekError(void)`: Die Methoden `zbl_nextError()` und `zbl_peekError()` liefern beide einen Strukturwert vom Typ `zbl_error_t` zurück, der die Daten des letzten aufgetretenen Fehlers in der Fehlerliste der Virtual Machine enthält. Dabei besteht der Unterschied zwischen den beiden Methoden darin, dass `zbl_peekError()` den Fehler nicht aus der Fehlerliste entfernt.

Integration der VM in eine Applikation

Die Integration der ZBL/0-Virtual Machine mit Hilfe der Methoden des VM-Interfaces der ZBL-API ist ein unkomplizierter Vorgang. Außer der Instanziierung der Virtual Machine durch die Deklaration eines Objekts der Klasse `zbl_vm` müssen für eine Minimal-Implementation nur die Game-Bots, die in der Virtual Machine ausgeführt werden sollen, mit der Methode

`zbl_addProcess()` in die Prozessliste der Virtual Machine aufgenommen werden, und die Scheduling-Methode `zbl_run()` muss einmal pro berechnetem Frame der Anwendung aufgerufen werden. Angenommen, eine Game-Bot-Klasse `CGameBot` wäre wie folgt definiert:

```
#include <zblbot.h>
...
class CGameBot : public CGameCharacter, public zblbot
{
    ...
};
```



Dann wäre ein Game-Bot, der von der Virtual Machine benutzt werden kann, jedes Objekt, das von dieser abgeleiteten Klasse instanziiert wird. Um also ein Bot-Programm für eine Instanz von `CGameBot` in der Virtual Machine laufen zu lassen, müssen wir einen Game-Bot instanziiieren (im Beispiel die Variable `myGameBot`) und eine Instanz der ZBL/0-Virtual Machine (Variable `myVirtualMachine`) anlegen.

```
#include <zblvm.h>
...
{
    zbl_vm    myVirtualMachine; // Instanz der ZBL/0-VM
    CGameBot  myGameBot;       // Instanz des Game-Bots
    ...
}
```



Wenn der Game-Bot `myGameBot` das in ZBL/0 geschriebene Bot-Programm *botprogram.zbp* ausführen soll, muss er mit der Methode `zbl_addProcess()` in die Prozessliste der Virtual-Machine-Instanz der Klasse, also `myVirtualMachine`, hineingeladen werden.

```
...
myVirtualMachine.zbl_addProcess( "botprogram.zbp",
                                &myGameBot );
...
```



Nachdem dies erledigt wurde, muss die `zbl_run()`-Methode, die die Prozessliste der Virtual Machine durchläuft und die Prozesse ausführt, einmal pro berechnetem Frame der Anwendung bzw. einmal in jedem Durchlauf der Hauptschleife der Anwendung aufgerufen werden, um die geladenen Bot-Programme auszuführen.

```
...
myVirtualMachine.zbl_run();
...
}
```



Das ist alles, was wir tun müssen, um eine funktionierende ZBL/0-Virtual Machine in eigene Projekte zu integrieren. Alle weiteren möglichen Operationen, wie das Abfragen von Laufzeitfehlern der Virtual Machine, sowie komplexeres Prozessmanagement sind optional und nicht zwingend notwendig.

16.4 ZBL/0 Toolkit – die ZBL-Entwicklungsumgebung

Das ZBL/0 Toolkit ist die ZBL/0-Entwicklungsumgebung, die Bestandteil des ZBL/0 SDKs ist. Der Kern des ZBL/0 Toolkits ist der Compiler *zblc*. Der Compiler kompiliert ZBL-Quellcode-Dateien⁴ zu ZBL-Bytecode-Programmen für die ZBL-Virtual Machine.⁵ Optional kann der Compiler auch ein dem Bytecode eins zu eins entsprechendes Zwischen-Code-Programm in ZBL-Assembler (Datei-Endung *.zba) erzeugen, das man dann von Hand optimieren kann. Zusätzlich beinhaltet der *zblc*-Compiler sogar einen vollwertigen ZBL-Assembler. Falls der Compiler keinen *.zbl-Programmquellcode findet, aber ein entsprechendes *.zba ZBL-Assembler-Programm gefunden wird, so wird dieses vom dem im Compiler integrierten Assembler assembliert und ein entsprechendes *.zbp-Bytecode-Programm erzeugt.

Compiler als
Konsolen-
anwendung

Der *zblc*-Compiler ist natürlich keine Anwendung mit grafischem Benutzer-Interface, sondern eine zweckmäßige Konsolenanwendung mit der folgenden Aufruf-Syntax:

```
zblc <Quellcode-Datei> [Optionen]
```

Die Quellcode-Datei kann jede ASCII-Textdatei sein, die ZBL-Quellcode enthält, wobei Dateien mit der Endung *.zbl bevorzugt werden. Wird die Dateinamenendung in der Kommandozeile nicht mit angegeben, wird sie vom Compiler automatisch ergänzt. Die Kommandozeilenoptionen des Compilers sind denen der gängigen C-Compiler nachempfunden und dementsprechend ähnlich:

Tabelle 16.3:
Kommandozeilen-
optionen des zblc-
Compilers

Kommando- zeilenoption	Bedeutung
-a	Erstelle eine zusätzliche Assembler-Output-Datei
-c	Erstelle nur Assembler-Output und keinen Bytecode
-d	Erstelle eine Debug-Version vom ZBL-Bytecode
-h	Zeige eine Hilfe zur Programm-Benutzung

4 ZBL-Code-Dateien sind ASCII-Text-Dateien mit der Endung *.zbl.

5 ZBL-Bytecode-Dateien sind Binär-Dateien mit der Datei-Endung *.zbp.

Kommandozeilenoption	Bedeutung
-o <Datei>	Wähle einen Ziel-Dateinamen aus
-r	Erstelle eine Release-Version vom ZBL-Bytecode

Tabelle 16.3:
Kommandozeilenoptionen des zblc-Compilers (Forts.)

Falls beispielsweise ein Programm mit dem Namen `ZBL-Prog.zbl` zu einem Bot-Programm mit dem Namen `Bot.zbp` kompiliert werden und zusätzlich eine ZBL-Assembler-Datei erzeugt werden soll, würde die Kommandozeile folgendermaßen lauten:

```
zblc ZBL-Prog.zbl -a -o Bot.zbp
```

Des Weiteren enthält das Toolkit, wie bereits erwähnt, den ZBL-Assembler *zblasm*, der *.zba-ZBL-Assembler-Programme assembliert und ein entsprechendes *.zbp-Bytecode-Programm erzeugt. Auch dieser Assembler ist eine Konsolenanwendung, deren Aufruf-Syntax wie folgt aussieht:

```
zblasm <Quelldatei> [Optionen]
```

Die Quelldatei kann jede ASCII-Textdatei sein, die validen ZBL-Assembler enthält. Wird die Dateinamenendung in der Kommandozeile nicht mit angegeben, wird sie vom Assembler automatisch ergänzt. Die Kommandozeilenoptionen des Assemblers ähneln denen des Compilers:

Kommandozeilenoption	Bedeutung
-h	Zeige eine Hilfe zur Programm-Benutzung
-o <Datei>	Wähle einen Ziel-Dateinamen aus

Tabelle 16.4:
Kommandozeilenoptionen des Assemblers

Zusätzlich enthält das Toolkit einen ZBL-Disassembler *zblasm*, der ZBL-Bytecode-Programme mit der Datei-Endung *.zbp in ZBL-Assembler-Quelldatei disassembliert. Der *zblasm*-Disassembler ist ebenfalls eine Konsolenanwendung:

```
zblasm <Programm> [Optionen]
```

Die Kommandozeilenoptionen des Disassemblers ähneln naturgemäß denen des Assemblers bzw. sind identisch mit den in Tabelle 16.4 aufgeführten.

16.5 ZBL/0-Bot-Design – Entwickeln von Bots mit der ZFX-Bot-Language

Die Scriptsprache ZBL/0 (Syntax)

Programm-Syntax von ZBL/0

ZBL/0 ist eine problemorientierte Hochsprache der dritten Generation, Das bedeutet diese Sprache ist nicht objektorientiert, sondern prozedural. Viele Konzepte von ZBL/0 sind denen der Programmiersprache *PASCAL* ähnlich. Dazu gehört unter anderem, dass jede Anweisung und jeder Befehl in ZBL/0 mit einem Semikolon abgeschlossen werden muss. Des Weiteren muss jedes Programm mit einem Punkt abgeschlossen werden. ZBL/0 enthält eine Anzahl von Funktionen zur Steuerung von Game-Bots, die (anders als bei Sprachen wie C) nicht in Bibliotheken ausgelagert, sondern in der Sprache selbst fest integriert sind. Zusätzlich zu diesen Standardfunktionen können natürlich auch beliebige weitere Funktionen vom Programmierer definiert werden. ZBL/0 ist dabei case-insensitive. Das bedeutet: Anders als bei der Programmiersprache C wird Groß-/Kleinschreibung vom ZBL/0-Compiler ignoriert, d.h. Variablen und Anweisungen können groß- oder kleingeschrieben werden, ohne dass ihre Anwendung davon beeinflusst wird.

Kommentare in ZBL/0- Programmen

Wie bei anderen Programmiersprachen auch können Programmquelltexte in ZBL/0 kommentiert werden. Kommentare sind eine der grundlegendsten und unverzichtbarsten Komponenten einer modernen Programmiersprache und dienen der leichten Wartbarkeit des Codes, denn durch sie ist der Programm-Sourcecode leichter zu lesen und leichter zu verstehen. In der Scriptsprache ZBL/0 sind Kommentare als so genannte Line-Comments (im Gegensatz zu Block-Comments, wie sie in der Programmiersprache C verwendet werden) implementiert. Kommentare beginnen mit dem Hash-Doppelkreuz-Symbol `#` und enden mit dem nächsten Newline-Character, d.h., sie sind nur bis zum Ende der aktuellen Zeile und nur für eine Zeile gültig. Sollen Kommentare mehrere Zeilen überspannen, so muss jede Zeile mit einem Hash-Zeichen beginnen.

Beispiel:

```
var a; # der Kommentar besagt, dass a eine Variable ist
# diese Zeile ist ebenfalls ein Kommentar
```

Programm-Aufbau

Programme in der Scriptsprache ZBL/0 selbst sind Funktionen ähnlich wie die `main()`-Funktion in C mit dem Unterschied, dass das Hauptprogramm in ZBL/0 eine unbenannte Funktion ist. Alle Funktionen in ZBL/0 sind nach folgendem Schema aufgebaut:

1. Konstantendefinitionen
2. Variablendeklarationen

3. Funktionsdefinitionen

4. Funktionskörper

Wie man hier sehen kann, kennt ZBL/0 nicht nur Konstanten und Variablen mit lokalem Gültigkeitsbereich, sondern auch lokale Funktionen (wie in der Programmiersprache *PASCAL*), d.h. Funktionen, die innerhalb anderer Funktionen definiert sind und dort ausschließlich lokal verwendet werden können. Das Schlüsselwort zur Definition von Konstanten ist `const`. Konstanten werden definiert durch einen Aufruf von `const`, gefolgt von einer Liste von Konstanten in der Form `<Bezeichner> = <Wert>`. Verschiedene Konstanten in der Liste werden durch Kommata getrennt, und die gesamte Liste wird mit einem Semikolon abgeschlossen.

*Globale
Konstanten*

Beispiel:

```
const TRUE=1, FALSE=0, PI=3.14159;
```

Jede Programmiersprache benötigt Variablen, da es sonst nicht möglich wäre, die (Zwischen-)Ergebnisse von Operationen zu speichern und weiterzuverwenden. ZBL/0 unterscheidet nicht zwischen verschiedenen Datentypen für Variablen. Alle Variablen sind vom selben Typ – ein numerischer Datentyp für Floating-Point-Werte, der aber ebenfalls mit Ganzzahlen arbeitet. Das Schlüsselwort zur Deklaration von Variablen ist `var`. Variablen werden deklariert durch einen Aufruf von `var`, gefolgt von einer Liste von Variablen in der Form `<Bezeichner>`. Sollen Variablen bei der Deklaration auch mit einem konstanten Wert (d.h. mit einer Zahl und keiner anderen Variable) initialisiert werden, so werden sie in der Form `<Bezeichner> = <Wert>` in die Liste eingetragen. Verschiedene Variablen in der Liste werden durch Kommata getrennt, und die gesamte Liste wird mit einem Semikolon abgeschlossen.

Globale Variablen

Beispiel:

```
var x,y=5,z;
```

Jede Funktion in ZBL/0 kann beliebig viele lokale Unterfunktionen, d.h. Funktionen mit lokalem Gültigkeitsbereich (Scope), beinhalten. Das Schlüsselwort zur Definition von Funktionen ist `function`. Funktionen in ZBL/0 nehmen keine Argumente über eine Parameterliste an. Sollen Parameter übergeben werden, so muss dies über Variablen geregelt werden. Aber Funktionen liefern immer einen Wert vom Typ `var` zurück. Sofern ein bestimmter Wert nicht explizit mit der `return`-Anweisung als Rückgabewert der Funktion bestimmt wird, wird automatisch der Wert 1 zurückgegeben. Funktionen werden definiert durch einen Aufruf von `function`, gefolgt von dem gewünschten Bezeichner für die Funktion, gefolgt von einem Semikolon, das den Funktionskopf abschließt.

Lokale Funktionen

Beispiel:

```
function func;
```

**Lokale Konstanten
und Variablen**

Dem Ganzen wiederum folgen die Definition von lokalen Konstanten der Funktion, die Deklaration von lokalen Variablen, die Definition von lokalen Funktionen und letztendlich der Funktionskörper. Dieser Funktionskörper ist normalerweise ein Block von Anweisungen. Diese Blöcke werden selbst wie Anweisungen behandelt, d.h., ein Block muss ganz genau wie alle anderen Anweisungen auch mit einem Semikolon abgeschlossen werden. Blöcke beginnen mit einer öffnenden geschweiften Klammer und enden mit einer schließenden geschweiften Klammer.

Beispiel:

```
{
    # dies ist ein ZBL/0-Block
    a = rnd 5;
};
```

Beispiel:

```
function  rand5;
var a;
{
    a = rnd 5;
    return a;
};
```

**Den Stack sauber
halten**

Um eine selbst definierte Funktion auszuführen, muss sie vom Programm genauso aufgerufen werden wie eine Standardfunktion (siehe Aufruf der rnd-Funktion im letzten Beispiel). Dabei müssen wir beachten, dass eine Funktion immer einen Wert zurückgibt, der auf dem Stack abgelegt wird. Wenn die zurückgegebenen Werte nicht vom Programm weiterverarbeitet werden, d.h., wenn sie nicht in Variablen gespeichert oder in Vergleichsoperationen benutzt werden, wächst der Stack, wenn viele Funktionen aufgerufen werden, was nach einiger Zeit zu einem Stack-Overflow führen kann. Aus diesem Grund ist es immer besser, den Rückgabewert von Funktionen in einer Variable aufzufangen.

**Bedingungen und
Schleifen**

Zusätzlich zu Funktionen bietet ZBL/0 die für Programmiersprachen üblichen Kontrollstrukturen wie Schleifen und Alternativen. Da die Ausführung von Programmen in einem Computer rein linear abläuft (d.h., alle Anweisungen werden der Reihe nach ausgeführt), sind derartige Kontrollstrukturen die einzige Möglichkeit, Befehlswiederholungen oder Abzweigungen im normalen Ablauf eines Programms herzustellen. Für Kontrollstrukturen in ZBL/0 gilt allgemein: Falls die Bedingung der Kontrollstruktur den Wert 0 annimmt, wird sie als FALSE und somit als nicht erfüllt interpretiert. Alle

anderen Werte werden als `TRUE` und somit als erfüllte Bedingung interpretiert. In ZBL/0 gibt es nur die kopfgesteuerte Schleife zur Programmierung von Iterationen. Kopfgesteuerte Schleifen werden in ZBL/0 durch eine **while-do**-Befehlskombination realisiert:

```
while <Bedingung> do <Aktion>
```

Beispiel:

```
while a <> 5 do
{
    a = rnd 5;
};
```

Bedingte Alternativen werden in ZBL/0 durch eine einfache Auswahl als **if-then**-Befehlskombination realisiert, und eine Alternativauswahl stellen wir durch die Befehlskombination **if-then-else** bereit:

```
if <Bedingung> then <Aktion>
```

oder:

```
if <Bedingung> then <Aktion1> else <Aktion2>
```

Beispiele:

```
if a <> 5 then
{
    a = 5;
};

if a = 1 then
{
    a = 0;
};
else
{
    a = 1;
};
```

Für mathematische Ausdrücke stellt ZBL/0 eine Auswahl von verschiedenen Operatoren zur Verfügung. Der wahrscheinlich wichtigste Operator ist dabei der Zuweisungsoperator `=`. Gleich danach kommen die arithmetischen Operatoren für die einfachen mathematischen Operationen Division `/`, Multiplikation `*`, Addition `+` und Subtraktion `-`. Für logische Operationen gibt es in ZBL/0 die Negation `!`, das logische UND `&` und das logische ODER `|`. Für relationale Operationen (Vergleichsoperationen) benutzt ZBL/0 ähnliche Symbole wie die Programmiersprache *PASCAL*, d.h. `=` für Gleichheit und `<>` für

*Mathematische
Ausdrücke in
ZBL/0*

Ungleichheit. Die übrigen Vergleichsoperatoren für größer und kleiner sind mit den entsprechenden relationalen Operatoren der Programmiersprache C identisch.

Tabelle 16.5:
Operator-Präze-
denz in ZBL/0

Priorität	Operator
1	! logische Negation
2	/ arithmetische Division
	* arithmetische Multiplikation
3	+ arithmetische Addition
	- arithmetische Subtraktion
4	= Vergleich auf Gleichheit
	<> Vergleich auf Ungleichheit
	< Kleiner-als-Vergleich
	<= Kleiner-oder-gleich-Vergleich
	> Größer-als-Vergleich
	>= Größer-oder-gleich-Vergleich
5	& logisches UND
	logisches ODER
6	= Wertzuweisung

EZBL/0-Standardbefehle und Anweisungen

Intrinsische Stan-
dardfunktionen

Die Scriptsprache ZBL/0 ist problemorientiert – ihr Sinn und Zweck ist es, Scripts zu entwickeln, die das Verhalten von so genannten Game-Bots in Computerspielen steuern. Dies geschieht mit Hilfe von bisher 45 Standardfunktionen, die die Funktionalität umfassen, die von einem Game-Bot erwartet wird. Die Standardfunktionen von ZBL/0 sind *intrinsisch*, d.h., sie sind nicht in einer externen Bibliothek, sondern direkt in der Definition der Scriptsprache integriert. Im Folgenden nehmen wir die wichtigsten Standardfunktionen von ZBL/0, die auch im Spiel *Pandoras Legacy* verwendet werden, genauer unter die Lupe. Eine ausführlichere Auflistung und Beschreibung aller Standardfunktionen von ZBL/0 findet ihr im Anhang dieses Buches. Die ZBL/0-Standardfunktionen lassen sich nach verschiedenen Aufgaben gliedern:

- ➡ Management-Funktionen
- ➡ Sensorfunktionen
- ➡ Kontrollfunktionen

Die Management-Funktionen von ZBL/0 regulieren die Initialisierung von Game-Bots und umfassen Aktionen, die die Existenz des Bots in der virtuellen Spielwelt direkt beeinflussen. Dies beinhaltet die Abfrage von World-State-Informationen, die nicht direkt als Sensorinformationen gelten, aber dennoch das Verhalten des Game-Bots verändern können. Dazu zählen die folgenden Funktionen:

Management-Funktionen

- ➔ **spawn:**
Die erste Funktion, die in jedem ZBL/0-Programm ausgeführt werden muss, ist die `spawn`-Management-Funktion, die dem Game-Bot »Leben einhaucht«. Durch Aufruf der Funktion `spawn` wird ein Bot initialisiert und im Spiel aktiviert.
- ➔ **respawn:**
Die Management-Funktion `respawn` führt ein Reset des Game-Bots durch und versetzt den Game-Bot in seinen ursprünglichen Zustand, den er beim Aufruf der `spawn`-Funktion hatte.
- ➔ **danger:**
Die `danger`-Funktion gibt dem Bot eine Art sechsten Sinn für Gefahr und drückt damit einen inneren Zustand des Game-Bots aus. Wenn ein Game-Bot sich in Gefahr befindet bzw. »das Gefühl hat«, in Gefahr zu sein, so gibt diese Management-Funktion den Wert 1 zurück, wenn nicht, gibt sie den Wert 0 zurück.

Die ZBL/0-Sensorfunktionen stellen für ZBL/0-Bot-Programme die Daten zur Verfügung, die die Sinnesorgane der Game-Bots über sie selbst und ihr Umfeld ermitteln. Die dadurch gewonnenen Informationen erlauben es den Game-Bots, sich in der virtuellen Welt des Spiels zurechtzufinden.

Sensorfunktionen

- ➔ **alive:**
Die Sensorfunktion `alive` liefert dem ZBL/0-Script den Wert 1 zurück, wenn der Game-Bot noch am Leben ist. Wenn der Game-Bot sein Leben verloren hat, wird der Wert 0 zurückgegeben. Analog gibt die Funktion `target_alive` Aufschluss darüber, ob das aktuelle Ziel des Game-Bots, d.h. der nächste gegnerische Körper, noch am Leben ist. In den meisten ZBL/0-Scripts wird die `alive`-Funktion als Bedingung für die Hauptschleife des Bot-Programms genutzt.
- ➔ **health:**
Die Sensorfunktion `health` gibt den Gesundheitszustand des Game-Bots, d.h. seine Lebensenergie zurück. Dies kann nützlich sein, wenn ein Programm einen Game-Bot mit zu geringer Lebensenergie in einen defensiven Zustand versetzen soll.

➔ `blocked+<modifier>`:

Die Funktion `blocked` wird immer zusammen mit einem der richtungs-angebenden Funktionsmodifikatoren *left* (links), *right* (rechts), *front* (vorne) und *back* (hinten) benutzt. Diese Sensorfunktion gibt den Wert 1 zurück, wenn der Weg des Game-Bots in der angegebenen Richtung blockiert ist, oder 0, wenn der Weg frei ist. Wenn beispielsweise die Funktionskombination *blocked left* aufgerufen würde, so würde sie den Wert 1 zurückgeben, wenn der Weg nach links blockiert wäre, aber den Wert 0, falls der Weg frei wäre.

➔ `target_ahead`:

Die Sensorfunktion `target_ahead` liefert den Wert 1 zurück, wenn sich ein Ziel in der Schusslinie des Game-Bots befindet, bzw. den Wert 0, wenn kein Gegner für den Bot sichtbar ist.

➔ `obstacle`:

Die Sensorfunktion `obstacle` gibt den Wert 1 zurück, wenn ein (womöglich bewegliches) Hindernis, z.B. eine geschlossene Tür (aber keine Wand), den Weg des Game-Bots blockiert.

Kontrollfunktionen

Die dritte Gruppe von ZBL/0-Standardfunktionen sind die Kontrollfunktionen, die die Befehle zur Steuerung der Game-Bots umfassen. Diese Kontrollfunktionen erlauben es den Game-Bots, aktiv mit ihrer Umgebung in der virtuellen Spielwelt zu interagieren.

➔ `step`

Jeder Aufruf der Kontrollfunktion `step` bewegt den Game-Bot um einen Schritt vorwärts.

➔ `backstep`

Analog zur Kontrollfunktion `step` bewegt die Funktion `backstep` den Game-Bot um einen Schritt rückwärts.

➔ `turn+<angle>`

Die Kontrollfunktion `turn` wird dazu benutzt, um einen Game-Bot zu drehen, wobei diese Funktion einen einzelnen Parameter akzeptiert, der den Drehwinkel enthält. Ist der Drehwinkel positiv, so dreht sich der Bot nach links. Ist der Drehwinkel negativ, dann dreht sich der Game-Bot nach rechts. Zusätzlich zur Funktion `turn` gibt es noch die zwei Kontrollfunktionen `turn_left` und `turn_right`, die den Game-Bot jeweils nach links bzw. rechts drehen. Der Aufruf von `turn_left` würde z.B. ebenso wie `turn 90` dafür sorgen, dass sich der Game-Bot um 90 Grad nach links dreht.

➔ `face+<modifier>`

Die Kontrollfunktion `face` wird normalerweise zusammen mit dem Funktionsmodifikator `target` aufgerufen. Die Befehlskombination `face target` führt dann dazu, dass der Game-Bot sich dem Gegner zuwendet, der ihm am nächsten ist. Das heißt, der Bot dreht sich, bis er sein Ziel erfasst hat.

➔ `fire`

Die Kontrollfunktion `fire` sorgt dafür, dass der Game-Bot einen einzelnen Schuss mit seiner Waffe abgibt.

➔ `use<object>`

Die Funktion `use` gibt dem Game-Bot den Befehl, ein bestimmtes Objekt, dessen Objekt-ID der Kontrollfunktion als Parameter übergeben wurde, zu benutzen bzw. zu aktivieren. Angenommen, der Bot stände vor einer geschlossenen Tür und die Objekt-ID für Türen sei 0, so würde die Anweisung `use 0` den Game-Bot dazu veranlassen, die Tür zu öffnen.

➔ `rnd<limit>`

Diese Funktion gehört zu keiner der drei genannten Funktionskategorien von ZBL/0. Sie erlaubt es Game-Bots, Werte von einem Zufallszahlengenerator zu verwenden. Die Funktion `rnd` akzeptiert einen einzelnen Parameter und liefert einen zufälligen Wert zwischen 0.0 (untere Grenze) und dem Wert des Parameters (obere Grenze) zurück.

Die Entwicklung eines Game-Bots mit ZBL/0

Mit Hilfe der Scriptsprache ZBL/0 ist es möglich, Programme zu schreiben, die das Verhalten eines Game-Bots in einem Spiel bestimmen. Die Definition von künstlichem Verhalten (engl. artificial behaviour), d.h. dem Verhalten einer künstlich intelligenten Entität, spielt in den Wissenschaften, die sich mit künstlicher Intelligenz und künstlichem Leben (engl. artificial life, ALife) befassen, eine wichtige Rolle. Es ist das Verhalten eines Game-Bots, an dem seine Intelligenz gemessen wird. In der Biologie gibt es die Fachrichtung der Ethologie (Verhaltenslehre), die sich mit der Analyse und dem Verständnis des Verhaltens von Lebewesen beschäftigt.

Ethologie

Ethologie, oder Verhaltenslehre, ist die Wissenschaft, die sich mit der Erforschung und der Analyse von tierischem Verhalten beschäftigt. Einer der Gründer dieser Lehre war Konrad Lorenz, der anhand seiner Beobachtungen von Graugänsen eine Vielzahl von ethologischen Grundbegriffen geprägt hat.

(Czihak, G. et al., Biologie, Kapitel 7, Springer-Verlag)

In der Ethologie gibt es eine Vielzahl von Modellen, die dabei helfen, Verhaltensweisen und ihre Ursachen zu erkennen und zu klassifizieren. Die damit gewonnenen Erkenntnisse kommen auch in der Forschung zu künstlichem Leben und künstlichem Verhalten zur Anwendung. Dabei sind ethologische Modelle für reaktive Verhaltensweisen (Verhalten, das sich als Reaktion auf einen Reiz in der Umgebung ergibt) für die Bot-Entwicklung am interessantesten. Hierbei wird sichtbar, dass die Verhaltensweisen direkt von der Wahrnehmung einer Entität abhängen. Wenn es um die Definition

*Verhaltensweisen
basieren auf
Wahrnehmung*

des Verhaltens künstlich intelligenter Agenten (und somit auch von Game-Bots) geht, so benutzt man häufig die Faustformel:

$$\text{Verhalten} = \text{Kontrolle} + \text{Wissen}.^6$$

Um also das Verhalten eines Game-Bots zu definieren, muss man das Wissen des Game-Bots mit den ihm zur Verfügung stehenden Kontrollen kombinieren. Die Kontrollen sind die ZBL/0-Funktionen, die Aktionen des Bots ausführen. Game-Bots in ZBL/0 sind rein reaktiv, d.h. ihre Aktionen werden als Reaktion auf einen Reiz in ihrer Umgebung ausgeführt. Diese Art des Verhaltens, die dem instinktiven, angeborenen Verhalten der meisten Lebewesen gleicht, führt zur folgenden Definition für das Wissen von ZBL/0-Game-Bots:

$$\text{Wissen} = \text{Instinkt} + \text{Wahrnehmung}$$

Instinkt Der Instinkt, d.h. programmierte Regeln, die die Reaktionen des Bots auf bestimmte Umgebungsreize festlegen, bilden die Wissensbasis des Game-Bots. Dieses im Bot-Programm einprogrammierte Regelwissen führt nur in Kombination mit der Wahrnehmung des Game-Bots zu verwendbaren Daten. Die Wahrnehmung, die die Reize aus der Umgebung für das instinktive Regelwissen des Bots bereitstellt, entsteht durch Abfragen der Sensorinformationen des Game-Bots. Das Wissen des Game-Bots über sein Umfeld in der virtuellen Spielwelt entsteht also, wenn Sensorinformationen mit den Regeln, die im Bot-Programm festgelegt sind, verrechnet werden.

Um einen einigermaßen intelligent erscheinenden Game-Bot, d.h. einen glaubhaften NPC, zu erschaffen, muss man also bestimmte Regeln für das Verhalten des NPCs aufstellen und diese in Form eines Bot-Programms formulieren, so dass der NPC seine Umgebung wahrnehmen und mit ihr interagieren kann. Nicht alle NPCs müssen dem Spieler dabei feindlich gesinnt sein. Oft findet man in FPS-Spielen auch neutrale NPCs, die als Statisten im Hintergrund umherwandern und folglich nicht direkt am Spielgeschehen teilnehmen. Da solche NPCs normalerweise nicht an Kämpfen teilnehmen und damit auch kein besonderes taktisches Geschick benötigen, sind sie relativ leicht zu programmieren, da sie natürlich keine allzu komplexen Aktionen ausführen müssen. Für einen solchen neutralen NPC, der z.B. einen Korridor auf- und abgeht, würden wir in ZBL/0 das Script in Listing 16.1 verwenden.



```
{
  spawn;
  while alive do
  {
```

6 Funge, J.: *AI for Games and Animation*, Kapitel 1, AK Peters

Listing 16.1:

Ein simples ZBL/0-Programm

```

if blocked front then
{
    # Drehung um 180 Grad mit Zwischenschritt
    turn_left;
    step;
    turn_left;
};
else
    step;
};
}.

```

Dies ist ein kurzes, einfaches Programm. Wie man hier aber schon sehen kann, würde ein komplexeres Programm schnell unübersichtlich werden. Aus diesem Grund macht es Sinn, das Programm nach dem Prinzip »Teile und herrsche« in verschiedene Funktionen zu zerlegen. In diesem Fall bietet es sich an, die Drehung des Bots in einer Funktion `turn180` unterzubringen und die Aktionen in der Hauptschleife ebenfalls in eine separate Funktion `patrol` zu schreiben, deren Bezeichner selbsterklärend die Aufgabe der Funktion beschreibt. Das modifizierte Programm mit den zwei neuen Funktionen würde dann so wie in Listing 16.2 aussehen:

Divide and conquer

```

function turn180;
{
    # Drehung um 180 Grad mit Zwischenschritt
    turn_left;
    step;
    turn_left;
};

function patrol;
{
    if blocked front then # wenn vorne blockiert ist
        turn180;           # dann umkehren
    else                  # sonst
        step;              # vorwärts
};

{
    spawn;
    while alive do
        patrol;
}.

```

Dieser Game-Bot wäre in der Lage, in einem Bereich des Levels zu patrouillieren. Sobald er auf diesem Patrouillengang an ein Hindernis stößt, dreht er sich links um und läuft den Weg, den er gekommen ist, zurück. Für einen Game-Bot, der einfach auf- und ablaufen soll, wäre solch ein Verhalten akzeptabel, nicht jedoch für einen Game-Bot, der Wache halten soll. In die-

**Listing 16.2:**

Diese Programmversion zeigt, wie man in ZBL/0 Funktionen definiert und aufruft.

sem Fall sollte ein Game-Bot immer seinen Rücken bedeckt halten und ihn bei Drehung der Wand zuwenden, um nicht durch einen von hinten kommenden Feind überrascht zu werden. Eine verbesserte Version der Funktion `turn180`, die dieses Problem berücksichtigt, muss dafür sorgen, dass ein gescripteter NPC sich nur dann in eine bestimmte Richtung dreht, wenn sich in dieser Richtung kein Hindernis befindet. Dies wird in der folgenden Version der `turn180`-Funktion gelöst:



```
function turn180;
{
  if !blocked left then # wenn links frei
  {                      # dann links drehen
    turn_left;
    step;
    turn_left;
  };
  else                  # sonst
  {                      # rechts drehen
    turn_right;
    step;
    turn_right;
  };
};
```

Reinkarnation

Damit hätten wir einen neutralen Game-Bot, der die Spielwelt um sich herum weitgehend ignoriert und unbeirrt derselben Tätigkeit (Patrouillieren in einem bestimmten Bereich des Levels) nachgeht, solange er am Leben ist. Aber was würde nun passieren, wenn dieser neutrale NPC ins Kreuzfeuer zwischen dem Spieler und anderen, feindlichen NPCs geraten würde? In dem Augenblick, in dem die Lebensenergie dieses Bots auf den Wert 0 herabsinkt, würde der Bot sein Leben verlieren, und bei der nächsten Abfrage der `alive`-Funktion im Kopf der `while alive`-Schleife, würde die Schleife verlassen werden. Damit wäre die letzte Instruktion des Programms ausgeführt worden und das Bot-Programm wäre beendet, mit dem Resultat, dass der entsprechende Bot-Prozess in der ZBL/0-Virtual Machine deaktiviert würde. Was aber, wenn es unerwünscht wäre, neutrale NPCs komplett aus der Spielwelt zu entfernen? In diesem Fall ist es nützlich, nach der `while alive`-Schleife im Hauptprogramm die ZBL/0-Standardfunktion `respawn` aufzurufen. Damit würde ein »toter« NPC sofort wieder auferstehen.



```
{
  spawn;
  while alive do
    patrol;
    respawn;
  }.
```

Der Game-Bot, den wir damit definiert haben, wird, solange er am Leben ist, unaufhörlich dieselbe Strecke auf- und ablaufen. Sein Patrouillengang würde erst durch den Verlust seines Lebens beendet, aber augenblicklich von neuem gestartet, da der Game-Bot mit Hilfe der `respawn`-Funktion ein neues Leben gewinnen würde. Dieser Game-Bot wäre aber immer noch darauf beschränkt, im selben Bereich des Levels zu verweilen. Angenommen, ein NPC sollte nun stattdessen einfach durch den Level laufen. Wenn dies der Fall ist, so sollte man für den Game-Bot eine Strategie wählen, mit der er im Level erfolgreich navigieren kann. Eine einfache, aber effektive Methode für die Lösung dieses Problems ist es, den Game-Bot dazu zu bringen, den Level-Wänden zu folgen, d.h., sich an den Wänden des Levels »entlangzutasten«:

*Pathfinding auf
primitiv*

```
function travel;
{
  if !blocked right then  # wenn rechts frei
  {
    turn_right;           # dreh dich nach rechts
    step;                 # und gehe vorwärts
  };
  else                    # sonst
  {
    if !blocked front then # wenn vorne frei
    {
      step;               # gehe vor
    };
    else                  # sonst
    {
      turn_left;          # dreh dich nach links
    };
  };
};

{
  spawn;
  while alive do
    travel;
    respawn;
};
```



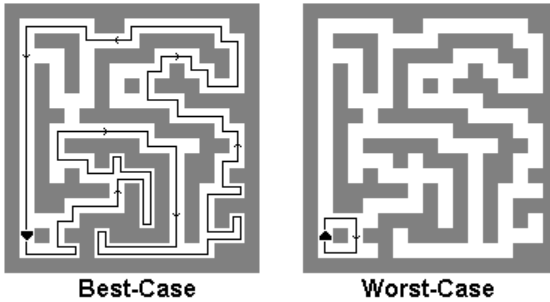
Der Game-Bot in diesem Programm folgt der Strategie, der von ihm rechts gelegenen Wand zu folgen. Diese sehr einfach zu implementierende Strategie hat aber auch Nachteile.

Im besten Fall würde der Bot den Level zu einem großen Teil durchlaufen und eventuell wieder an seiner ursprünglichen Position ankommen. Im schlechtesten Fall jedoch würde der Bot nur einen Bruchteil des Levels durchlaufen und sich immerzu im Kreis bewegen. Wenn anstatt der Strategie, der rechten Wand zu folgen, der linken Wand gefolgt würde, so wäre der Effekt derselbe. Eine bessere Möglichkeit wäre ein Hybrid-Algorithmus, bei dem für einen

*Verbessertes
Wand-Hangeln*

Abbildung 16.5:
Gegenüberstellung
der bestmöglichen
und der schlimmsten
Situationen der
Strategie »Folge der
rechten Wand« für
einen Bot

Strategie: folge der rechten Wand



bestimmten Zeitraum der rechten Wand gefolgt wird, dann aber die Seite gewechselt und für eine bestimmte Zeit der linken Wand entlang gelaufen wird. Dies erfordert eine Modifizierung der Funktion `travel` und die Deklaration von zwei globalen Variablen: zum einen von `direction`, die die aktuell gewählte Richtung enthält, und zum anderen von einer Zählervariable `i`:



```
var direction=1,          # 1= rechts, 0=links
    i=0;                  # Zählervariable
function travel;
{
  if i < 8 then            # wenn weniger als 8 Schritte
  {
    if direction = 1 then  # wenn rechts gewählt, dann
    {                     # an rechter Wand entlang
      if !blocked right then
      {
        turn_right;
        step;
      };
    }
    else
    {
      if !blocked front then
        step;
      else
        turn_left;
    };
  };
  else                    # sonst
  {                       # an linker Wand entlang
    if !blocked left then
    {
      turn_left;
      step;
    };
    else
```

```

{
    if !blocked front then
        step;
    else
        turn_right;
};
};
i=i+1;
};
else                                # nach 8 Schritten
{                                    # Richtungswechsel
    i=0;
    if direction = 1 then
        direction=0;
    else direction=1;
};
};

```

Wenn die Variable `direction` 1 ist, wird an der rechten Wand entlanggelaufen, wenn `direction` den Wert 0 enthält, wird an der linken Wand entlanggelaufen. Alle acht Schritte wird die Richtung geändert. Die Chance, dass hierbei immer wieder dieselbe Route beim Durchlauf des Levels gewählt wird, ist bei diesem Game-Bot schon wesentlich geringer. Dieses Verhalten lässt sich aber noch etwas verbessern, indem man die Wahl der Richtung dem Zufall überlässt bzw. einen zufälligen Faktor mit ins Spiel bringt. Dafür fügen wir eine weitere Funktion zu dem ZBL/0-Bot-Programm hinzu, die zufällig die Richtung wählt, an der sich der Game-Bot orientiert:

Probabilistik

```

function choosedir;
const rc=16,min=4,max=12;
var r;
{
    r = rnd rc;                # r=Zufallswert<=16(rc)
    if r > min & r < max then  # wenn min < r < max
        return 1;             # wähle rechts
    else                       # sonst
        return 0;             # wähle links
};

```



Diese Funktion muss natürlich vom Programm auch benutzt werden, und dies geschieht im letzten `else`-Zweig der `travel`-Funktion, der entsprechend verändert werden muss:

```

else                                # nach 8 Schritten
{                                    # Richtungswechsel
    i=0;
    direction=choosedir;
};

```



**Selbst-
verteidigung**

Dieser Game-Bot ist nun in der Lage, sich allein zurechtzufinden und durch einen Level zu laufen. Aber zurück zu dem ersten Bot-Programm mit dem Game-Bot, der Wache schiebt und in einem kleinen Bereich des Levels patrouilliert. Da der Bot immer noch nicht kämpfen kann, wäre er als Wächter ungeeignet, da er jedem Gegner schutzlos ausgeliefert wäre. Um dies zu ändern, muss das Programm modifiziert werden. Auf der Ebene (in der Hierarchie der Funktionen) des Hauptprogramms des Bots geschieht dies durch Integration einer Abfrage, die den Game-Bot nur dann patrouillieren lässt, wenn er nicht in Gefahr ist. Bei Gefahr wechselt er in einen Gefechtsmodus. Die entsprechend veränderte Version des Hauptprogramms sieht dann folgendermaßen aus:



```
# Hauptprogramm des Kampf-Bots
{
    spawn;
    while alive do
    {
        if !danger then      # wenn keine Gefahr
            patrol;          # dann patrouilliere
        else                 # sonst
            skirmish;        # auf in den Kampf!
    };
    respawn;
}.
```

Wenn keine Gefahr herrscht (d.h., wenn kein Gegner in der Nähe ist), dann wird patrouilliert (d.h., die Funktion `patrol` aufgerufen), ansonsten wird die Funktion `skirmish` aufgerufen, die einen einfachen Algorithmus für Kampfhandlungen ausführt:



```
function skirmish;
{
    # function skirmish - Ziel anvisieren/angreifen
    if target_ahead then    # wenn Ziel im Visier
    {
        fire;               # Salve 1
        fire;               # Salve 2
        fire;               # Salve 3
        step;               # vorrücken
    };
    else                    # ansonsten
    {
        if !blocked front then # wenn Weg frei
        {
            face target;      # Ziel anvisieren
            if target_alive then # wenn Ziel noch lebt
            {
                fire;         # Schuss 1
            }
        }
    }
}
```



```

        step;           # vorrücken
        fire;           # Schuss 2
    };
    else                # sonst
        step;           # nachrücken
    };
    else                # wenn Schusslinie versperrt
        backstep;       # zurückweichen
};
};

```

Damit wäre dies ein voll einsatzfähiger Game-Bot, der Gegner, die ihm zu nahe kommen, angreifen würde. Das Bot-Programm könnte natürlich auch die Funktion `travel` statt der Funktion `patrol` verwenden, wenn es kein patrouillierender, sondern ein frei im Level umherlaufender Game-Bot sein soll. Wenn einer dieser Bots jedoch auf seinem Weg zu einer verschlossenen Tür kommen würde, so würde er umkehren und niemals die Tür öffnen und durch sie hindurch gehen. Dem lässt sich aber leicht vorbeugen, indem man die Funktion `check_door` verwendet, die Türen öffnet und es dem Bot ermöglicht dann einen freien Weg vor sich zu haben.

*Sesam,
öffne dich!*

```

function check_door;
{
    if obstacle then    # ist Weg durch Tür blockiert
    {
        use 0;          # öffne die Tür
        step;           # gehe durch die Tür
        return 1;       # gib TRUE zurück
    };
    else                # sonst
        return 0;       # gib FALSE zurück
};

```



Um die Funktion `check_door` effektiv in ein Bot-Programm zu integrieren, muss sie dann aufgerufen werden, wenn der Weg des Bots (nach vorne) zwar blockiert ist, aber noch nicht feststeht, ob dies durch eine Wand oder durch eine Tür geschieht. In der Funktion `patrol` z.B. würde dies eine Umstrukturierung der Funktion erfordern, die dann so aussehen würde:

```

function patrol;
{
    if !blocked front then # wenn vorne frei ist
        step;              # vorwärts
    else                    # sonst
    {
        if check_door then # wenn es eine Tür ist
            step;          # vorwärts
    }
}

```



```

        else                # sonst
            turn180;         # umkehren
    };
};

```

Letzte Worte

Dies war nur eine kurze Einführung in die Entwicklung von Game-Bots, die dieses komplexe Themengebiet nur angerissen hat. Der Befehlsumfang von ZBL/0 erlaubt die Programmierung wesentlich umfangreicherer und auch intelligenterer Bots als denen in den hier gezeigten Beispielen, die für den Einsatz im Spielprojekt *Pandoras Legacy* entwickelt wurden. Die Implementierung der Funktionen von ZBL/0 in *Pandoras Legacy* ist so einfach wie möglich gehalten und lässt eine Menge Raum für eigene Erweiterungen, die das Verhalten der Game-Bots verbessern können. Das ZBL/0 SDK, das die Entwicklungstools für ZBL/0 und die ZBL/0-Virtual Machine beinhaltet, enthält weitere Beispielskripts und eine Anzahl von Anwendungen, die demonstrieren, wie Game-Bots implementiert werden können.

16.6 Ein Blick zurück, zwei Schritt nach vorn

Mach dir den Bot

Die Demo-Applikation zu diesem Kapitel, eine um Bots erweiterte Version von *Pandoras Legacy*, hat nur minimale Erweiterungen. Dabei benötigt man natürlich die Klasse `CGameBot`, die zum einen von der uns bereits bekannten Klasse `CGameCharacter` abgeleitet ist. Zum anderen ist diese Klasse aber auch von der hier vorgestellten abstrakten Bot-Klasse `zblbot` von ZBL/0 abgeleitet, damit sie über Scripts gesteuert werden kann. Die Implementierung der Klasse ist ganz analog zu der Klasse `CGamePlayer`. Jedoch kommen hier noch die entsprechenden Methoden aus dem Interface von `zblbot` hinzu. Ihr solltet jedoch keine Schwierigkeiten haben, die Implementierungen nachzuvollziehen, denn sie sind nicht besonders komplex.

**Änderungen an
CGame**

Weitere Änderungen fanden in der Klasse `CGame` statt. Hier muss man sämtliche Funktionalitäten, die man auch für die Player-Liste brauchte, für die Bot-Liste ebenfalls implementieren. Der Server muss die Liste der Bots dann und wann zu einem Update über das Netzwerk animieren, ein neuer Client muss zuerst überhaupt eine Liste von Bots erhalten usw. Aber auch hier sind die Methodennamen jeweils so gewählt, dass man anhand der Namen schon gut sehen kann, was die einzelnen Methoden tun. Nun solltet Ihr also die CD-ROM ins Laufwerk legen, den entsprechenden Projektbereich starten und eine Runde gegen die Bots zocken. Danach empfehle ich aber auch, den Code gründlich zu untersuchen und insbesondere in `CGame` die Änderungen für die Bots zu verfolgen.

Und weiter geht's

Dies war zwar das letzte Kapitel dieses Buches, aber gleichzeitig bietet es euch auch das meiste Potenzial für Erweiterungen. Ihr solltet als Nächstes die Beispielskripts der Bots für ZBL/0 nehmen und mit diesen herumspielen. Die Bots in *Pandoras Legacy* sind nicht sonderlich schlau. Es liegt nun bei

euch, dies im Rahmen der Möglichkeiten und darüber hinaus zu ändern. Nachdem ihr nun ein gutes Gefühl für das Game-Scripting entwickelt habt, empfehle ich euch, euch im Bereich Scripting fortzubilden. Ob ihr nun *LUA*, *Python* oder *ZBL/0* verwendet, ist dabei nicht so entscheidend. Viel wichtiger ist dabei das Potenzial, das ihr euren Games durch Scripting hinzufügt. Und denkt immer daran: Man kann nicht nur Bots scripten. In großen kommerziellen Spielen sind für gewöhnlich auch die Events per Script implementiert, die man durch das Betätigen eines Schalters oder einer Tür auslöst. Viel Spaß.

Teil 5 Anhang

Anhang A:	Die Funktionen von ZBL/0	915
Anhang B:	CD-ROM und begleitendes Material zu diesem Buch	925
Anhang C:	Internetseiten rund um die Spieleentwicklung	927
Anhang D:	Epilog	929

A Die Funktionen von ZBL/0

Die folgenden Abschnitte beschreiben die ZBL/0 Standardfunktionen, jeweils mit kurzer Erklärung ihrer Funktionalität und Hinweisen zur Implementierung der Funktionen in der Host-Anwendung. Zu jeder der ZBL/0 Funktionen ist der Prototyp der entsprechenden Funktion der ZBL-API gegeben.

A.1 House-Keeping Funktionen zur generellen Botsteuerung

Die House-Keeping Funktionen umfassen Aktionen die die Existenz des Bots in der virtuellen Welt der Host-Anwendung direkt regeln und World-State Informationen, die nicht direkt als Sensorinformationen gelten, die direkten Einfluss auf die Wahrnehmung und das Wissen des Bots über seine Umgebung in der virtuellen Spielwelt haben.

ZBL/0: danger

ZBL-API: `int zb_checkDanger(void);`

Diese Funktion gibt den Wert 1 zurück, wenn sich der Bot in Gefahr befindet (z.B. wenn sich ein Gegner in der Nähe aufhält). Dies lässt sich auf verschiedene Arten implementieren: entweder über ein internes, regelmäßig geprüftes Attribut des Bots das abgefragt wird, oder durch eine explizite Analyse der aktuellen Situation, in der sich ein Bot befindet. Falls diese Funktion nicht benötigt wird, sollte die Implementierung des Bots als Default-Wert den Wert 1 zurückgeben.

ZBL/0: die

ZBL-API: `void zb_doDie(void);`

Diese Funktion beendet die Ausführung des Bots. Aktionen in der Host-Anwendung die vor Beendung des Botprozesses ausgeführt werden sollen, müssen in der entsprechenden ZBL-API Funktion implementiert werden.

ZBL/0: `find <object>`

ZBL-API: `void zb_doFind(int);`

Die Funktion `find` dient der Suche, bzw. Auffindung des im Parameter `<object>` gegebenen (vorher gemerkten) Objekts (siehe dazu auch Funktion `memorize`) – unabhängig von der relativen Position des Objekts zum Bot. Die `find` Funktion plant dazu einen Weg/Pfad zum Objekt, dem der Bot dann über verschiedene Aufrufe der Funktion folgt. Dabei ist zu beachten, dass die Funktion `find` den Pfad schrittweise über mehrere Aufrufe und nicht während eines Aufrufs abläuft. Eine gute Methode für die Implementierung dieser Funktion wäre die Umsetzung des A* Algorithmus zur Wegfindung. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` als leer definiert werden.

ZBL/0: `idle`

ZBL-API: `int zb_checkIdle(void);`

Die `idle` Funktion gibt den Wert 1 zurück, wenn sich der Bot nicht in Gefahr befindet (siehe auch Funktion `danger`) oder wenn der Bot gerade keinem Pfad folgt (siehe Funktion `find`). In allen anderen Fällen gibt die Funktion den Wert 0 zurück. Implementieren lässt sich das ähnlich wie die `danger` Funktion. Falls diese Funktion nicht benötigt wird, sollte die Implementierung des Bots als Default-Wert den Wert 0 zurückgeben.

ZBL/0: `initialize <xpos> , <ypos> [ , <zpos>]`

ZBL-API: `void zb_doInitialize(double,double,double);`

Diese Funktion initialisiert den Bot und positioniert ihn an den als Parameter an die Funktion übergebenen Koordinaten der virtuellen Welt (implementierungsabhängig könnte dies auch an zufälligen Koordinaten geschehen). Zwei der Koordinaten müssen mindestens gegeben werden (für Positionierung in 2 Dimensionen), die dritte Koordinate (z-Wert) ist optional, sollte bei der Implementierung aber auf 0 initialisiert werden. Die Positionierung muss entsprechend der Host-Anwendung in der Host-Anwendung implementiert werden. Die Funktion `initialize` sollte die erste Funktion sein, die in einem Bot ausgeführt wird.

ZBL/0: `memorize <object>`

ZBL-API: `void zb_doMemorize(int);`

Die Funktion `memorize` »merkt« sich die Position des im Parameter `<object>` gegebenen Objektes. Bei der Implementation des Bots könnte dieses über eine im Bot integrierte Liste der bekannten/gemerkten Objekte mit ihren Positionen im Level gelöst werden. Die `memorize` Funktion ist insbesondere für Kombinationen mit der `find` Funktion ausgelegt, d.h. `find` sollte es ermöglichen, mit `memorize` »gemerkten« Objekte und Orte auszuwählen und zu »finden« (siehe auch Funktion `find`).

ZBL/0: respawn

ZBL-API: void zb_doRespawn(void);

Die `respawn` Funktion führt ein Reset des Bot-Programms aus und startet die Ausführung des Programms vom Anfang an. Die Implementierung der `respawn` Funktion kann auch dafür genutzt werden, den Bot an einem Spawn-Point im Level zu repositionieren. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` als leer definiert werden.

ZBL/0: spawn

ZBL-API: void zb_doSpawn(void);

Die Funktion `spawn` stellt eine Alternative zur Funktion `initialize` dar und kann anstelle von `initialize` als erste Funktion zur Initialisierung (dazu gehört auch die Positionierung des Bots im Level) eines Game-Bots verwendet werden. Anders als bei der `initialize` Funktion wird die Wahl der Startposition des Bots komplett der Host-Anwendung überlassen, könnte also auch mit Zufallswerten implementiert werden.

ZBL/0: spawned

ZBL-API: int zb_checkSpawned(void);

Diese Funktion liefert die Anzahl der Spawns und Respawns des Bots zurück. Die Funktion braucht nicht gesondert implementiert werden und sollte daher generell nur ein eine Rückgabe des Wertes 1 durchführen.

A.2 Modifikatoren zur Problemspezifikation

Einige der Funktionen in ZBL/0 benötigen zusätzliche Informationen zu ihren Aufgaben, da die Ausführung je nach Situation anders ablaufen kann, bzw. soll. Zu diesem Zweck gibt es in ZBL/0 eine Reihe von sogenannten Modifikatoren – Funktionen, die in Kombination mit anderen Funktionen eine modifizierte Funktionalität herbeiführen.

ZBL/0: back

ZBL-API: int zb_mdfBack(void);

Die `back` Funktion wird in Kombination mit der Sensor-Funktion `blocked` verwendet und liefert den Wert 1 zurück, falls ein Hindernis sich hinter dem Bot befindet, bzw. den Wert 0 wenn sich kein Hindernis hinter dem Bot befindet.

ZBL/0: front

ZBL-API: int zb_mdfFront(void);

Die Funktion `front` wird in Kombination mit der `blocked` Sensor-Funktion verwendet und liefert den Wert 1 zurück, falls sich vor dem Bot ein Hindernis befindet, bzw. den Wert 0 wenn sich vor dem Bot kein Hindernis befindet.

ZBL/0: `left`

ZBL-API: `int zb_mdflLeft(void);`

Die `left` Funktion wird in Kombination mit der Sensor-Funktion `blocked` verwendet und liefert den Wert 1 zurück, falls ein Hindernis sich links vom Bot befindet, bzw. den Wert 0 wenn sich kein Hindernis auf der linken Seite des Bots befindet.

ZBL/0: `object`

ZBL-API: `int zb_mdfoObject(void);`

Diese Funktion wird zusammen mit der `face` Funktion verwendet. Der `object` Modifikator sorgt dann dafür, dass der Game-Bot das sich ihm am nächsten befindliche Objekt auswählt und liefert die ID des Objektes zurück.

ZBL/0: `right`

ZBL-API: `int zb_mdfrRight(void);`

Die Funktion `right` wird in Kombination mit der `blocked` Sensor-Funktion verwendet und liefert den Wert 1 zurück, falls auf der rechten Seite vom Bot ein Hindernis ist, bzw. den Wert 0 wenn sich auf der rechten Seite des Bots kein Hindernis befindet.

ZBL/0: `target`

ZBL-API: `int zb_mdftTarget(void);`

Diese Funktion wird zusammen mit der `face` Funktion verwendet. Der `target` Modifikator sorgt dann dafür, dass der Game-Bot das sich ihm am nächsten befindliche Zielobjekt auswählt und liefert die Objekt ID dieses Gegners zurück.

A.3 Game-Bot Kontrollfunktionen

Kontrollfunktionen erlauben es dem Game-Bot, sich in der virtuellen Spielwelt zu bewegen und mit Objekten in der virtuellen Spielwelt zu interagieren. Bei einigen der Kontrollfunktionen kann es sinnvoll sein, dass eine gewünschte Aktion in der Engine, für die sie implementiert wird nicht sofort, sondern über einen bestimmten Zeitraum hinweg ausgeführt werden soll. Ist dies der Fall, so muss der Botprozess für die Dauer der Aktion blockiert werden. Dies tut man, indem man die Funktion `zb_setBusy` bei der Implementierung aufruft. Diese führt dann zur Blockierung des Botprozesses. Man darf in diesem Fall jedoch nicht vergessen, für den Botprozess nach Ausführung der Aktion die Blockierung durch Aufrufen der Funktion `zb_unSetBusy` wieder aufzuheben.

ZBL/0: backstep

ZBL-API: void zb_doBackstep(void);

Die Funktion `backstep` bewegt einen Bot um einen Schritt rückwärts. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein.

ZBL/0: crawl

ZBL-API: void zb_doCrawl(void);

Die `crawl`-Funktion bewegt einen geduckten Bot um einen Schritt nach vorn (vorwärts kriechen). Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots die `step` Funktion aufrufen.

ZBL/0: duck

ZBL-API: void zb_doDuck(void);

Die Funktion `duck` sorgt dafür, dass sich der Bot auf der Stelle duckt (z.B. um hinter einem Hindernis in Deckung zu gehen). Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` leer definiert werden.

ZBL/0: face <modifier>

ZBL-API: int zb_doFace(int);

Diese Funktion dreht den Bot so, dass er sich dem im Parameter <modifier> genannten Objekt zuwendet. Dabei wird entweder der Modifikator `object` oder der Modifikator `target` benutzt. Die `face` Funktion gibt die ID des über den Modifikator bestimmten Zielobjektes zurück. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` leer definiert werden.

ZBL/0: fire

ZBL-API: void zb_doFire(void);

Die Funktion `fire` lässt den Bot mit der aktuell ausgewählten Waffe schießen (ein einzelner Schuss wird abgegeben).

ZBL/0: jump

ZBL-API: void zb_doJump(void);

Aufrufen der `jump` Funktion lässt einen Bot vorwärts springen. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots die `step` Funktion aufrufen.

ZBL/0: `jump_back`

ZBL-API: `void zb_doJumpBack(void);`

Aufrufen der Funktion `jump_back` führt zum rückwärts springen eines Bots. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots die `backstep` Funktion aufrufen.

ZBL/0: `jump_left`

ZBL-API: `void zb_doJumpLeft(void);`

Diese Funktion lässt einen Bot um eine Einheit (Distanz abhängig von Host-Anwendung) nach links springen. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots die `strafe_left`-Funktion aufrufen.

ZBL/0: `jump_right`

ZBL-API: `void zb_doJumpRight(void);`

Der Aufruf dieser Funktion lässt einen Bot um eine Einheit (Distanz abhängig von Host-Anwendung) nach rechts springen. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots die `strafe_right`-Funktion aufrufen.

ZBL/0: `jump_up`

ZBL-API: `void zb_doJumpUp(void);`

Diese Funktion lässt den Bot auf der Stelle senkrecht hochspringen (Gegensstück zur Funktion `duck`). Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` leer definiert werden.

ZBL/0: `step`

ZBL-API: `void zb_doStep(void);`

Die `step` Funktion bewegt einen Bot um einen Schritt nach vorn. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein.

ZBL/0: `strafe_left`

ZBL-API: `void zb_doStrafeLeft(void);`

Die Funktion `strafe_left` bewegt einen Bot um einen seitlichen Schritt nach links. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` leer definiert werden.

ZBL/0: strafe_right

ZBL-API: void zb_doStrafeRight(void);

Diese Funktion bewegt einen Bot um einen seitlichen Schritt nach rechts. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein. Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` leer definiert werden.

ZBL/0: turn <angle>

ZBL-API: void zb_doTurn(double);

Die `turn` Funktion dreht einen Bot um den im Parameter `<angle>` gegebenen Winkel (in Grad). Dabei wird für eine Drehung nach links (gegen den Uhrzeigersinn) ein positiver Winkel benutzt, wogegen eine Drehung nach rechts (im Uhrzeigersinn) einen negativen Winkel benötigt. Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein.

ZBL/0: turn_left

ZBL-API: void zb_doTurnLeft(void);

Die Funktion `turn_left` dreht einen Bot um 90 Grad nach links (gegen den Uhrzeigersinn). Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein.

ZBL/0: turn_right

ZBL-API: void zb_doTurnRight(void);

Die `turn_right` Funktion dreht einen Bot um 90 Grad nach rechts (im Uhrzeigersinn). Bei der Implementierung dieser Funktion kann das Blockieren des Botprozesses sinnvoll sein.

ZBL/0: use <object>

ZBL-API: void zb_doUse(int);

Diese Funktion wählt das über den Parameter `<object>` gegebene Objekt aus und macht es zum aktuell vom Bot benutzten Gegenstand (Werkzeug, Waffe etc.). Wird diese Funktion nicht benötigt, dann sollte sie bei der Implementierung des Bots bis auf ein parameterloses `return` leer definiert werden.

A.4 Game-Bot Sensor-Funktionen

Sensor-Funktionen dienen dazu, dem Game-Bot Informationen über sich selbst und seine Umgebung zu vermitteln. Auf diesen Informationen basiert das Wissen des Game-Bots, welches ihm erlaubt, sich in der virtuellen Spielwelt zurechtzufinden und zu agieren.

ZBL/0: `alive`

ZBL-API: `int zb_checkAlive(void);`

Die `alive` Funktion gibt den Wert 1 zurück, wenn der Bot (noch) am Leben ist und 0 wenn nicht.

ZBL/0: `armour`

ZBL-API: `double zb_checkArmour(void);`

Die Funktion `armour` liefert den Wert der Rüstung / des Schutzpanzers des Bots zurück. Wenn der Bot keine Rüstung/keinen Schutzpanzer besitzt weil diese Funktionalität nicht in der Host-Anwendung implementiert ist bzw. diese Funktion nicht benötigt wird, so sollte sie den Wert 1 zurückgeben.

ZBL/0: `blocked <modifier>`

ZBL-API: `int zb_checkBlocked(int);`

Diese Funktion stellt fest, ob der Weg des Bots in Richtung des im Parameter `<modifier>` genannten Richtungsmodifikators frei, bzw. blockiert ist. Dabei wird einer der Modifikatoren `left`, `right`, `front` oder `back` benutzt. Die `blocked` Funktion gibt den Wert 1 (bzw. den vom Richtungsmodifikator zurückgelieferten Wert) zurück, wenn der Weg des Bots blockiert ist, bzw. 0, wenn der Weg frei ist.

ZBL/0: `health`

ZBL-API: `double zb_checkHealth(void);`

Die Funktion `health` liefert den aktuellen Gesundheits- bzw. Energiewert des Bots zurück. Wenn diese Funktion in der Host-Anwendung nicht benötigt wird, so sollte sie den Wert 1 zurückgeben.

ZBL/0: `object_ahead`

ZBL-API: `int zb_checkObjectAhead(void);`

Die Funktion `object_ahead` liefert den Wert 1 zurück, wenn der Bot einem Objekt, das kein Zielobjekt ist und welches ihm den Weg verstellt zugewendet ist.

ZBL/0: `obstacle`

ZBL-API: `int zb_checkObstacle(void);`

Die `obstacle` Funktion stellt fest, ob sich direkt vor dem Bot ein Hindernis befindet. Wenn ja, so liefert sie die Objekt-ID des Hindernisses oder alternativ den Wert 1 zurück, wenn nicht, dann liefert sie den Wert 0 zurück.

ZBL/0: `owns <object>`

ZBL-API: `int zb_checkOwns(int);`

Diese Funktion gibt den Wert 1 zurück, wenn das durch den Parameter `<object>` spezifizierte Objekt im Inventory bzw. Besitz des Bots ist. Wenn

nicht, so wird von der Funktion `owns` der Wert 0 zurückgegeben. Wenn diese Funktion in der Host-Anwendung nicht benötigt wird, so sollte sie den Wert 1 zurückgeben.

ZBL/0: `target_ahead`

ZBL-API: `int zb_checkTargetAhead(void);`

Die Funktion `target_ahead` liefert den Wert 1 zurück, wenn der Bot einem Ziel zugewendet ist. Diese Funktion ist die Zielerfassung des Bots.

ZBL/0: `target_alive`

ZBL-API: `int zb_checkTargetAlive(void);`

Diese Funktion gibt den Wert 1 zurück, falls das dem Bot am nächsten befindliche Ziel am Leben ist. Wenn nicht, so sollte diese Funktion den Wert 0 zurückgeben. Anhand dieser Funktion kann ein Bot entscheiden, ob ein Ziel weiterhin eine Bedrohung darstellt oder ob es ignoriert werden kann.

ZBL/0: `target_armour`

ZBL-API: `double zb_checkTargetArmour(void);`

Die `target_armour` Funktion liefert den Wert der Rüstung / des Schutzpanzers des dem Bot nächsten Zielobjektes zurück. Wenn diese Funktion in der Host-Anwendung nicht benötigt wird, so sollte sie den Wert 0 zurückgeben.

ZBL/0: `target_health`

ZBL-API: `double zb_checkTargetHealth(void);`

Die Funktion `target_health` liefert den aktuellen Gesundheits- bzw. Energiewert des dem Bot am nächsten befindlichen Gegners zurück. Wenn diese Funktion in der Host-Anwendung nicht benötigt wird, so sollte sie den Wert 0 zurückgeben.

ZBL/0: `using`

ZBL-API: `int zb_checkUsing(void);`

Diese Funktion gibt die Objekt-ID des aktuell vom Bot benutzten Objekt (bzw. der aktuell genutzten Waffe) zurück. Die Funktion `using` ist damit das Sensor-Funktion Gegenstück zur Kontrollfunktion `use`. Wenn diese Funktion in der Host-Anwendung nicht benötigt wird, so sollte sie den Wert 0 zurückgeben.

A.5 Andere Funktionen in ZBL/0

ZBL/0: `rnd <limit>`

Diese Funktion ist innerhalb der ZBL/0 Virtual Machine definiert und muss deshalb nicht in der Host-Anwendung implementiert werden. Sie liefert eine Zufallszahl zwischen 0.0 und dem Wert des Parameters `<limit>` zurück.

B CD-ROM und begleitendes Material zu diesem Buch

Auf der CD-ROM zu diesem Buch befindet sich zunächst einmal natürlich der gesamte lauffähige Quellcode der im Rahmen dieses Buches entwickelt wurde. Ihr könnt also die ganzen Beispiele der einzelnen Kapitels in den entsprechenden Unterverzeichnissen finden, sie ausprobieren und dann nach Herzenslust verändern und mit ihnen herumspielen. Sie haben es nicht anders verdient :-).

Quelltexte

Daneben befinden sich auf der CD-ROM noch einige andere nützliche Dinge. Natürlich ist eine aktuelle Version des DirectX SDK unabdingbar, denn schließlich haben wir ja die meisten Komponenten unserer Engine mit einer Implementierung in DirectX umgesetzt. Damit ihr auch gleich loslegen könnt findet ihr auf der CD-ROM auch eine Autoren-Version der Entwicklungsumgebung Microsoft Visual C++ 6.0. Mit dieser Version ist es zwar nicht erlaubt, Anwendungen zu erstellen die man in irgendeiner Form vertreiben oder frei verteilen darf. Aber man kann damit dennoch vollwertig ohne weitere Einschränkungen C/C++ Programme erstellen und damit die Beispiele aus diesem Buch auch praktisch nachvollziehen.

Tools

Da man eine CD-ROM leider nur schwer updaten kann, außer bei einer Neuauflage des Buches, kann natürlich das Material auf der CD-ROM nicht ständig aktuell gehalten werden. Insbesondere was Fehler im Quelltext angeht, die erst nach der Veröffentlichung des Buches bekannt werden, beispielsweise weil aufmerksame Leser diese an den schlampigen Autoren melden.

Updates

Es ist also durchaus sinnvoll, neben der CD-ROM noch ein anderes Medium zu nutzen, um euch die Quelltexte und andere Updates zu diesem Buch anzubieten. Und was wäre da besser geeignet als das Internet? Auf der Webseite von ZFX unter <http://www.zfx.info> findet ihr einen Menüpunkt namens BÜCHER und dort werdet ihr einen Link zu einer Sektion für dieses Buch entdecken. Auf der entsprechenden Webseite findet sich alles, was rund um dieses Buch aktuell an Updates und weiteren Informationen verfügbar ist.

C Internetseiten rund um die Spieleentwicklung

<http://www.zfx.info> [deutsch]

Auf der Homepage von ZFX findet sich alles, was das Herz des Hobby-Spieleentwicklers begehrt. Viele ausführliche Tutorials, aktuelle News und ein belebtes Diskussionsforum sowie Screenshots von Hobby-Projekten.

<http://www.flipcode.com> [englisch]

Flipcode bietet neben Screenshot anderer Entwickler insbesondere aktuelle News aus dem semi-professionellen und dem professionellen Bereich. Dazu ein großes Forum und Tutorials und Artikel.

<http://www.gamasutra.com> [englisch]

Das wohl Interessanteste an der Webseite sind die Postmortem-Artikel, in denen professionelle Entwickler den Entstehungsprozess ihrer Spiele beschreiben. Daneben finden sich insbesondere aktuelle News aus dem professionellen Bereich.

<http://www.gamedev.net> [englisch]

Was soll man da sagen? Gamedev.net halt. Eine der größten Seiten zu dem Thema überhaupt. Haufenweise brandaktuelle News aus dem semi-professionellen und dem professionellen Bereich. Viele Tutorials und die wohl belebtesten Diskussionsforen der Szene.

<http://www.vterrain.org> [englisch]

Wie oft habe ich diese URL schon irgendwo gepostet? Und trotzdem scheint sie niemand zu kennen. Es handelt sich dabei um die Top-Adresse, wenn man etwas über Terrain-Programmierung lernen will.

<http://nehe.gamedev.net> [englisch]

Auch zu NeHe muss man eigentlich nichts mehr sagen. Dort spricht man zwar nur OpenGL, aber das sehr ausführlich. Wer mehr über OpenGL lernen will, findet keinen besseren Platz dafür.

<http://www.magic-software.com> [englisch]

Auf der Homepage von David Eberly findet man nicht nur Informationen über die Bücher die er zu diesem Thema verfasst hat, sondern auch jede Menge Quelltexte zu allen Bereichen rund um 3D-Engines.

<http://www.realtimerendering.com> [englisch]

Und noch eine Webseite zu noch einem Buch. Hier findet man insbesondere eine ganze Menge an Links auf verschiedene Seiten mit interessanten Implementierungen.

<http://www.mr-gamemaker.co.uk> [englisch]

Das waren noch Zeiten, als es den guten alten Mr. Gamemaker noch gab. Heute besteht die Seite »nur« noch aus einem Forum, aber dort kann man Leute mit viel Ahnung treffen. Darunter auch Microsoft-Entwickler.

<http://www.scherfgen-software.net> [deutsch]

Autoren unter sich. Auf der Seite von David findet ihr eine kleine Auswahl sehr interessanter Tutorials, sowie ein kleines Diskussions-Forum.

<http://www.codesampler.com> [englisch]

Auf dieser Seite findet man einen Haufen Tutorials sowohl zu OpenGL als auch zu DirectX. Teilweise kann man die einzelnen Projekte mit identischem Output daher in beiden APIs vergleichen.

<http://www.c-plusplus.de/forum/> [deutsch]

In den Foren der C-PlusPlus Webseite findet sich unter anderem ein gut frequentiertes Forum für Spiele- und Grafikprogrammierung.

<http://www.dth.de> [deutsch]

Kauft die CDs, Leute. Wir sehen uns auf dem nächsten Konzert. :-)

D Epilog

*»Rydell had a theory about virtual real estate. The smaller and cheaper the physical site of a given operation, the bigger and cheesier the website. According to this theory, Selwyn FX was probably operating out of rolled-up newspaper.«
(William Gibson , All Tomorrow's Parties)*

Nun sind wieder über Neunhundert Seiten ins Land gezogen, und wir stehen hier am Ende einer langen Reise. Sind erschöpft von den Strapazen des Verarbeitens unseres neu erworbenen Wissens und doch voller Tatendrang, das Gelernte zu nutzen um die virtuellen Welten in unseren Köpfen im Computer zum Leben zu erwecken und die symbolische Zeitung von William Gibson auf die Größe eines zusammengeknüllten Papier-Taschentuchs zu verkleinern.

Auf den vorangegangenen Seiten haben wir zusammen eine Reise durch die schier unendlichen Weiten der 3D-Programmierung unternommen. Ausgehend von den Grundlagen der DirectX-API haben wir eine einfache 3D-Engine entwickelt, auf deren Grundlage wir ein umfassendes, komplexes Tool entwickelt haben um 3D-Level über eine grafische Benutzer-Oberfläche zu erstellen. Diese beiden Komponenten, den Level-Editor und die 3D-Engine, haben wir dann eingesetzt, um ein einfaches, netzwerkfähiges 3D-Spiel zu programmieren.

Für ein einziges Buch haben wir damit schon eine ganze Menge geschafft. Dennoch möchte ich an dieser Stelle darauf hinweisen, dass ein Buch immer nur ein Buch ist, und damit zu einer Moment-Aufnahme mit vergleichsweise langer Belichtungszeit im Entwicklungsstadium der 3D-Grafik-Programmierung avanciert. Die in diesem Buch gezeigten Techniken und Ansätze sind größtenteils auf die Hardware ausgerichtet, die zum Erscheinen dieses Buches aktuell ist. Ich für meinen Teil möchte nicht darüber spekulieren, ob wir eventuell in drei oder vier Jahren Grafikkarten verwenden, die auf Voxel und nicht mehr auf Triangle ausgerichtet sind. Dies ist nur ein Beispiel, wenn auch ein besonders krasses, was sich in unserem Feld innerhalb kürzester Zeit um 180 Grad drehen kann, und eine entsprechend angepasste Programmierung voraussetzt.

Daher ergehen an dieser Stelle meine mahnenden Worte an euch, dieses Buch als Inspiration zu begreifen, und nicht als Quelle für optimalen Quelltext den man in seine eigenen Projekte ohne weitere Gedanken übernehmen kann. Jedes Buch, egal welchen Umfang es nun hat, stellt eine diskret abgegrenzte Menge an Information zu einem klar fixierten Zeitpunkt dar. Die absolut verfügbare Menge an Information zu diesem Thema ist aber weitaus mächtiger – die implizite Menge an potenziell vorhandener Information in all den schlaun Köpfen da draußen ist aber noch weitaus umfassender.

Mehr als in jedem anderen Berufsfeld gilt für Programmierer, die am Rande der Möglichkeiten der Hardware und darüber hinaus arbeiten, dass sie die Verpflichtung haben sich über neue Technologien zu informieren und dem aktuellen Stand der Masse um ein Jahr voraus sein müssen. Euer Job ist es daher, ständig das Internet auf der Suche nach Informationen und neuen Trends zu durchforsten, um noch bessere und noch schnellere Grafik-Applikationen zu entwickeln.

Ich für meinen Teil werde dieses Dokument gleich abspeichern und das damit komplettierte Manuskript an den Verlag übergeben. Wieder liegt über ein Jahr Arbeit hinter mir, und wieder hat es mir unheimlich viel Spaß gemacht, dieses Werk mit all seinen Höhen und seinen Tiefen zu schreiben. Und wieder warte ich gespannt auf das Feedback, was mich hoffentlich wieder so zahlreich erreichen wird. Ich hoffe, dass ich einige der Wünsche und der konstruktiven Kritik aus dem bisherigen Feedback zu meiner Arbeit in eurem Sinne umsetzen konnte. Ebenso hoffe ich, dass euch dieses Buch als nützlicher Gefährte ein Stück weit auf eurem Weg begleiten wird. Ganz besonders würde es mich freuen, wenn ihr euch später immer noch daran erinnern könnt, wie ihr einmal angefangen habt, und dass ihr euch dann darauf besinnt anderen Anfängern eine helfende Hand zu reichen.

In diesem Sinne wünsche ich euch viel Erfolg bei dem 3D-Projekt, welches sich da gerade in euren Köpfen manifestiert, und warte auf die Einsendung eines Screenshots für die www.ZFX.info Webseite und euer Feedback und die konstruktive Kritik per E-Mail an Stefan.Zerbst@zfx.info.

Stefan Zerbst, Braunschweig den 8. Oktober 2003

Stichwortverzeichnis

- !**
 - `__declspec(aligned(16))` 128
 - `__declspec(dllexport)` 78
- 16 Bit aligned** 128
- 3D-Engine** siehe **Engine**
- 3DNow!** 126
- 3DNow!, Enhanced** 126
- 3DNow!-Professional** 132
- 3D-Pipeline** 363, 605
- 3D-Pipeline** siehe **Shader**
- 3ds max** 694
- 3D-Sound** 509, 520
- 4D-Raum** 203
- A**
 - AABB** 176
 - Ebenen erstellen 185
 - Extrempunkte 176
 - Schnitt mit Strahl 186
 - Achsen** siehe **Kamera**
 - ActiveX** 512
 - AGP-Bus** 308
 - Ambient Light** siehe **Licht, ambientes**
 - Anatomie** siehe **Biped-Anatomie**
 - Animation**
 - Bindposition 417
 - Bones 415
 - Charaktere 411
 - Joint 415
 - über Keyframes 413
 - über Skin und Bones siehe **skeletale Animation**
 - Vertex-Transformation 459
 - Anweisungen**
 - ADD** 122
 - ADDPS** 127
 - ADDSS** 127
 - CPUID** 135
 - MOV** 121
 - MOVAPS** 128
 - MOVD** 125
 - MOVQ** 125
 - MOVUPS** 128
 - PADD** 125
 - PFADD** 126
 - SHUFPS** 130
 - API** siehe **Application Programming Interface**
 - Apollo 13** 205
 - Application Programming Interface** 28, 38
 - API-Unabhängigkeit 60, 66
 - DirectX vs OpenGL 59
 - Arithmetic Logical Unit** siehe **Vertex-ALU** bzw. **Pixel-ALU**
 - Assembler** 118
 - asynchrone Kommunikation**, **Windows Sockets API** 564
 - Attenuation** 390
 - Attenuation Map** 390
 - Audiopath** siehe **DirectMusic**
 - AutoCAD** 694
 - Autonomer Agent** 877
 - Axis Aligned Bounding Box** siehe **AABB**
- B**
 - Backface** 191, 724
 - Backface Culling** 607
 - Backface Removal** siehe **Backface Culling**
 - Back-to-Front** 625, 657
 - Batching** siehe **Caching**
 - Berkeley Sockets API** 533
 - Bibliotheken** 77
 - DLLs und VC++ .NET 92
 - Dynamic Link Library (DLL) 77
 - Funktionen in DLLs finden 88
 - Funktionen über DEF Files exportieren 91
 - statische 77
 - Big Endian** 547
 - Binary Space Partitioning (BSP)** siehe **Scene-Management**
 - Bindposition** 417
 - Binormale** 400
 - Biped-Anatomie** 416
 - Bitbefehle** 119

Bitmaps 245
Bitshifting 399
Block 616
Blocking-Calls siehe Netzwerk
Bones 415
 rendern 464
 Setup 447
Bottleneck 366
Bounding Box siehe AABB bzw. OBB
Bounding-Box, berechnen 722
Broadcasting 132
Brush-Objekte 705
Brute-Force siehe Scene-Management
BSP siehe Scene-Management
Budget 53
build! 635
Bump-Mapping 220, 394
 Perturbed Normal 398
 Tangent Space 398, 400
Bus 309
Bus-Traffic 364
Bytecode-Programme 885
Byte-Reihenfolge 547

C

Caching 310
CAD siehe Computer Aided Design
Capture the Flag 525
Carmacks Reverse siehe Schatten, Shadow-Volume (Z-Fail)
CBF siehe Chunkbased Format
Central Processing Unit siehe CPU
Chat-Programm 574
Chunk 419
Chunkbased Format (CBF) 418
 Animation.txt File 438
 Chunk-ID 419
 Header 422
Client-Server-Architektur 528
Clipping
 Polygon an einer AABB 192
 Polygon an einer Ebene 194
Clipping Planes 271
COM siehe Component Object Model
Competitive Analysis 52
Component Object Model
 CoCreateInstance() 512
 ColInitialize() 512
 CoUninitialize() 515

Component Object Model (COM) 512
Computer Aided Design (CAD) 693
Constructive Solid Geometry (CSG) 705
Cooperative Level siehe DirectInput
CPU 117
 limited 364
Cracks 618
CSG siehe Constructive Solid Geometry
Culling 179
 mit Bounding Boxen 199
 mittels AABB 179
 mittels OBB 181
Cut-Brush siehe Brush-Objekte

D

D3DXFont 345
Deathmatch 524
dedicated Server siehe dezidiert Server
delete 704
Depth-Fail siehe Schatten, Shadow-Volume
Depth-First 857
Depth-Pass siehe Schatten, Shadow-Volume
Dequeue 540
Design-Dokument siehe Lead Programmer
Detailmap 394
dezidiert Server 528
Dialoge mit der Win32 API siehe Fehlerwerte
Diffusreflexionsmodell 225
Direct Light siehe Licht, direktes
Direct3D
 3D-Pipeline 363
 D3DPRESENT_PARAMETERS 105
 dynamic Buffer Flags 319
 FVF-Codes 290
 HAL-Device 94
 initialisieren 103
 Lock-Flags 322
 REF-Device 94
DirectAudio siehe DirectMusic bzw. DirectSound
DirectInput 474
 Cooperative Level 477
 Daten-Abfrage 478
 Fehlerwerte 481
 Initialisierung 476
 Joystick siehe Joystick
 Keyboard siehe Keyboard
 Maus siehe Maus
 Tastatur siehe Keyboard
Directional Light 220

DirectMusic

- Audiopath 509
- Default-Audiopath 509
- Initialisierung 514
- Loader 511
- Performance-Objekt 511
- Primary-Segment 519
- Secondary-Segment 519
- Segment 509
- Sounds abspielen 519
- Suchpfad 513
- Tutorials, SDK 505

DirectPlay 534**DirectSound**

- 3D-Sound-Quelle 520
- Listener 511, 520

DirectX 61

- D3DX Bibliothek 62

Discreet siehe 3ds max**diskretes LOD** siehe Scene-Management**DLL** siehe Bibliotheken**DLOD** siehe Scene-Management**DLS-Format** siehe Downloadable Sounds**Downloadable Sounds (DLS)** 516**Dualsystem** 130**Duke Nukem 3D** 635**Dynamic Link Library** siehe Bibliotheken**E****Early-Z-Culling** 638**Ebene** 169**Engine** 37

- Aufgaben 39
- Beispiel 39
- ZFXEngine 2.0 65

Enhanced 3DNow! 126**Enqueue** 540**Entities** 756

- Türen 756
- Typen 758
- Verbindung zu Portalen 760

Enumerationen

- ZFXENGINEMODE 264
- ZFXENTITY 757
- ZFXINPUTDEV 473
- ZFXNETMODE 557
- ZFXRENDERSTATE 302
- ZFXVERTEXID 289

Extended Real 122**F****Fall Off** siehe Attenuation**Far Clipping Plane** siehe Clipping Planes**Farbformate** 246**Farbkonvertierung** 246**Fehlerwerte** 93**FIFO-Warteschlange** siehe Queue**File-Funktionen** 584**File-Transfer-Programm** 579**Fixed-Function-Pipeline** 362**Flexible-Pipeline** siehe Shader**Flexible-Vertex-Format (FVF)** 289**Floating Point Unit** siehe FPU**Font** 344**FPU** 122**Fragger** 523**Fragrate** 524**Fraqs** 523**Friendly Fire** 525, 609**Front (einer Queue)** 541**Front-Cap** 826**Frontend** 815**Frontface** 191, 724**Front-To-Back** 626**Front-to-Back** 656**Frustum Culling** 607**Funktionen**

- accept() 549
- bind() 546
- BuildAndSetShader() 374
- CalcTangentSpace() 403
- CalcTransAttenNoRot() 386
- CheckMem() 704
- CoCreateInstance() 512
- Colnitalize() 512
- connect() 550
- CoUninitialize() 515
- CreateRenderDevice() 92
- D3DXAssembleShader 298
- D3DXAssembleShaderFromFile 298
- DialogBox() 95
- EndDialog() 99
- free() 704
- GetCPUInfo() 137
- htonl() 548
- htons() 548
- InitMem() 703
- listen() 548
- malloc() 704
- ntohl() 548

ntohs() 548
OSSupportsSSE() 140
realloc() 704
recv() 553
ReleaseRenderDevice() 93
RenderPolylist() 679
send() 552
socket() 545
VectortoRGBA() 400
WSAAsyncSelect() 565
WSACleanup() 562
WSAStartup() 562
ZFX3DInitCPU() 140

FVF siehe **Flexible-Vertex-Format**

G

Game Treatment 51
Game-Proposal 50
 Gliederung 51
Game-Server 870
gerichtetes Licht siehe **Directional Light**
GetProcAddress() 88
Gimbal Lock 205
Gimbal-Lock 590
Glowmaps 379
GPU 117
 -limited 364
Grafikkarten-Architektur 308, 363
Graphic Processing Uni siehe **GPU**
GUI siehe **Interface**

H

Half-Life 412
Hammer-Editor 694
Hardware-Architektur 308
Heap 121, 648
Heightmap 614
Hexadezimalsystem 130
Hidden Surface Removal (HSR) 308, 606
Hochsprache 119
Host Byte Order siehe **Netzwerk**

I

id Software 628
IDE siehe **Integrated Development Environment**
Inbox siehe **Queue**
Index-Buffer
 dynamisch 311
 statisch 311

Indoor 607
Infix-Notation 885
Input siehe **DirectInput**
Instinkt 902
Integrated Development Environment (IDE) 29
Interface 68, 73
 Graphical User Interface (GUI) 73
 Sinn und Zweck 74
 ZFXAudioDevice 507
 ZFXInputDevice 473
 ZFXNetworkDevice 558
 ZFXRenderDevice 83, 263
 ZFXSkinManager 231
 ZFXVertexCacheManager 313
Interpreter-Code 885
IP-Adresse siehe **Netzwerk**
IPX siehe **Netzwerk-Protokolle**
ISO 530

J

Java Virtual Machine (JVM) 878
Joint 415
Joystick 490
 Daten-Abfrage 479, 493
 Enumeration 491
 Flattern 501
 Initialisierung 491

K

Kamera 590
 1st Person Kamera 590
 3rd Person Kamera 591
 Achsenneuberechnung 600
 fixierte Kamera 592
 freie Kamera 590
 im Shadow-Volume 824
 Motion-Sickness 591
 Rotationsreihenfolge 601

Keyboard

 Daten-Abfrage 479
 Initialisierung 482
 Tastenstatus 484

Keyframe-Animation 413

KISS-Prinzip 508

Klassen

 abstrakte 74
 CApplication 807
 – OnPolygonCreationEvent 812
 – Update 811

- CEntity 757
 - ConnectTo 761
 - RemovePortalConnection 760
 - SetSubType 759
- CGame
 - ProcessNetwork 872
 - SendMsg_PlayerlistTo 874
 - Update 871
- CGameCharacter 866
 - SendMsg_TouchAndUse 869
- CGameLevel
 - BuildAllPortalRelationships 853
 - FindBrother 854
 - Load 842
 - Render 856
 - RenderShadows 863
- CGameSector
 - BuildAllPortalRelationships 854
 - CompilePolygonList 850
 - CreateOctree 844
 - FindBrother 855
 - Load 843
 - LoadMeshs 845
 - LoadSinglePolygon 847
 - RenderDepth 859
 - RenderLightPass 862
 - RenderLit 860
 - RenderShadows 864
- CLevel 790
 - LoadLevel 792
 - SaveLevel 791
- CLevelObject 702
- CLight 753
 - CopyOf 755
 - Init 754
 - Load 755
 - Save 755
- CPolygon 706
 - CalcBoundingBox 723
 - CalcNormals 723
 - CalcTextureCoords 720
 - CopyOf 712
 - CreatePortal 725
 - GetIntersectingVerts 714
 - GetTextureTrans 722
 - Init 707
 - InsideOut 724
 - Intersects 713
 - Load 711
 - Mirror 719
 - Picked 715
 - Release 708
 - Render 716
 - Rotate 717
 - Save 710
 - SetAsPartOfMesh 724
 - SetIndices 709
 - SetVertices 709
 - Translate 716
 - TransTextureCoords 721
- CPolymesh 726
 - AddPolygon 729
 - CalcBoundingBox 743
 - CalcTextureCoords 738
 - CopyOf 734
 - CreateFromPolygons 728
 - GetIntersectingPolys 735
 - GetIntersectingVerts 735
 - GetPolygon 743
 - GetTextureTrans 739
 - HideInvertPolys 742
 - HideNoPolys 741
 - HideSelectedPolys 741
 - HideUnselectedPolys 742
 - Init 728
 - Intersects 734
 - Load 731
 - Picked 736
 - PickPoly 736
 - Release 728
 - RemovePolygon 730
 - Render 737
 - Save 731
 - SavePurePolys 732
 - SelectAllPolys 740
 - SelectInvert 740
 - TransTextureCoords 739
- CPortal 746
 - CopyOf 750
 - Init 747
 - Load 749
 - Release 748
 - Save 749
 - SetGeometry 748
 - SetItsTarget 751
- CSector 764, 767
 - AddObject 784
 - GetName 766
 - HideInvert 781
 - HideNone 780

Stichwortverzeichnis

- HideSelected 780
- HideUnselected 781
- Init 768
- Load 772
- LoadPrefab 774
- NameEqual 767
- NewDoorInTown 789
- NewPortalInTown 788
- Pick 776
- Release 769
- RemoveObject 782
- RemovePortalConnections 789
- Render 785
- RenderLit 787
- Save 770
- SelectAll 777
- SelectIntersecting 778
- SelectInvert 778
- SetName 766
- SetPointers 765
- CSelectionBuffer 794
 - AddLob 797
 - AddVertex 797
 - CopySelectedObjects 800
 - DeleteSelectedObjects 798
 - DestroyWithoutNotify 799
 - FragmentMeshes 802
 - Init 796
 - MergePolysToMesh 801
 - Release 796
 - Reset 796
 - SaveAsPrefab 804
- CShadowVolume 828
 - AddEdge 831
 - BuildShadowVolume 832
 - Extrude 831
 - Init 829
 - Render 834
- CSpawnPoint 762
- CZFXModel 419
 - Animation 452
 - AnimationPrepare 454
 - AnimationVertices 460
 - CheckForChunks 421
 - GetNextChunk 421
 - Prepare 443
 - ReadAnimations 438
 - ReadFaces 427
 - ReadHeader 423
 - ReadJoint_KeyFrame_Pos 436
 - ReadJoint_KeyFrame_Rot 435
 - ReadJoint_Main 434
 - ReadJoints 432
 - ReadMaterials 430
 - ReadMesh 428
 - ReadVertices 425
 - Render 463
 - RenderBones 465
 - RenderNormals 467
 - SetScaling 440
 - SetupBones 448
 - Update 463
- zbl_vm 888f.
- zblbot 879, 881
- ZFXAabb 177
 - Construct 178
 - Contains 186
 - Cull 180
 - GetPlanes 185
- ZFXBspTree 649
 - BuildTree 651
 - CreateChilds 652
 - FindBestSplitter 654
 - LineOfSight 659
 - TestCollision 658
 - TraverseBtF 657
 - TraverseFtB 656
- ZFXD3D 90
 - ActivatePShader 301
 - ActivateVShader 300
 - BeginRendering 107
 - CalcPerspProjMatrix 276
 - CalcViewProjMatrix 277
 - CalcWorldViewProjMatrix 278
 - CreateFont 346
 - CreateVShader 296
 - DlgProc 96
 - DrawText 347
 - GetFrustum 270
 - GetShadeMode 305
 - Go 103, 352
 - Init 101
 - InitStage 281
 - OneTimelnit 353
 - Prepare2D 275
 - PrepareShaderStuff 292
 - Release 100
 - SetBackfaceCulling 303
 - SetClippingPlanes 272
 - SetDepthBufferMode 304

- SetMode 279
- SetShadeMode 304
- SetView3D 266
- SetViewLookAt 269
- SetWorldTransform 287
- Transform2Dto3D 284
- Transform3Dto2D 282
- UseStencilShadowSettings 836
- UseWindow 106
- ZFXD3DSkinManager 233
 - AddSkin 240
 - AddTexture 243
 - AddTextureHeightmapAsBump 397
 - ColorEqual 237
 - ConvertToNormalmap 398
 - CreateTexture 247
 - GetMaterial 239
 - GetSkin 238
 - GetTextureName 239
 - MakeD3DColor 251
 - MaterialEqual 238
 - SetAlphaKey 253
 - SetTransparency 255
- ZFXD3DVCache 316
 - Add 320
 - Flush 325
 - SetSkin 320
- ZFXD3DVCMManager 328
 - CreateStaticBuffer 338
 - ForcedFlush 336
 - ForcedFlushAll 336
 - InvalidateStates 349
 - Render 334, 341
 - RenderLines 350
 - RenderPoints 349
- ZFXDA 510
 - Init 512
 - LoadSound 516
 - PlaySound 518
 - Release 514
 - SetListener 520
 - SetSoundDirection 521
 - SetSoundMaxDist 521
 - SetSoundPosition 521
 - StopSound 519
- ZFXDI 494
 - GetPosition 498
 - HasJoystick 498
 - Init 496
 - IsPressed 499
 - IsReleased 499
 - Release 495
 - Update 497
- ZFXDIDevice 474
 - CrankUp 477
 - Create 476
 - GetData 480
 - Release 476
- ZFXJoystick 490
 - EnumJoyCallback 493
 - Init 492
 - Update 493
- ZFXKeyboard 482
 - Init 483
 - IsPressed 484
 - IsReleased 484
 - Update 483
- ZFXMatrix 154
 - Identity 155
 - RotaArbi 156
 - RotaX 155
 - RotaY 155
 - RotaZ 156
 - Translate 156
 - TransposeOf 156
- ZFXMCEgo 599
 - GetRotation 599
 - RecalcAxes 600
 - SetRotation 599
 - Update 602
- ZFXMCFree 596
 - AddRotationSpeed 597
 - SetRotation 597
 - SetRotationSpeed 597
 - Update 598
- ZFXMouse 485
 - Init 487
 - Update 488
- ZFXMovementController 593
 - Init 594
 - RecalcAxes 595
- ZFXObb 177
 - Cull 181
 - DeTransform 181
 - Intersects 183
 - ObbProj 183
 - TriProj 183
- ZFXOctree 662
 - BuildTree 666
 - ChopListToMe 668

- CreateChilds 667
- GetAabbAsPolygons 676
- GetFloor 674
- InitChildObject 664
- IntersectsDownwardsRay 673
- TestCollision 670f.
- Traverse 676
- ZFXPlane 170
 - Classify 172
 - Distance 172
 - Intersects 173, 175f.
 - Set 171
- ZFXPolygon 188
 - CalcBoundingBox 191
 - Clip 193, 195
 - Cull 200
 - Intersects 202
 - Set 190
 - SwapFaces 192
- ZFXPolylist 646
 - AddPolygon 646
 - CheckMem 647
 - Reset 646
- ZFXQuat 206
 - GetMatrix 210
 - MakeFromEuler 209
- ZFXQueue 539
 - Dequeue 540
 - Enqueue 540
 - Front 541
- ZFXQueueElem 539
- ZFXRay 159
 - DeTransform 160
 - Intersects 161, 163, 165, 167
 - Set 160
- ZFXRenderer 85
 - CreateDevice 87
 - Release 88
- ZFXSocketObject 543
 - Accept 549
 - Bind 547
 - Connect 550
 - CreateSocket 545
 - Disconnect 551
 - GetNextPkg 556
 - Listen 549
 - Receive 554
 - Send 552
- ZFXVector 142
 - AngleWith 146

- Cross 151
- Difference 146
- GetLength 147
- GetSqrLength 146
- Negate 146
- Normalize 150
- Set 145
- ZFXWS 559
 - CreateClient 566
 - CreateServer 564
 - Init 562
 - MsgProc 569
 - OnAccept 571
 - OnDisconnect 572
 - OnReceive 573
 - Release 561
 - SendToClients 569
 - SendToServer 568

Klassifizierung 170, 194

Kollisionen

- Ebenen vs Bounding Boxen 174
- Ebenen vs Dreiecke 172
- Ebenen vs Ebenen 173
- im BSP-Baum 658
- im Octree 670
- Liniensegmente vs Dreiecke 162
- Liniensegmente vs Ebene 164
- Liniensegmente vs OBB 168
- OBB vs Dreiecke 182
- OBB vs OBB 185
- Polygone vs Strahlen 201
- Strahlen vs AABB 165
- Strahlen vs Dreiecke 161
- Strahlen vs Ebenen 162
- Strahlen vs OBB 167

komplexe Zahlen 204

kontinuierliches LOD siehe Scene-Management

Kontrollfunktionen 900

Kreuzprodukt 151

künstliches Verhalten 901

L

LAG 527

Laichplätze siehe Spawn-Points

Lara Croft 591

Latency siehe LAG

Lead Designer 43, 49

Lead Programmer 48

Leaf 612

Least Significant Bit (LSB) 547

Level-Editor

- als Basis für Spiel 816
- Anforderungen 697
- brush-basiert 705
- Exporter 803
- GUI-Attribute 809
- Licht-Quellen 751
- Performance 699
- Polygon-Erstellung 812
- polygonorientiert 705
- Portal-Verbindungen 744
- rendern siehe Level-Objekte
- Schatten-Quellen 751
- Select-Mode 766
- Texturen 775
- Tools 694
- Türen 756

leveln 526

Level-Objekte 700

- hinzufügen 783
- kopieren 712, 801
- löschen 781
- Maus-Auswahl 713
- Picking 715
- rendern 785
- rotieren 718
- selektieren 797
- Textur-Koordinaten 719
- verstecken 779

Level-of-Detail-Ansatz (LOD) siehe Scene-Management

LIB siehe Bibliotheken

Library siehe Bibliotheken

Licht

- ambientes 220
- diffus 225
- direktes 220
- emissive 376
- gerichtetes siehe Directional Light
- Omni-Lights 384
 - Light-Cache 850
- paralleles siehe Licht, gerichtetes
- Quelle 228
- spekulär 225

Light-Cache siehe Licht, Omni-Lights

Lightmaps 819

Line-of-Sight siehe Sichtlinie

Listener siehe DirectSound

Little Endian 547

LoadLibraryEx() 87

LOB siehe Level-Objekte

LOD siehe Scene-Management

Log-File 234

lokaler Host siehe Netzwerk

LUA 911

M

Management 216

Management-by-Objective 230

Master-Socket-Objekt siehe Windows Sockets API

Material 218

- spiegelnd 223
- stumpf 223

Matrizen 153

- transponierte 449f.

Maus

- Daten-Abfrage 480, 488
- Initialisierung 486
- Käfig 489, 497

Memory Manager siehe Speichermanager

Mesh siehe Polymesh

MIG 508

MMORPG 527

MMX 123

Mnemo-Code 118

Mnemonik 118

Modellkoordinatensystem 382

Most Significant Bit (MSB) 547

Motion-Sickness 591

Movement-Controller siehe Kamera

Multi-Media-Extension siehe MMX

Multipass-Rendering 229

Multitexturing 378

Multitexturing-Unit 363

N

Nachrichten von Steuerelementen 98

Near Clipping Plane siehe Clipping Planes

NETBEUI 530

Network Byte Order siehe Netzwerk

Netzwerk

- Blocking-Calls 565
- Byte-Reihenfolge 547
- Empfangen von Daten 554
- IP-Adresse 550
- lokaler Host 563
- Nachrichten 558
- Nachrichten von Characters 869
- Port-Nummer 550

Netzwerk-Protokolle 532

- IPX 530
- TCP/IP 530
- Transmission Control Protocol (TCP) 533
- User Datagram Protocol (UDP) 533

Netzwerk-Spiele

- Capture the Flag 525
- Client-Server 528
- Deathmatch 524
- Peer-to-Peer 528
- persistente Welten 526
- Server-Verteilung 527
- session-basiert 524
- Team bzw. Coop 525

Netzwerk-Traffic 536

new 704

Nodoor 608

Non-Player-Character 877

Normalenvektoren 170

- als RGBA-Codierung 399
- per-Pixel 395
- Perturbed Normal 398
- rendern 466

Normalisierung 149

Normalizing Cube-Maps 408

Normalmap 396

- aus Heightmap 397

NPC siehe **Non-Player-Character**

O

OB 176

Object Culling siehe **Frustum Culling**

Objektkoordinatensystem 382

Occluder 607, 818

Occlusion Culling 606

Omni-Lights 751

- siehe auch **Licht**

Open File Name-Dialog 580

Open Systems Interconnection (OSI) 530

- Application Layer 531
- Datalink Layer 532
- Network Layer 532
- Physical Layer 532
- Presentation Layer 531
- Session Layer 531
- Transport Layer 532

OpenGL 61

Operatoren-Überladung 144

operator-Schlüsselwort 143

Optimierungen

- Alpha-Blending-Deaktivierung 230
- Assembler 119
- Bus-Transfer 216
- Depth-First-Pass 857
- Grafikkarten-Architektur 308
- Light-Cache 850
- Lowlevel 118
- Netzwerk-Paketgröße 537
- Polygon-Count 219
- schnelle Rekursion 653
- Switch-Vermeidung 365
- Triangle Culling 606

Oriented Bounding Box siehe **OB**

Orthogonale Projektion siehe **Projektion**

OSI-Modell siehe **Open Systems Interconnection**

Outdoor 608

Overdraw 626

P

PanBox-Edit 695

Pandoras Legacy

- als Game-Server 870
- Game-Design 816
- Hauptklassen 817
- Netzwerk-Nachrichten 872

Pathfinding 905

Peer-to-Peer 528

Per-Pixel-Lighting 383, 394

persistente Welten siehe **Netzwerk-Spiele**

Perspektivische Projektion siehe **Projektion**

Per-Vertex-Lighting 384

Photon-Mapping 819

Picking 284

Pixel-ALU 368

Pixel-Programs siehe **Shader**

Pixel-Shader siehe **Shader**

Pixel-Treppchen 820

Plane siehe **Ebene**

Point Light 220f.

Polygon-Erstellung 812

Polylines 709

Polymesh 701, 725

- fragmentieren 802

Port-Nummer siehe **Netzwerk**

Postfach siehe **Queue**

Postfix-Notation 885

Post-Rollenspiele 524

Preemptives Multitasking 888

PreFabs 681, 697, 733

laden 773

speichern 773

Primary Segment siehe **DirectMusic**

Probabilistik 907

Progressive Meshs siehe **Scene-Management, CLOD**

Projekt

Dateien hinzufügen mit VC++ 80

dynamische Bibliothek mit VC++ 81

statische Bibliothek mit VC++ 79

Projektion

orthogonale 273

parallele 273

perspektivische 275

Protokolle siehe **Netzwerk-Protokolle bzw. Sternenflotte**

Prozessor-Aufbau 120

Punktlichtquelle siehe **Point Light**

PVS siehe **Scene-Management**

Python 911

Q

Quadtree siehe **Scene-Management**

Quaternions 203

aus Euler-Winkeln 208

Einheits-Quaternion 208

Multiplikation 207

Rotationsmatrix erzeugen 209

Sinn und Zweck 205

Queue 538

als Netzwerkpaket-Postfach 556

Dequeue 540

Enqueue 540

Front 541

R

Radiosity 386, 819

Ray siehe **Strahl**

Ray-Tracing 819

Register

128 Bit SSE-Register 127

FPU-Register 126

MMX-Register 124

normale 32 Bit-Register 120

Pixel Shader-Register 369

Vertex Shader-Register 366

Rekursion 647

Beschleunigung 653

BSP-Baum-Erstellung 651

endlos 648

Octree-Erstellung 665

Stack-Overflow 648

Suche in Bäumen 656

Respawn siehe **Spawn-Points**

reziproke Wurzel 150

Role Playing Games (RPG) 527

Root-Node 612

Rotationsreihenfolge 601

Round-Robin-Prinzip 888

RPG siehe **Role Playing Games**

S

Scene-Management 606

Begrifflichkeiten 606

Binary Space Partitioning (BSP) 623

– achsen-ausgerichtet 623

– Algorithmus 626

– Historie 628

– Implementierung 645

– leaf-based BSP 634

– leaf-based BSP, solid 634

– node-based BSP 633

– Sichtlinie 659

– Sinn und Zweck 624

– Splitter 626, 654

– Splitter-Auswahl 632

Brute-Force-Ansatz 608

continuous Level-of-Detail (CLOD) 610

discreet Level-of-Detail (DLOD) 610

Level-of-Detail-Ansatz 609

Octree 621, 844

Octree, dynamisch 622

Octree-Implementierung 661

Octree-Traversal 675

Performance Octree vs BSP 687

Portal-Engine 635

– automatisierte Portalerzeugung 641

– Bruder-Portal 747

– Portale verbinden 788

– Sinn und Zweck 637

– Spiegel 639

– Star Gates 639

– Tür-Entities 760

– versus BSP-Baum 638

– Zufallsziele 639

Portal-Engines 744

– Area 744

– Sektor 744

Potential Visibility Set (PVS) 643

Quadtree dreidimensional siehe Octree

- Quadtree-Algorithmus 612
- Quadtree-Ansatz für Terrain 616
- Quadtree-Ansatz, allgm. 611
- US-Patente 623
- Schatten** 818
 - Shadow-Maps 819
 - Shadow-Volume
 - Algorithmus 821
 - Back-Cap 820
 - Carmacks Reverse siehe Depth-Fail
 - Depth-Fail (Z-Fail) 825
 - Depth-Pass (Z-Pass) 822
 - Erstellung 832
 - Front-Cap 820
 - Implementierung 827
 - Kamera im Volume 824
 - Stencil-Buffer Einstellungen 836
 - Theorie 820
 - über Shader 838
- Schedule** 54
- Secondary Segment** siehe DirectMusic
- Segment (Audio)** siehe DirectMusic
- Sehen** 222
- Sektor** siehe Portal-Engines
- Sektoren** 700, 762
- Selektieren** 739
- Selektions-Buffer** 707, 793
- Sensorfunktionen** 899
- Serialisierung** 556, 567
- Shader** 290
 - _bx2 389
 - _sat 389
 - add 381
 - dp4 372
 - m4x4 371
 - mul 373
 - tex 373
 - texcoord 389
- Shadow Volumes** siehe Schatten
- Shadow-Caster** 752
- Shadow-Maps** siehe Schatten
- Shufflen** 130
- Sichtlinie** siehe Scene-Management, BSP
- SIMD** 122
 - Compiler-Voraussetzungen 133
 - Performance-Einbußen 147
- Single Instruction Multiple Data** siehe SIMD
- Singlepass-Rendering** 229
- skeletale Animation** 414, 418
- Skin** 219
- Skinned-Mesh** siehe skeletale Animation
- Socket-Objekte** siehe Windows Sockets API
- Sound-Buffer** siehe DirectSound
- Sounds**
 - laden 515
 - mit DirectMusic abspielen 519
 - Wave-Format 516
- Spawn-Points** 701, 761
- Speichermanager** 704
- Spekulärreflexionsmodell** 225
- Spiele-Entwicklung**
 - als Kunst 43
 - berufliche Perspektiven 46
 - Budget 53
 - Beispiel 53f.
 - Competitive Analysis 52
 - Design-Document siehe Lead Programmer
 - Ethik und Moral 44
 - Game Treatment 51
 - Game-Proposal 50
 - Gliederung 51
 - Lead Designer 43, 49
 - Lead Programmer 48
 - Schedule 54
 - Team Introduction 52
 - Verkaufserlöse 56
 - Verkaufszahlen 57
 - Veröffentlichung 55
- Spiel-Entwurf** siehe Game-Proposal
- Splitter** siehe Scene-Management, BSP
- Spot Light** 220f.
- SSE** 126
- SSE2** 132
- Stack** 121
- Stack Machine** siehe Stack-Automat
- Stack-Automat**
 - paralleler 884
 - Programmablauf 886
 - Trace für Addition 885
- Stack-Overflow** 648
- Stand-Alone-Server** 536
- Stencil-Buffer** 821
- Stencil-Shadows** siehe Schatten, Shadow-Volume
- Strahlen** 158
 - Transformationen 160
- Streaming SIMD Extensions** siehe SSE
- Strukturen**
 - ANIMATION_S 437
 - CHUNK_S 420
 - CHUNKHEAD_S 422

CPUINFO 135
 CVERTEX 425
 D3DVERTEXELEMENT9 294
 ENTITYDOOR 757
 FACE_S 427
 GLOBALS 806
 JOINT_S 431
 KF_POS_S 436
 KF_ROT_S 435
 LVERTEX 289
 MATERIAL_S 430
 MESH_S 428
 POLYGON 842
 SECBUF 841
 sockaddr_in 546
 TANGENTVERTEX 401
 TVERTEX 371
 UPDATEPKG 868
 VERTEX 289
 VERTEX_3F_S 424
 VERTEX_SB 794
 ZFXCLIENT 560
 ZFXCOLOR 224
 ZFXLIGHT 227
 ZFXLIGHTID 227
 ZFXMATERIAL 228
 ZFXPACKAGE 537
 ZFXSKIN 229
 ZFXSOUND 509
 ZFXSTATICBUFFER 329
 ZFXTEXTURE 229
 ZFXVIEWPORT 264

Swap Chains 105
Swap Chains von Direct3D 89
Sybil 878

T

T'ker siehe **Team-Killer**
Tangent Space
 Berechnung 402
 Binormale 400
 Tangent-Vektor 400
Tangent-Normale 401
Tangent-Space 398, 400
Tangent-Vektor 400
Tastatur siehe **Keyboard**
TCP/IP siehe **Netzwerk-Protokolle**
Team Introduction 52
Team Killer 525
Technical Director siehe **Lead Programmer**

Texturen 219
 Multitexturing 378
 samplen 373
Texturkoordinaten 719, 721, 738
Textur-Projektion 719
Tomb Raider 591
Transformation
 2D zu 3D siehe **Picking**
 3D zu 2D 282
 über Shader 370
Transformation and Lighting (TnL) 362
Transparenz 249f.
Triangulation 709
Tunnelung (Quanten) 605

U

UDP siehe **Netzwerk-Protokolle**
umgekehrte polnische Notation 885
User Datagram Protocol siehe **Netzwerk-Protokolle**
US-Patente siehe **Scene-Management**

V

Vektoren 141
 Kreuzprodukt 151
 Multiplikation mit Matrix 152
Verkaufserlöse 56
Vertex-ALU 366
Vertex-Buffer
 dynamisch 311
 statisch 311
Vertex-Programs siehe **Shader**
Vertex-Shader siehe **Shader**
Video Random Access Memory siehe **VRAM**
View-Frustum 270, 820
 Beschneidung 637
 n-sided 745
Viewport 283
virtual 74
Virtual Machine 878
 als Stack-Automat 887
virtuelle Destruktoren 84
Virtuelle Klasse siehe **Interface**
VRAM 216

W

Warteschlangen 538
Wave-Format 516
Weltkoordinatensystem 382

Windows Sockets API 534

- accept 549
- asynchrone Kommunikation 564
- bind 546
- closesocket 551
- connect 550
- Events 567
- herunterfahren 562
- initialisieren 562
- Kapselung 559
- listen 548
- Master-Socket-Objekt 542
- recv 553
- send 552
- shutdown 551
- Socket-Objekte 542

WinSock siehe **Windows Sockets API**

WordCraft-Editor siehe **Hammer-Editor**

Z

ZBL/O 878

- Compiler 892
- Syntax 894

Z-Culling, early 388

Zero Overdraw 638

ZFX 31

- Historie 32

Webseite 31

zfxCON Event 33

ZFXEngine

- VC++ Arbeitsbereich 77

ZFXEngine 2.0 65

- 3D-Mathematik 72
- Audio-Device 71
- Aufbau-Diagramm 69
- General 72
- Input-Device 71
- Netzwerk-Device 71
- Render-Device 70

ZFX_FAIL siehe **Fehlerwerte**

ZFX_OK siehe **Fehlerwerte**

ZFXEngine

Hilfsbibliothek ZFXGeneral 589

Interface

- ZFXAudioDevice 507
- ZFXInputDevice 473
- ZFXNetworkDevice 558
- ZFXRenderDevice 83, 263
- ZFXSkinManager 231
- ZFXVertexCacheManager 313

ZFXD3D 80

ZFXRenderDevice 79

ZFXRenderer 79



Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.

Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Platzierung im Internet, in Intranets, in Extranets anderen Websites, der Veränderung, des Weiterverkaufs und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.

Bei Fragen zu diesem Thema wenden Sie sich bitte an:

<mailto:info@pearson.de>

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website



(<http://www.informit.de>)

herunterladen