Ulrich Kaiser

Spieleprogrammierung in C++

Eine projektorientierte Einführung in die Programmierung von 2D-, 3D- und Netzwerkspielen mit DirectX

Galileo Computing

Inhalt

Einleitung 9

Danksagung 11

1 Vorbereitung 13

1.1 Vorkenntnisse 13

1.2 Hardware und Betriebssystem 14

1.3 Netzwerk 14

1.4 Internet 15

1.5 Entwicklungsumgebung 15

1.6 DirectX 16

1.7 Grafik-Editor 19

1.8 3D-Modellierung 20

2 2D-Projekt (Ultris) 21

2.1 Aufgabenstellung 22

2.2 Die Entwicklungsumgebung und die bereitgestellten Programme 25

2.3 Design der Oberfläche 28

2.4 Realisierung 32

2.4.1 V01 Der Windows-Rahmen 32

2.4.2 V02 Laden und Initialisieren der Sounds 43

2.4.3 V03 Laden und Initialisieren der Grafiken 52

2.4.4 V04 Definition und Implementierung der Formen 68

2.4.5 V05 Das Menü und die einfachen Dialoge 73

2.4.6 V06 Der Konfigurationsdialog 78

2.4.7 V07 Vorbesetzen des Spielfeldes 86

2.4.8 V08 Erzeugen der Formen und Anzeigen der Vorschau 92

2.4.9 V09 Fallende Formen, Geschwindigkeitsregelung und Timing 98

2.4.10 V10 Bewegen der Formen und Kollisionserkennung 106

2.4.11 V11 Auflösen der Formen und Abräumen der Reihen 112

2.4.12 V12 Highscores 116

3 Geometrische Grundlagen 125

3.1 Die Geometrie der Ebene 125

3.1.1 Koordinaten und Vektoren 125

3.1.2 Bewegungen 136

3.1.3 Projektionen 145

3.2 **Die Geometrie des Raumes 150**
3.2.1 Koordinaten und Vektoren **150**
3.2.2 Bewegungen **154**
3.2.3 Projektionen **157**

3.3 **Geometrie in DirectX 159**
3.3.1 Trigonometrische Funktionen **159**
3.3.2 Vektoren **160**
3.3.3 Vektorfunktionen **162**
3.3.4 Matrizen **165**
3.3.5 Matrizenfunktionen **168**
3.3.6 Vektor-Matrizenfunktionen **173**
3.3.7 Modellierung dreidimensionaler Objekte **176**

4 3D-Projekt (Balance) 181

4.1 **Aufgabenstellung 181**

4.2 **Die Entwicklungsumgebung und die bereitgestellten Programme 184**

4.3 **Die Bausteine des Spiels 185**

4.4 **Realisierung 192**
4.4.1 V01 Der Windows-Rahmen **193**
4.4.2 V02 Initialisieren und Darstellen des Spielfeldes **196**
4.4.2.1 Die Klasse directx **197**
4.4.2.2 Die Klasse objekt **204**
4.4.2.3 Die Klasse objekte **207**
4.4.2.4 Die Klasse spielfeld **208**
4.4.2.5 Die Klasse balance **212**
4.4.2.6 Einbau in den Windows-Rahmen **220**
4.4.2.7 Die Klasse timer **222**
4.4.3 V03 Laden und Speichern von Spielfeldern und der Spielfeld-Editor **223**
4.4.4 V04 Die Steuerung des Spiels **234**
4.4.4.1 Gamecontroller **235**
4.4.4.2 Tastatur **248**
4.4.4.3 Maus **251**
4.4.5 V05 2D- und 3D-Textausgaben **253**
4.4.5.1 2D-Text **253**
4.4.5.2 3D-Text **258**
4.4.5.3 Einbau einer Textkonsole **264**
4.4.6 V06 Positionierung und Ausrichtung der Kamera **266**
4.4.7 V06a Sprites **275**
4.4.8 V07 Drehen und Kippen des Spielfeldes **282**
4.4.9 V08 Die Kugel rollt **296**
4.4.10 V09 Kameraführung **301**
4.4.11 V10 Kollisionen **309**

4.4.12 V11 Farben, Licht und Schatten **321**
4.4.12.1 Farben **321**
4.4.12.2 Das Flexible Vertex Format **325**
4.4.12.3 Licht und Lichtquellen **328**
4.4.12.4 Materialeigenschaften **336**
4.4.12.5 Integration in die Benutzerschnittstelle **339**
4.4.13 V11a Partikelsysteme **344**
4.4.14 V12 Zeitkontrolle **352**

5 Netzwerkprojekt (Duell) 361

5.1 Aufgabenstellung 362

5.2 Asynchronität und verteilte Systeme 369

5.3 Die Entwicklungsumgebung und die bereitgestellten Programme 376

5.4 Realisierung 377
5.4.1 V01 Der Windows-Rahmen **377**
5.4.1.1 Common **377**
5.4.1.2 Server **378**
5.4.1.3 Client **378**
5.4.2 V02 Die Hauptdialoge **379**
5.4.2.1 Common **379**
5.4.2.2 Server **379**
5.4.2.3 Client **385**
5.4.3 V03 Erste Kontaktaufnahme **389**
5.4.3.1 Common **389**
5.4.3.2 Server **390**
5.4.3.3 Client **407**
5.4.4 V04 Anmeldung **423**
5.4.4.1 Client **423**
5.4.4.2 Common **427**
5.4.4.3 Server **428**
5.4.5 V05 Anmeldebestätigung **437**
5.4.5.1 Common **437**
5.4.5.2 Server **437**
5.4.5.3 Client **442**
5.4.6 V06 Einseitiger Verbindungsabbruch **446**
5.4.6.1 Common **447**
5.4.6.2 Server **447**
5.4.6.3 Client **449**
5.4.7 V07 Chat **452**
5.4.7.1 Common **452**
5.4.7.2 Server **452**
5.4.7.3 Client **456**

5.4.8 V08 Stimmübertragung **460**
5.4.8.1 Common **463**
5.4.8.2 Server **463**
5.4.8.3 Client **466**
5.4.9 V09 Balance integrieren **474**
5.4.9.1 Common **475**
5.4.9.2 Server **475**
5.4.9.3 Client **475**
5.4.10 V10 Spiel starten und beenden **483**
5.4.10.1 Common **483**
5.4.10.2 Server **485**
5.4.10.3 Client **492**
5.4.11 V11 Client-Server-Aktualisierung **498**
5.4.11.1 Common **498**
5.4.11.2 Server **499**
5.4.11.3 Client **507**
5.4.12 V12 Serverseitige Spielsteuerung **517**
5.4.12.1 Common **517**
5.4.12.2 Client **518**
5.4.12.3 Server **518**

6 Epilog 531

Index 535

Einleitung

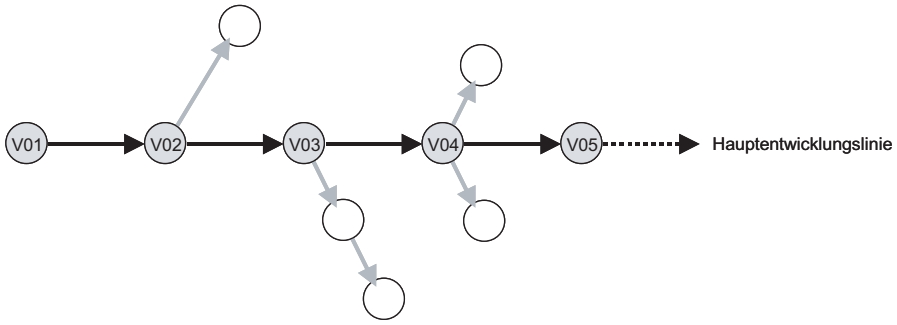
Als mein Sohn Peter zwölf Jahre alt war, fragte ich ihn, ob er nicht Interesse am Programmieren hätte. Er sah nur kurz von seinem Computerspiel – ich glaube, es war ein Autorennen – auf und sagte: »Klar, tolle Idee, lass uns gleich morgen damit anfangen! Als erstes erklärst du mir dann, wie man dieses Spiel hier macht!« Da stand ich nun mit mehr als 20 Jahren Berufspraxis in der Computerbranche auf dem Buckel und hatte noch nie ein Spiel programmiert. Von den wirklich wichtigen Aspekten des Programmierens verstand ich offensichtlich nichts. Um in den Augen meines Kindes nicht als Versager dazustehen, habe ich am nächsten Tag das *DirectX Software Development Kit* aus dem Internet heruntergeladen und mit der Spieleprogrammierung begonnen. Inzwischen sind aus diesem Impuls mehrere Spiele, eine Vorlesung, ein Praktikum, ein Skript und jetzt auch dieses Buch hervorgegangen.

Damit Sie den größtmöglichen Gewinn aus diesem Buch ziehen, möchte ich Ihnen einige Hinweise zum Vorgehen geben.

Ich habe mich für ein projektorientiertes Vorgehen entschieden. Das heißt, dass im Buch drei vollständige Entwicklungsprojekte durchgeführt werden:

- 2D-Spiel *Ultris*
- 3D-Spiel *Balance*
- Netzwerkspiel *Duell*

Am Ende eines jeden Projekts steht als Ergebnis ein vollständiges Computerspiel. Die Projekte werden in einzelnen, aufeinander aufbauenden Entwicklungsstufen Schritt für Schritt durchgeführt. Ich empfehle Ihnen, dieser schrittweisen Vorgehensweise zu folgen und Ihre persönliche Kreativität in der Lernphase zunächst noch im Zaum zu halten. Wenn Sie zu sehr von den Vorgaben abweichen, wird es für Sie sehr aufwändig sein, sich in den nachfolgenden Entwicklungsstufen wieder mit den Inhalten des Buchs zu »synchronisieren«. Das soll natürlich nicht heißen, dass Sie den Quellcode aus dem Buch nur abschreiben sollen. Dabei lernen Sie nichts. Ich ermuntere Sie ausdrücklich, auf jeder Stufe zu experimentieren und eigene Ideen auszuprobieren. Die Entwicklungsstufen des Buches geben dabei die Hauptentwicklungslinie vor. Sie können jederzeit Varianten mit eigenen Ideen realisieren. Auf keinen Fall sollten Sie aber eine eigene Entwicklungslinie eröffnen, diese über mehrere Stufen weiterentwickeln und dann versuchen, wieder in die Hauptentwicklungslinie zurückzukehren. Eigenentwicklung sollten Sie immer nur innerhalb einer Stufe machen und am Ende jeder Stufe wieder zur Hauptentwicklungslinie zurückkehren:



Damit das problemlos geht, habe ich Ihnen die Ergebnisse jeder Projektstufe als vollständiges Projektverzeichnis auf der CD zur Verfügung gestellt. Die Bereitstellung des vollständigen Quellcodes verleitet allerdings dazu, anstelle mühseliger Tipparbeit den Code einfach zu kopieren. Sie sollten sich darüber im Klaren sein, dass Sie Ihren Lernerfolg dadurch drastisch mindern. Der Lernerfolg besteht nicht darin, den Code zu besitzen, sondern darin, ihn in Besitz zu nehmen oder zu erwerben.[1] Sicherlich haben Sie schon die Erfahrung gemacht, dass man eine Strecke in einer unbekannten Stadt selbst gefahren oder gegangen sein muss, um das Ziel später selbstständig wiederzufinden. Nur in einem Taxi auf dem Rücksitz gesessen zu haben, reicht dafür nicht aus.

Leider eignet sich das Buch über weite Strecken nicht zum Lesen auf dem Sofa oder im Bett, da Sie parallel zur Lektüre immer am Quellcode Ihres Projekts arbeiten sollten. Das Buch nimmt direkten Bezug auf den Quellcode der Projekte, wobei aus Platzgründen im Buch immer nur der gerade relevante Ausschnitt des Quellcodes gezeigt werden kann. Damit Sie begleitend zum Lesen jederzeit auf den Quellcode Zugreifen können und das Projekt als Ganzes nicht aus den Augen verlieren, gehört dieses Buch neben den Rechner. Um Ihnen die Orientierung im Quellcode zu erleichtern, habe ich im Quelltext Markierungen in Form von Kommentaren angebracht, sodass Sie den für eine bestimmte Projektstufe relevanten Code leicht finden können.

Nehmen Sie sich bei der Umsetzung der einzelnen Stufen Zeit. Gehen Sie erst dann zur nächsten Stufe, wenn Sie die vorherige Stufe vollständig verstanden haben. Manch geometrische Sachverhalte muss man auch mal aufzeichnen oder mit Bleistiften und sonstigen verfügbaren Utensilien auf dem Schreibtisch nachbauen, um sie wirklich zu verstehen. Vielleicht muss man auch mal in einem Mathematikbuch oder einer Formelsammlung nachschlagen, um Kenntnisse aus der Geo-

1 Was du ererbt von deinen Vätern, erwirb es, um es zu besitzen!
(Johann Wolfgang von Goethe)

metrie oder der Vektorrechnung aufzufrischen. Nutzen Sie intensiv das Hilfesystem zu *DirectX*, um sich zusätzliche Informationen zu besorgen.

Alle von mir erstellten Quellen aus diesem Buch und von der CD können Sie frei verwenden, um Ihre eigenen Spiele zu erstellen.

Ansonsten wünsche ich, dass Ihnen die Spieleprogrammierung mit diesem Buch so viel Spaß macht wie mir das Schreiben dieses Buches.

Bocholt, im September 2002

Ulrich Kaiser

P.S.: Peter erstellt inzwischen seine eigenen Computerspiele und behauptet, ich sei nur noch der zweitbeste Programmierer in unserer Familie.

Danksagung

Ich danke dem Galileo-Verlag, der die Idee, ein solches Buch herauszubringen, bereitwillig aufgegriffen hat und bei der Realisierung meinen Vorstellungen und Wünschen sehr entgegengekommen ist. Namentlich erwähnen möchte ich in diesem Zusammenhang meine Lektorin, Frau Judith Stevens-Lemoine, die mich bereits zum zweiten Mal perfekt betreut hat.

Ich danke meinen Studenten Christoph Kecher und Marco Schlüter, die mit zahlreichen Fehlermeldungen und Verbesserungsvorschlägen dazu beigetragen haben, mein Vorlesungsskript zur Spieleprogrammierung zu einem Buch auszugestalten.

Besonderen Dank schulde ich aber meinem Entwicklungs- und Test-Team: Thorsten Humberg, Klaus Kaiser, Peter Kaiser und Moritz Rasche. Sie haben mit großem Engagement die von mit erstellten Programme getestet und auf der Basis meines Manuskripts mit Erfolg eigene Spiele erstellt. Die praktischen Erfahrungen, die sie als Einsteiger in die Spieleprogrammierung dabei gemacht haben, sind in die Spieleprojekte dieses Buchs eingegangen und haben die Qualität des Endprodukts deutlich verbessert.

1 Vorbereitung

Auf keinen Fall sollten Sie sich unvorbereitet in ein Abenteuer, wie wir es hier vorhaben, stürzen. Am Anfang steht daher die Frage: Was benötigen Sie an Vorkenntnissen und an Hard- und Software, um die Projekte aus diesem Buch erfolgreich durchführen zu können?

1.1 Vorkenntnisse

Die wichtigsten Voraussetzungen sind natürlich Ihre persönlichen Vorkenntnisse und die Erfahrungen, die Sie mitbringen. Voraussetzung für die Durchführung der Projekte ist, dass Sie in C und C++ programmieren können. Gute C- und darüber hinaus elementare C++-Kentnisse reichen allerdings aus, da keine speziellen objektorientierten Techniken, wie zum Beispiel Vererbung, verwendet werden. Von C++ benutze ich nur das Klassenkonzept, um Daten und Funktionen zu kapseln. Spezielle Kenntnisse in der Windows-Programmierung (*Windows-SDK*, *Microsoft Foundation Classes*) sind hilfreich, werden aber nicht unbedingt benötigt. Zwar setzt die Programmierung auf dem *Windows-SDK*[1] auf, aber ich werde alle hier verwendeten Windows-Funktionen so erklären, dass keine speziellen Vorkenntnisse zu deren Verständnis erforderlich sind. Dieses Buch enthält damit sozusagen als Nebenprodukt einen Grundkurs in Windows-Programmierung.[2]

Sollten Sie feststellen, dass Ihre Programmierkenntnisse nicht ausreichen, so sollten Sie dieses Buch zunächst einmal zur Seite legen, um sich mit C/C++-Programmierung zu beschäftigen. Dazu empfehle ich Ihnen mein ebenfalls bei Galileo Computing erschienenes Buch:

1 Das ist die Programmierschnittstelle von Windows.

2 Wobei ich Ihnen empfehlen würde, dass Sie, wenn Sie vertieftes Interesse an Windows-Programmierung haben, nicht auf der Ebene des SDK, sondern auf der Ebene der MFC (Microsoft Foundation Classes) zu arbeiten.

Grundsätzlich gilt natürlich: Je mehr praktische Programmiererfahrung Sie mitbringen, umso leichter wird Ihnen der Einstieg in die Spieleprogrammierung fallen.

Hier finden Sie das programmiertechnische Rüstzeug, um die Projekte dieses Buches erfolgreich durchführen zu können.

1.2 Hardware und Betriebssystem

2D-Spiele sind, was den Ressourcenbedarf auf einem Rechner betrifft, in der Regel nicht sehr anspruchsvoll. Solche Spiele laufen auch auf einfacher, nicht besonders hochgerüsteter Hardware. Viele 2D-Spieleklassiker, wie zum Beispiel *Pacman*, liefen ja schon in der »Steinzeit« der Spieleprogrammierung auf Rechnern, die im Vergleich zu den heutigen Rennpferden eher wie eine Schnecke daherkrochen. In diesem Bereich gibt es keine besonderen Anforderungen. Ein einfacher Pentium mit »gewöhnlicher« Grafik- und Soundkarte ist hier mehr als ausreichend.

3D-Spiele stellen dagegen erheblich höhere Anforderungen an die Leistungsfähigkeit eines Rechners. Insbesondere sind dies Anforderungen an die Grafikkarte des Systems. Wie Sie feststellen können, ob Ihr System den hier gestellten Anforderungen genügt, erfahren Sie in Abschnitt 1.6 über *DirectX*.

Im Prinzip können Sie die Spiele aller drei Projekte mit der Tastatur spielen. Sinnvollerweise sollten Sie aber für die 3D-Spiele ein Gamepad oder einen Joystick haben.

Als Betriebssystem dient in allen Projekten Microsoft Windows; ob 98, ME, 2000 oder XP, das macht keinen Unterschied. Windows 95 und Windows NT kommen jedoch nicht in Frage.

1.3 Netzwerk

Für die ersten beiden Projekte benötigen Sie kein Netzwerk. Für das dritte Projekt sollten Sie zumindest zwei vernetzte Rechner haben, die jeder für sich von der Hardware her den Anforderungen des zweiten Projekts genügen. Nur einen der beiden Rechner benötigen Sie als Entwicklungsrechner. Den anderen benötigen Sie nur zeitweise zum Spielen oder Testen. Von Vorteil wäre, wenn Sie einen dritten Rechner hätten, um ihn als Server einzusetzen. Den Server benötigen Sie nicht unbedingt, da Sie die Serverfunktionen auch auf einem der Spielsysteme implementieren können. Der Server muss auch in Bezug auf die Grafikleistung nicht besonders ausgelegt sein. Im Wesentlichen übernimmt der Server Aufgaben im Bereich der Kommunikation. Auch hier sind die Anforderungen nicht sehr hoch, da das Transfervolumen für die Kommunikation minimal ist. Wenn Sie allerdings zusätzlich zum normalen Spielbetrieb die Sprachübertragung nutzen wollen, so benötigen Sie einen Netzwerkzugang mit größerer Bandbreite und ein Headset für jeden Spieler.

Je mehr Rechner Sie zum Testen zur Verfügung haben, umso besser. Vielleicht veranstalten Sie während des dritten Projektes gelegentlich eine LAN-Party, um Ihre Software in angemessenem Rahmen zu testen.

1.4 Internet

Wenn Sie nicht nur im lokalen Netz, sondern auch über das Internet spielen wollen, so müssen natürlich alle am Spiel beteiligten Rechner einen Internet-Zugang haben. Darüber hinaus benötigen Sie zumindest für den Server (beziehungsweise für den Rechner, der als Server fungiert) eine IP-Adresse, die aus dem Internet angesprochen werden kann. Am besten ist natürlich eine feste IP-Adresse, die Sie dauerhaft verwenden können. Aber auch eine temporäre IP-Adresse, wie Sie sie üblicherweise von Ihrem Provider bekommen, reicht aus. Sie müssen nur dafür sorgen, dass alle Teilnehmer am Spiel die jeweils gültige Adresse kennen. Nicht verwenden können Sie eine dynamisch im lokalen Netz vergebene Adresse, die maskiert wird und von außen unsichtbar ist.

Ein Internet-Zugang ist darüber hinaus in jedem Fall sinnvoll, auch wenn Sie nicht über das Internet spielen wollen. Das Internet ist eine Fundgrube für Spiele-Entwickler. Sie finden dort Diskussionsgruppen, Tutorials, Code-Beispiele, 3D-Modelle, Texturen und viele nützliche Hilfsprogramme. Da Verweise auf Internetseiten oft sehr kurzlebig sind, habe ich hier nur selten konkrete Links angeführt. Mit einer guten Suchmaschine (*www.google.de*) können Sie sich aber schnell eigene, an Ihre Interessen angepasste Links zusammenstellen. Eine wichtige Startadresse ist sicherlich das Microsoft Developer Network (*msdn.microsoft.com*). Dort erhalten Sie detaillierte Informationen über alle hier zum Einsatz kommenden Microsoft-Produkte.

An der Fachhochschule in Bocholt (*www-et.bocholt.fh-gelsenkirchen.de*[3]) entsteht zurzeit ein Internetangebot, das sich speziell an Spieleprogrammierer und solche, die es werden wollen, richtet. Sie können dort die im Rahmen meiner Vorlesung *Spieleprogrammierung* von Studenten erstellten Spiele im Quellcode herunterladen. Sie werden dort aber auch die Möglichkeit haben, mit anderen Entwicklern zu diskutieren, eigene Spiel-Ideen zu präsentieren, Kontakte zu knüpfen oder Gleichgesinnte für eigene Entwicklungsprojekte zu finden. Besonders freuen würde ich mich, wenn Sie dort eigene Spiele zum Download anbieten.

1.5 Entwicklungsumgebung

Als Entwicklungsumgebung habe ich *Microsoft Visual Studio* und den *Microsoft Visual C++ Compiler* (Version 6.0) verwendet. Im Prinzip können Sie auch mit ei-

3 Leider kann ich Ihnen zur Zeit der Drucklegung dieses Buches noch nicht die endgültige Adresse nennen.

ner anderen Programmierumgebung arbeiten, sofern diese die Windows-Programmierung mit dem *Windows-SDK* und die Verwendung von Windows-Ressourcen unterstützt. Sie müssen dann nur unter Umständen einen gewissen Aufwand in Kauf nehmen, um meine Beispielprogramme aus der Microsoft-Umgebung in Ihre Umgebung zu portieren. Zur Unterstützung habe ich Ihnen die Makefiles meiner Projekte mit auf die CD gelegt. In der Regel können Sie diese Makefiles in Ihre Entwicklungsumgebung importieren und dann zusammen mit meinen Quellcodedateien Ihr eigenes Projekt erstellen.[4]

Falls Sie die Microsoft-Entwicklungsumgebung haben, müssen Sie nur das gewünschte Projektverzeichnis von der CD auf die Festplatte Ihres Rechners kopieren und gegebenenfalls noch den Schreibschutz von den Dateien entfernen. Dann müssen Sie nur noch auf den Projekt-Arbeitsbereich (`.dsw`) im kopierten Projektverzeichnis klicken, und schon kann es losgehen.

An einigen Stellen – zum Beispiel dort, wo es darum geht, Ressourcen für ein Projekt zu erstellen[5] – nehme ich in meinen Anweisungen konkret Bezug auf die Microsoft-Umgebung. In der Regel können Sie die Anweisungen leicht in eine andere Entwicklungsumgebung übertragen, auch wenn die konkrete Handhabung dort eine andere ist.

Dem Buch liegt auf der zweiten CD eine Autorenversion der Microsoft-Entwicklungsumgebung bei. Als Leser dieses Buches können Sie diese Entwicklungsumgebung für Ihre private Programmierung im Rahmen der Spieleprojekte nutzen. Informationen über die genauen Nutzungsbedingungen und die Einschränkungen dieser Version gegenüber einer Vollversion finden Sie in der zugehörigen Dokumentation.

1.6 DirectX

DirectX ist ein Sammelbegriff für die von Microsoft zur Multimedia-Programmierung bereitgestellten Hilfsmittel. Um mit *DirectX* entwickelte Anwendungen auf einem PC zu benutzen, benötigen Sie die *DirectX*-Laufzeitbibliotheken, die üblicherweise schon auf jedem PC vorhanden sind. Um aber Anwendungen mit *DirectX* zu entwickeln, benötigen Sie das *DirectX-SDK* (Software Development Kit). Dieses SDK können Sie kostenlos von Microsoft aus dem Internet herunterladen – aber Achtung, es handelt sich um über 170 MByte.[6]

4 Der *Borland C++ Builder* erlaubt sogar das direkte Importieren der Microsoft-Projektdateien.
5 Ressourcen sind die Elemente der Benutzeroberfläche, zum Beispiel Menüs oder Dialoge.
6 Vielleicht nehmen Sie es dann doch lieber von der beiliegenden CD.

Innerhalb der *DirectX*-Familie finden Sie die folgenden Komponenten:

- *DirectX Graphics* (*DirectDraw*, *Direct3D*): 2D- und 3D-Grafikprogrammierung
- *DirectXAudio* (*DirectSound*, *DirectMusic*): Abspielen und Bearbeiten von Sounddaten
- *DirectInput*: Programmierung von Eingabegeräten (z. B. Joystick)
- *DirectPlay*: Programmierung von Netzwerkspielen
- *DirectShow*: Aufnahme und Wiedergabe von Multimedia-Inhalten
- *DirectSetup*: Programmierbare Installation von *DirectX*

Bei allen Komponenten handelt es sich um Funktionsbibliotheken, die über eine C/C++-Schnittstelle programmiert werden können.

Im 2D-Projekt (*Ultris*) werden wir uns mit *DirectDraw* und *DirectSound*, im 3D-Projekt (*Balance*) mit *Direct3D* und *DirectInput* beschäftigen. Im Netzwerkprojekt (*Duell*) kommt dann *DirectPlay* hinzu.

Bezüglich der Installation der Entwicklungsumgebung und des SDK kann ich Ihnen natürlich keine Hilfestellung geben. Dafür ist der Hersteller zuständig. In der Regel ist die Installation aber problemlos. Trotzdem können bei speziellen Systemkonfigurationen schon mal Fehler auftreten. Wichtig ist, dass Sie im Falle der Microsoft-Umgebung zuerst die Entwicklungsumgebung und dann das SDK installieren, da bei der Installation des SDK die erforderlichen Änderungen der Einstellungen der Entwicklungsumgebung automatisch vorgenommen werden. Das geht natürlich nur, wenn die Entwicklungsumgebung bereits installiert ist. Wichtig ist, dass Sie auch die zugehörigen Hilfedateien installieren. Die werden Sie, sobald Sie Ihre ersten eigenen Schritte unternehmen, dringend benötigen. Wenn Sie die Entwicklungsumgebung und das *DirectX-SDK* erfolgreich installiert haben, sollten Sie testen, ob auf Ihrem Entwicklungsrechner alles zusammenspielt.

Dazu sollten Sie zunächst einmal einige der Demoprogramme, zum Beispiel aus den Verzeichnissen im Ordner `<installationspfad>\samples\Multimedia`, starten.[7]

Wenn Sie dabei eine Meldung wie die folgende sehen, sollten Sie sich schon einmal nach Preisen für Grafikkarten oder neue PCs erkundigen.

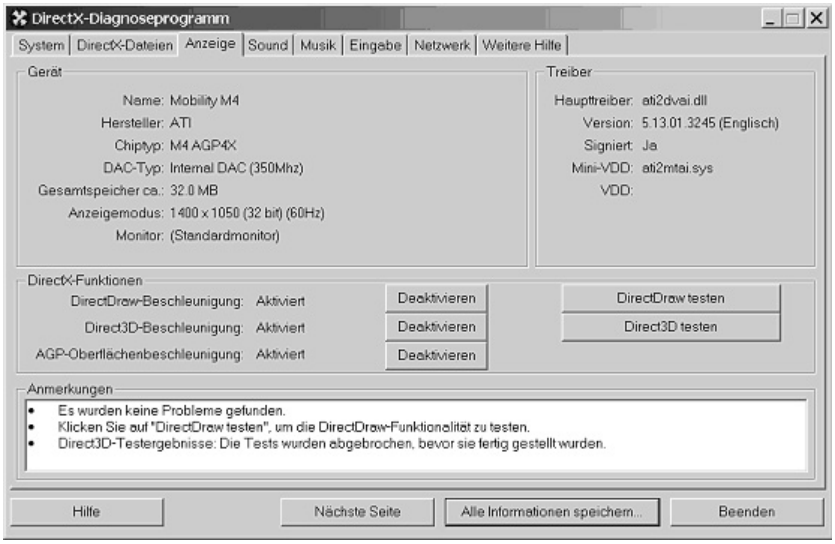
7 Lauffähige Programme finden Sie jeweils in den Unterverzeichnissen `bin`.

Anschließend testen Sie die Entwicklungsumgebung, indem Sie versuchen, einige der von Microsoft mitgelieferten Beispielprogramme (im Verzeichnis `samples\Multimedia`) zu kompilieren und zum Laufen zu bringen. Auch das sollte zumindest in der Microsoft-Umgebung kein Problem sein, da für jedes Beispiel das vollständige Projektverzeichnis vorliegt.

Zusätzlich zu den Funktionsbibliotheken wird mit *DirectX* ein Control-Panel in der Systemsteuerung installiert, über das Sie die *DirectX*-Komponenten konfigurieren können:

Hilfreich sind in diesem Zusammenhang zwei Werkzeuge, die ebenfalls mit *DirectX* geliefert werden:

- Der *CapsViewer* (`<installationspfad>\Bin\DXUtils\DXCapsViewer.exe`) zeigt Ihnen die Eigenschaften aller an Ihrem Rechner angeschlossenen Ein- und Ausgabegeräte an.
- Das *DirectX-Diagnoseprogramm*, das Sie aus dem oben gezeigten Control-Panel heraus starten können, erlaubt es, eine Reihe von Tests durchzuführen, um festzustellen, ob alles sauber funktioniert.



Wenn Ihr System die Tests erfolgreich absolviert, Sie die Beispielprogramme kompilieren und linken können und das Laufzeitverhalten der Beispielprogramme akzeptabel ist, dann können Sie davon ausgehen, dass Ihr System den hier gestellten Anforderungen genügt.

Wenn Sie Ihre Grafikkarte auf Herz und Nieren testen wollen, so können Sie aus dem Internet – zum Beispiel von *pc-extreme.de* oder *tuning-freeware.de* – Benchmarks zum Messen der Performance Ihrer Grafikkarte herunterladen.

1.7 Grafik-Editor

Zur Erstellung von 2D-Grafiken benötigen Sie einen Grafik-Editor. Neben dem Grafik-Editor *Paint-Brush*, den Sie auf jedem Windows-System finden, gibt es dutzende von Grafik-Editoren (zum Beispiel *Paint Shop Pro*) als Freeware oder Shareware. Nehmen Sie Ihren favorisierten Editor. Wichtig ist nur, dass er Dateiformate wie `bmp` oder `jpg` unterstützt und dass Sie mit ihm zurechtkommen. Alternativ

können Sie auch die von mir bereitgestellten Grafiken benutzen. Auf diese Weise können Sie allerdings kein individuelles Spiel mit Ihrer persönlichen Note erstellen.

1.8 3D-Modellierung

Für das 3D-Projekt benötigen Sie einen 3D-Modeller. Das ist ein Editor, der es erlaubt, ein dreidimensionales Drahtmodell von einer räumlichen Figur zu erstellen und dieses dann mit einer gemusterten Haut (Textur) zu überziehen. Auch 3D-Modeller gibt es als Shareware. Gute Werkzeuge sind hier aber sehr teuer. High-End-Werkzeuge, mit denen Filme wie Jurassic Park gestaltet werden, sind für den Normalverbraucher unerschwinglich. Wichtig ist, dass Sie einen 3D-Modeller haben, aus dem Sie das Drahtmodell und die Texturen in das so genannte x-Format (Dateierweiterung `.x`) entladen können. Dieses Format verwenden wir, um das Modell in *DirectX* einzulesen, damit es auf dem Bildschirm dargestellt werden kann. Den 3D-Modeller benötigen wir ja erst für unser zweites Projekt, und Sie haben genügend Zeit, Ihre Entscheidung zu treffen. Ich verwende übrigens AC3D. Bei AC3D handelt es sich um einen preiswerten Modeller, der aber über alle notwendigen Funktionen verfügt. Dazu mehr im zweiten Projekt.

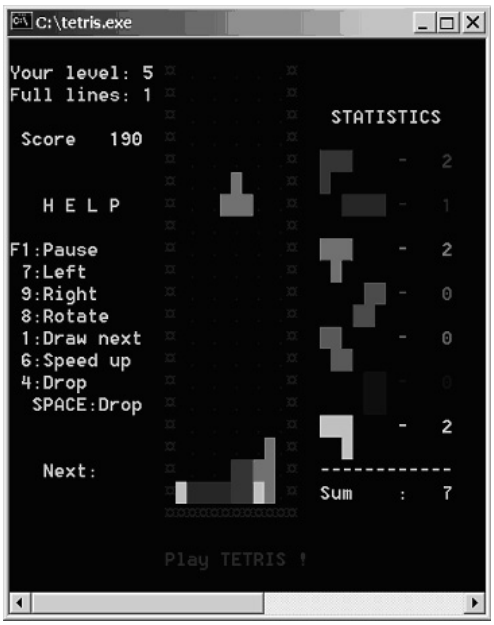
In Ermangelung eines Modellers können Sie auch meine 3D-Modelle übernehmen oder Modelle aus dem Internet herunterladen. Der Programmierung tut das keinen Abbruch. Es entgeht Ihnen nur der Spaß, ein eigenes Modell zu erstellen und Sie sind in der kreativen Umsetzung eigener Spiel-Ideen beschränkt.[8]

Das soll zum Thema Vorbereitung erst einmal reichen. Wenn Sie alles heruntergeladen, installiert, aktualisiert, ausprobiert, kompiliert, gelinkt und getestet haben, treffen wir uns auf der nächsten Seite wieder zu unserem ersten Spieleprojekt.

8 Da das x-File-Format ein lesbares Format ist, kann man theoretisch auch x-Files mit einem Texteditor erstellen. Aber diese Sisyphusarbeit sollten Sie nicht auf sich nehmen.

2 2D-Projekt (Ultris)

Auf dem Spielemarkt dominieren heute 3D-Spiele. Diese Spiele versuchen sich gegenseitig durch immer ausgefeiltere Grafiken und 3D-Effekte zu übertrumpfen. Häufig ist aber die Spielidee immer die gleiche und nur in ein anderes Szenario eingekleidet. Im Gegensatz dazu gibt es unter den 2D-Spielen viele Klassiker mit originellen Spielideen, die ganze Spielergenerationen geradezu süchtig gemacht haben. *Tetris* ist ein solches Spiel. *Tetris* wurde 1985/86 von dem russischen Informatiker Alexej Padschitnow erfunden und verbreitete sich wie ein Lauffeuer um die ganze Welt. Es wurde auf allen nur denkbaren Systemen implementiert und war das Spiel, das den Nintendo Game Boy so erfolgreich machte. Wahrscheinlich ist es das erfolgreichste Computerspiel aller Zeiten. Es ist schwer zu sagen, was dieses Spiel so erfolgreich macht. Das Spiel ist bemerkenswert einfach, was seine Regeln und seine Bedienung betrifft und zugleich bemerkenswert vielfältig, was die möglichen Spielabläufe angeht. Der Anreiz für den Spieler besteht darin, Ordnung in das Chaos der immer schneller fallenden Puzzlesteine zu bringen. Offensichtlich weckt das Spiel den in uns allen sitzenden Putzteufel. Aus nostalgischen Gründen habe ich Ihnen auf der CD im Verzeichnis *Spiele/Tetris* die Originalversion des Spiels (`tetris.exe`) zur Verfügung gestellt:

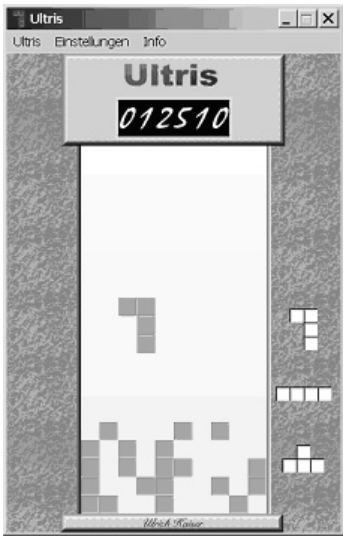


Das Original ist mit den einfachen, seinerzeit verfügbaren Mitteln realisiert und wirkt, was die Oberfläche betrifft, heute schon sehr angestaubt. Der Vergleich mit heutigen Spielen zeigt, welch rasante Entwicklung die Spieleprogrammierung und

insbesondere die Programmierung grafischer Oberflächen in den letzten 15 Jahren genommen hat. Im Laufe der Jahre hat es auch eine Reihe von Neuimplementierungen von *Tetris* gegeben, darunter auch 3D-Versionen mit teilweise erweiterten Spielideen. Keine 3D-Version konnte aber an den Erfolg des 2D-Spiels anknüpfen. Dies zeigt, dass 2D-Spiele nach wie vor eine Existenzberechtigung haben. Welche Art der Visualisierung für ein Spiel geeignet ist, hängt letztlich von der Spielidee ab, und manche Ideen können offensichtlich in zwei Dimensionen wirkungsvoller umgesetzt werden als in dreien. In unserem ersten Projekt werden wir einen vollwertigen Tetris-Klon mit dem Namen *Ultris* entwickeln. Die Oberfläche wird etwas moderner sein, als die des Originals, aber die Spielidee werden wir vom Original übernehmen, da sie nicht verbessert werden kann.

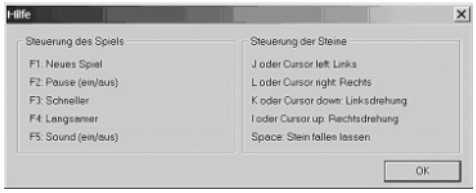
2.1 Aufgabenstellung

Natürlich kennen Sie *Tetris,* und ich muss Ihnen nicht erklären, wie man dieses Spiel spielt. Anstelle einer Spielanleitung zeige ich Ihnen hier nur ein paar spärlich kommentierte Screenshots des fertigen Spiels, das wir später Schritt für Schritt erstellen werden. Wir beginnen mit dem eigentlichen Spielfeld:



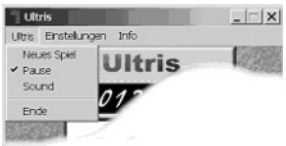
Unter der oberen Abdeckung fallen die Puzzlesteine mit wachsender Geschwindigkeit heraus. Die Aufgabe des Spielers ist es, die Steine durch Drehung und Rechts-Links-Verschiebung so einzuordnen, dass unten vollständige Reihen entstehen, die dann abgeräumt werden. Alle Aktionen werden durch entsprechende Sounds untermalt. Am rechten Spielfeldrand sehen Sie eine Vorschau auf die nächsten drei Steine. Oben auf der Abdeckung wird Ihnen Ihr aktueller Punktestand angezeigt.

Das Spiel bedienen Sie über die Tastatur:



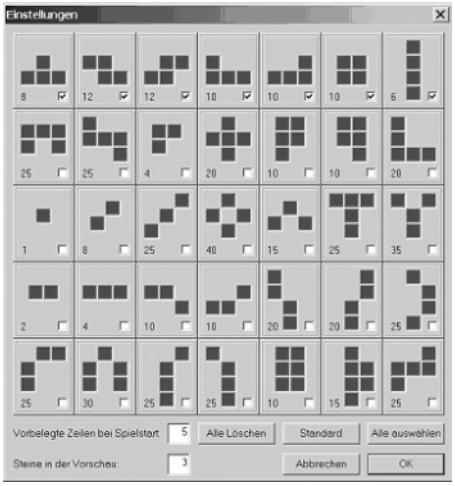
Die Tasten [J], [K], [L], [I] und [Space] erlauben eine bequeme Einhandbedienung mit der rechten Hand.

Einige der oben genannten Tastaturbefehle ([F1], [F2] und [F5]) können Sie auch aus dem *Ultris*-Menü heraus absetzen:



Alle wichtigen Spielparameter können Sie über den Menüpunkt *Konfiguration* im Menü *Einstellungen* ändern.

Bei Wahl dieses Menüpunktes erhalten Sie den folgenden Dialog:



Im Dialog können Sie wählen, mit welchen Steinen Sie spielen wollen, wie viele vorbelegte Zeilen (0–20) es beim Spielstart geben soll und wie viele Steine (0–5) in der Vorschau erscheinen sollen. Der Dialog zeigt Ihnen auch, wie viele Punkte es für die jeweiligen Steine gibt. Diese Punktwerte können Sie allerdings nicht ändern.

Schließlich gibt es noch das Menü *Info*:

In diesem Menü kann der oben bereits gezeigte Hilfedialog, der die Tastaturbelegung zeigt, aufgerufen werden. Darüber hinaus gibt es eine Highscore-Tabelle

und einen allgemeinen Informationsdialog:

Der Button *Weitere Informationen* führt Sie, sofern Ihr Computer mit dem Internet verbunden ist, auf die Homepage des Herstellers – in diesem Fall auf die Homepage der Fachhochschule in Bocholt.

Um das Spiel wirklich zu verstehen, muss man es natürlich spielen – oder besser gesagt ausgiebig testen. Das empfehle ich Ihnen an dieser Stelle, damit Sie genau wissen, was Sie implementieren sollen. Sie finden die Testversion des Spiels auf der CD im Verzeichnis *Spiele/Ultris*, das Sie vollständig auf die Festplatte Ihres Rechners kopieren sollten, um das Spiel dann von der Festplatte aus zu starten:

Um der drohenden Suchtgefahr zu entgehen, sollten Sie sich dabei aber ein striktes Zeitlimit setzen.

2.2 Die Entwicklungsumgebung und die bereitgestellten Programme

Wir werden *Ultris* in mehreren Schritten entwickeln. In jedem Schritt entsteht eine neue Version (`V01`, `V02`, ...) mit erweiterter Funktionalität. Die erste Version, `V01`, steht Ihnen als vollständiges Microsoft Visual C++-Projekt[1] auf der CD im Verzeichnis *Entwicklung/Ultris/V01* zur Verfügung. Dieses Projekt können Sie als Startpunkt für die weitere Entwicklung verwenden.[2] Am besten kopieren Sie dazu das komplette Verzeichnis *Entwicklung* von der CD auf die Festplatte Ihres Entwicklungsrechners. Auch für die weiteren Entwicklungsstufen habe ich, wie Sie feststellen werden, vollständige Lösungen erstellt, die Sie als Wiederaufsetzpunkt verwenden können, wenn Sie Versionen überspringen wollen oder einmal zu weit vom Kurs abgewichen sind. Um maximalen Gewinn aus dem Kurs zu ziehen, ist es sinnvoll, mit der Version `V01` zu starten und dann alle Folgeversionen selbst zu erstellen. Natürlich können Sie bei der Entwicklung eigene Ideen einbringen und von meinem Implementierungsvorschlag abweichen. Sie sollten jedoch immer darauf achten, nicht so weit abzuweichen, dass Ihnen die Rückkehr zu den Wiederaufsetzpunkten unmöglich wird. Sie können beispielsweise von Anfang an andere Abmessungen als die von mir vorgeschlagenen für die Spielfeldgröße verwenden, weil Ihnen ein anderes Layout besser gefällt. Sie müssen sich nur darüber im Klaren sein, dass Sie dann in allen nachfolgenden Berechnungen andere Bezugspunkte und Koordinatenwerte verwenden müssen. Sollte dann einmal die Notwendigkeit bestehen, Quellcode aus meiner Musterlösung zu übernehmen, müssen Sie entsprechende Anpassungen vornehmen. Problematisch wird es, wenn Sie strukturell, also in Datenstrukturen oder Funktionsschnittstellen, von meiner Vorlage abweichen. Insbesondere, wenn Sie noch unsicher in der C-Programmierung sind, sollten Sie das Streben nach Kreativität zunächst noch hinten

1 Erstellt mit Version 6.0

2 Ich wollte Sie nicht damit belasten, was (z. B. welche Libraries) Sie alles zusammenfügen müssen, um ein solches Projekt zu erstellen. Wenn Sie noch unerfahren in der Microsoft-Programmierumgebung oder in der Erstellung komplexerer Projekte sind, würde Sie das an dieser Stelle nur verwirren. Wenn Sie hingegen bereits erfahren sind, können Sie diese Informationen problemlos aus dem Projekt herauslesen, indem Sie in den entsprechenden Einstellungsdialogen nachsehen oder sich ein Makefile erzeugen lassen.

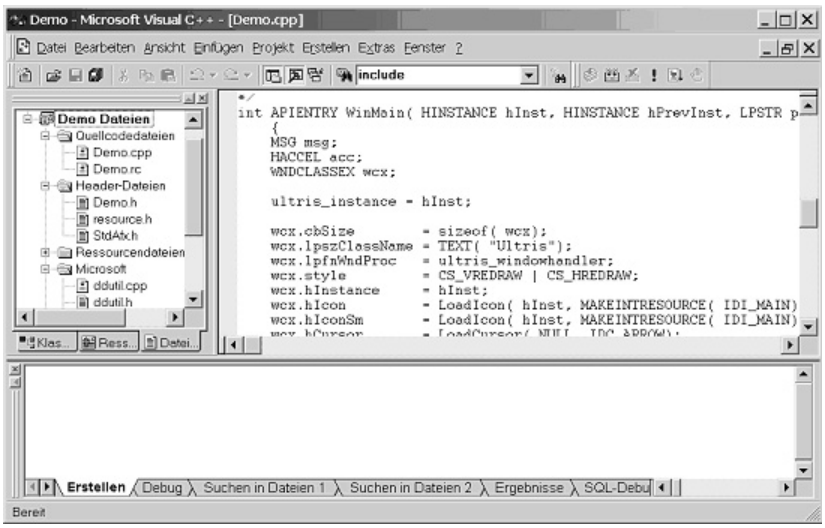
anstellen und möglichst detailgenau meinen Vorgaben folgen. Wenn Sie die Spieleprogrammierung dann beherrschen, haben Sie noch ausreichend Gelegenheit, Ihre Kreativität in selbst gestellten Aufgaben unter Beweis zu stellen.

Falls Sie nicht mit der Microsoft-Entwicklungsumgebung arbeiten, können Sie, sofern Ihre Entwicklungsumgebung es erlaubt, das Microsoft-Projekt (*Demo.dsp*) importieren oder mit den im Verzeichnis *Entwicklung/Make* bereitgestellten Quellen und dem Makefile (*Demo.mak*) Ihr eigenes Projekt aufsetzen. Insbesondere finden Sie im Makefile die wichtige Information, welche Libraries Sie für dieses Projekt benötigen:

```
winmm.lib dxguid.lib dxerr8.lib ddraw.lib kernel32.lib
user32.lib gdi32.lib winspool.lib comdlg32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib dsound.lib advapi32.lib
```

Diese Information benötigen Sie aber nur dann, wenn Sie Ihr Projekt individuell erstellen wollen oder müssen. Haben Sie bitte Verständnis dafür, dass ich Sie bei konkreten Problemen mit Ihrer Entwicklungsumgebung nicht weiter unterstützen kann. Konsultieren Sie in diesem Fall die Internetseiten des Herstellers oder entsprechende Diskussionsgruppen im Internet.

Wenn Sie mit der Microsoft-Entwicklungsumgebung arbeiten wollen, mit dieser zuvor aber noch nicht in Berührung gekommen sind, sollten Sie sich die Zeit nehmen, mit der Umgebung vertraut zu werden. Sie starten die Umgebung am besten durch einen Doppelklick auf die Projektdatei demo.dsw im Verzeichnis der Version V01. Sie sollten dann in etwa das folgende Bild sehen:

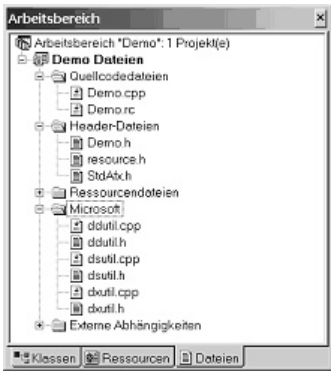


Nach dem Start zeigt die Entwicklungsumgebung drei Fenster. Das Fenster im linken oberen Teil ist der Arbeitsbereich. Der **Arbeitsbereich** zeigt die Struktur des Projekts gegliedert in **Klassen**, Ressourcen und **Dateien**. Ein Doppelklick auf ein Objekt in diesem Fenster (zum Beispiel auf eine Datei oder eine Klasse) öffnet das Objekt zur Bearbeitung im **Editor-Fenster**. Das ist das Fenster oben rechts.

Bei einem Klick auf das rote Ausrufezeichen im Toolbar unter der Menüleiste wird Ihr Programm, sofern es geändert wurde, zunächst kompiliert und anschließend, sofern es fehlerfrei kompiliert werden konnte, gelinkt und nach fehlerfreiem Linken gestartet. Werden beim Übersetzen Fehler entdeckt, so werden diese in dem dritten Fenster, dem **Ausgabefenster**, angezeigt.

Die volle Funktionalität der Entwicklungsumgebung kann ich Ihnen an dieser Stelle und auch im weiteren Verlauf dieses Kurses nicht erklären.[3] Bei Bedarf sollten Sie die Dokumentation der Entwicklungsumgebung zu Rate ziehen. Besonders wichtig ist, dass Sie sich dabei auch in die Bedienung des Debuggers einarbeiten. Der Debugger dient eigentlich zur Fehlersuche. Sie können ihn aber auch verwenden, um meine Programme Schritt für Schritt zu durchlaufen, um so ein besseres Verständnis des Programmablaufs zu gewinnen.

Wir werfen einen Blick auf die Dateien im Arbeitsbereich des Projekts `V01`:



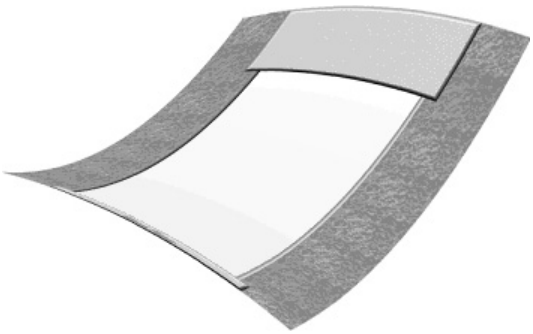
Im Ordner *Microsoft* befinden sich einige Header- und Quellcodedateien, die Teil der *DirectX*-Auslieferung von Microsoft sind. Es handelt sich um Hilfsprogramme, die die Verwendung von *DirectDraw* und *DirectSound* durch spezielle Anwendungsklassen etwas komfortabler gestalten. Ich werde diese Klassen und ihre Schnittstellen kurz besprechen, sobald wir sie benötigen.

3 Wenn Sie ein Kochbuch kaufen, wird dort ja auch nicht erklärt, wie Sie Ihren Herd bedienen müssen. Diese Informationen bekommen Sie aus der Bedienungsanleitung des Herstellers.

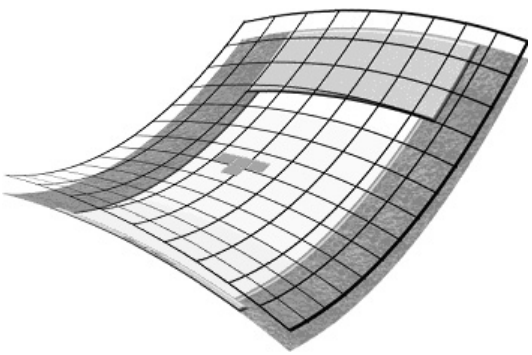
Wichtig sind für uns die Dateien *Demo.cpp* und *Demo.rc*. Die Datei *Demo.cpp* enthält in der Version `V01` den Quellcode für eine einfache Windows-Applikation. Diesen Code werden wir in den nachfolgenden Versionen erweitern, bis das Ultris-Spiel fertig ist. Die Datei *Demo.rc* ist eine so genannte Ressourcen-Datei und enthält die Oberflächenelemente unserer Windows-Applikation. Auch diese Datei werden wir von Version zu Version weiterentwickeln. Den Inhalt der beiden Dateien werden wir später ausführlich diskutieren. Zuvor beschäftigen wir uns mit dem Design der Oberfläche unseres Spiels.

2.3 Design der Oberfläche

Bevor wir mit der Implementierung des Spiels beginnen, wollen wir eine präzise Vorstellung von seiner Oberfläche entwickeln. Die Oberfläche besteht aus vier Ebenen. Die vier Ebenen werden wie Folien aufeinander gelegt und ergeben so das Gesamtbild. Die unterste Ebene ist der Spielhintergrund:

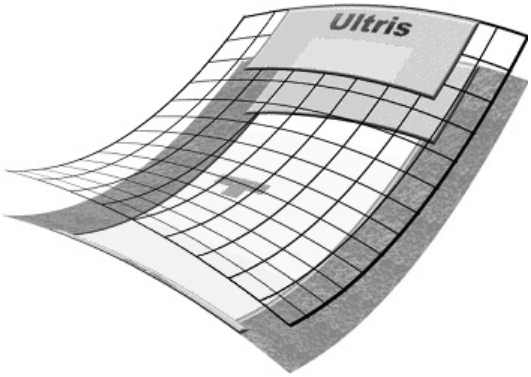


Auf der nächsten Ebene bewegen sich die Spielsteine über den Hintergrund. Wichtig ist dabei, dass die Steine in einem ganz bestimmten, allerdings unsichtbaren Raster bewegt werden:

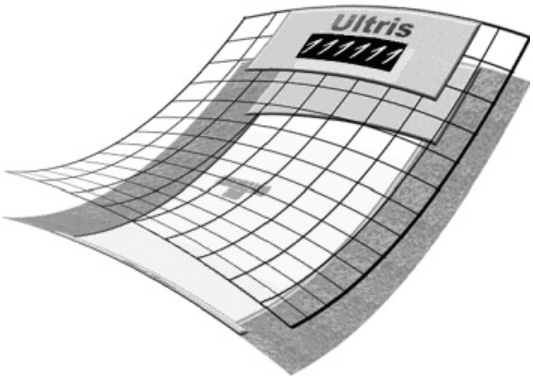


Auch die Steine der Vorschau werden am rechten Rand dieser Ebene angezeigt.

Im oberen Bereich des Spielfeldes werden die Spielsteine durch einen Deckel abgedeckt:

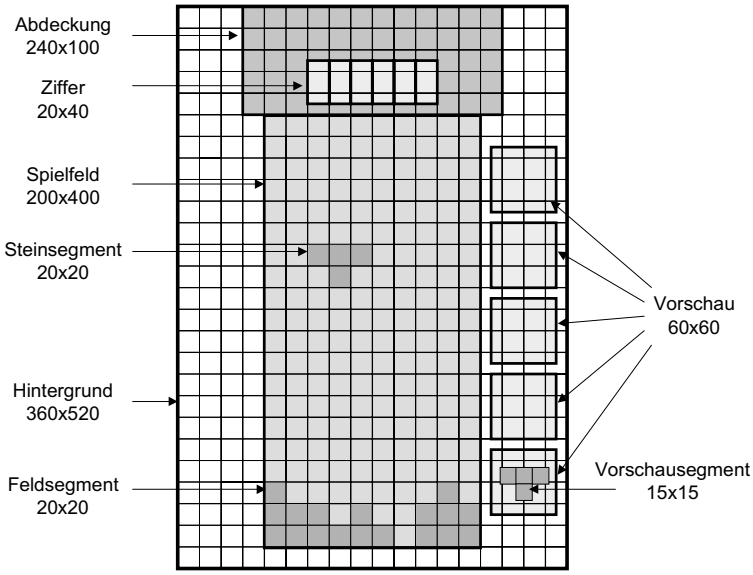


Auf dem Deckel wird schließlich in der vierten und letzten Ebene der aktuelle Punktestand angezeigt:



Die konkret verwendeten Grafiken und Sounds sollen bei unserem Programm austauschbar sein. Wir werden die Grafiken daher im Bitmap-Format (`.bmp`-Dateien) und die Sounds als Wave-Dateien (`.wav`-Dateien) auf der Festplatte ablegen und zur Laufzeit in unser Programm einlesen. Ein im Umgang mit Grafik- oder Soundwerkzeugen geübter Spieler kann dann eigene Grafiken und Sounds erstellen und so dem Spiel seine persönliche Note verleihen.

Ich hatte oben bereits erwähnt, dass wir bei der Oberflächengestaltung ein bestimmtes Raster hinterlegen müssen, in das wir alle Oberflächenelemente einpassen müssen. Die Festlegung eines solchen Rasters ist der entscheidende Designschritt. Im Wesentlichen werden wir die Oberfläche auf einem Raster von 20x20 Pixeln in einer Gesamtgröße von 360x520 Pixeln aufbauen. Die folgende Skizze zeigt das Layout des Spielfeldes:



Die Bildschirm- und Fensterkoordinaten werden, anders als Sie es von der Mathematik her kennen, gezählt. Man beginnt in der linken oberen Ecke und zählt dann die Pixel (Bildpunkte) nach rechts (x-Richtung) und nach unten (y-Richtung). Damit ergeben sich für unsere Oberflächenelemente (in Pixeln gemessen) die folgenden Bildschirmkoordinaten:

Oberflächenelement	**linke obere Ecke**			
	x	**y**	**Breite**	**Höhe**
Hintergrund	0	0	360	520
Abdeckung	60	0	240	100
Ziffern	120 140 160 180 200 220	50	20	40
Vorschau	290	130 200 270 340 410	60	60

Oberflächenelement	linke obere Ecke			
	x	y	Breite	Höhe
Steinsegment	–	–	20	20
Feldsegment	–	–	20	20
Vorschausegment	–	–	15	15

Auf diese Daten werden wir später bei der Programmierung zurückgreifen.

Bei der Erstellung eigener Grafiken sollten Sie sich an die in der folgenden Tabelle angegebenen Dateinamen und Abmessungen halten:

Dateiname	Inhalt	Abmessung der Grafik in Pixeln
`ul_hgrnd.bmp`	Spielhintergrund	`360x520`
`ul_adeck.bmp`	Die Abdeckung im oberen Bereich des Spielfeldes	`240x100`
`ul_z0.bmp –ul_z9.bmp`	Die Ziffern 0-9	`20x40`
`ul_stein.bmp`	Ein Segment des fallenden Steins	`20x20`
`ul_feld.bmp`	Segment im Feld	`20x20`
`ul_prev.bmp`	Steinsegment in der Vorschau	`15x15`

Um eigene Sounds zu verwenden, müssen Sie nur die Sound-Dateien von Ultris (`wav`-Dateien) durch eigene Sound-Dateien ersetzen. Im Einzelnen verwendet Ultris die folgenden Sound-Dateien:

Dateiname	Inhalt
`ul_start.wav`	Sound beim Start eines neuen Spiels
`ul_dreh.wav`	Sound beim Drehen eines Steins
`ul_move.wav`	Sound beim Bewegen (links/rechts) eines Steins
`ul_down.wav`	Sound beim Aufprall eines Steins
`ul_row1.wav`	Sound beim Abräumen einer Reihe
`ul_row2.wav`	Sound beim Abräumen von zwei oder mehr Reihen
`ul_ende.wav`	Sound beim Ende eines Spiels
`ul_win.wav`	Sound bei Erreichen eines Highscores

Zur Entspannung können Sie jetzt eigene Grafiken und Sounds für das Spiel erstellen. Dazu kopieren Sie erneut das Verzeichnis *Spiele/Ultris* von der CD auf die Festplatte Ihres Rechners und modifizieren oder ersetzen dann ausgesuchte Bitmap- oder Sounddateien. Sie werden sehen, dass Sie auf diese Weise ein zumindest optisch völlig neues Spiel erstellen können. Funktionell bleibt natürlich alles beim Alten.

2.4 Realisierung

Auch wenn durch Änderung der Grafiken im letzten Abschnitt ein »neues« Spiel entstehen kann, kann man das, was wir bisher gemacht haben, nicht als Spieleprogrammierung bezeichnen. Das war im weitesten Sinne noch die Anpassung oder Konfiguration eines bestehenden Spiels. In diesem Abschnitt geht es aber mit richtiger Entwicklung, das heißt mit der Programmierung, los.

2.4.1 V01 Der Windows-Rahmen

Ein Spiel auf einem Windows-PC ist natürlich auch eine Windows-Applikation und muss sich als solche in die vom Windows-Betriebssystem vorgegebenen Rahmenbedingungen einfügen. Früher, als Spiele noch unter dem Betriebssystem DOS erstellt wurden, bemächtigte sich ein Spiel einfach des gesamten Systems, um es optimal für seine Zwecke zu nutzen. Dass das unter Windows so nicht gehen kann, sehen Sie sofort, wenn Sie sich vorstellen, dass Ihr Spiel nicht im Fullscreen-Modus, sondern nur in einem Fenster laufen soll. Ihr Spiel muss dann akzeptieren, dass es nur eines von mehreren Programmen auf dem Rechner ist, und es muss sich entsprechend kooperativ verhalten. Für uns heißt das, dass wir zunächst einmal die elementaren Spielregeln der Windows-Programmierung erkennen und dann einen Windows-Rahmen erstellen müssen, in dem unser Spiel später laufen wird. Da die Windows-Programmierung aber nicht das eigentliche Anliegen dieses Kurses ist, werden wir uns hier auf das Notwendigste beschränken.

Ein wesentliches Merkmal von Windows-Programmen ist, dass sie mit anderen Programmen oder dem Betriebssystem **Botschaften** (Messages) austauschen. Wichtig für das Verständnis ist dabei, dass der Auslöser beziehungsweise der Grund für eine Botschaft Ihrem Programm in der Regel nicht bekannt ist, weil er sich außerhalb des Wahrnehmungshorizontes Ihres Programms befindet. Das klingt komplizierter, als es ist. Stellen Sie sich vor, dass das Fenster Ihres Programms durch das Fenster eines anderen Programms teilweise oder ganz verdeckt ist. Durch eine Benutzeraktion wird jetzt das andere Programm geschlossen, und Ihr Programm muss daraufhin seinen Fensterinhalt, der ja wieder sichtbar geworden ist, neu zeichnen. Ihr Programm bekommt in dieser Situation eine Botschaft des Betriebssystems, die sinngemäß lautet: »Bitte zeichne sofort die linke Hälfte

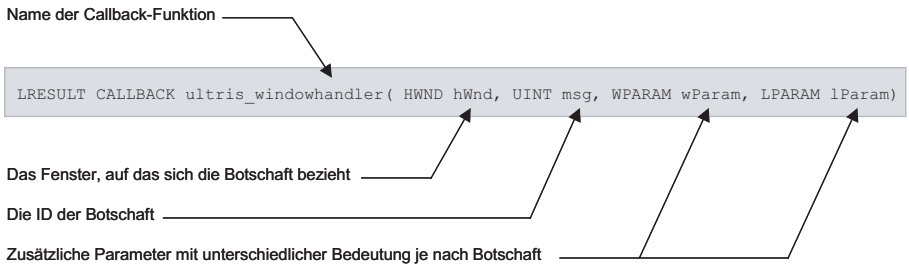
deines Fensters neu!«. Den Grund für das Neuzeichnen kennt Ihr Programm aber nicht, da das andere Programm ja nicht unter seiner Kontrolle läuft. Ähnlich ist es mit den Benutzereingaben für Ihr Programm. Auch diese werden durch Botschaften übermittelt. Sie wissen nie, was der Benutzer als Nächstes tun wird (macht er eine Eingabe, klickt er auf ein bestimmtes Bedienelement oder schließt er das Fenster). Sie müssen jederzeit mit allem rechnen. Ein Windows-Programm befindet sich aus diesem Grund immer in einer so genannten **Hauptverarbeitungsschleife**. In dieser Schleife prüft es, ob Botschaften in der sogenannten **Message-Queue** (Botschaften-Warteschlange) vorliegen. Ist das der Fall, so werden diese Botschaften bearbeitet. Ist das nicht der Fall, so kann sich das Programm anderen Dingen (z. B. der Berechnung eines neuen Bildes für eine Animationssequenz) zuwenden. Auf keinen Fall darf das Programm sich etwa ausschließlich mit der Berechnung und Präsentation einer Animation beschäftigen und dabei seine Pflichten als Windows-Programm vernachlässigen. Es muss sichergestellt sein, dass das Programm regelmäßig prüft, ob zwischenzeitlich neue Botschaften eingetroffen sind, und es muss sichergestellt sein, dass das Programm in angemessener Zeit auf diese Botschaften reagiert.

Ein weiterer kleiner, aber wichtiger Unterschied zwischen einer Windows-Applikation und einem gewöhnlichen C-Programm ist, dass die Windows-Applikation keine `main`- sondern stattdessen eine `WinMain`-Funktion als Einstiegspunkt hat. Diese hat die folgende Schnittstelle:

```
int APIENTRY WinMain( HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR pCmdLine, int nCmdShow)
```

Das Betriebssystem ruft diese Funktion auf, wenn unsere Applikation gestartet wird. Im Parameter `hInst` wird uns ein Handle auf unsere Applikation mitgegeben. Über diesen Handle können wir auf unsere Applikation zugreifen, um beispielsweise einen neuen Dialog zu unserer Applikation hinzuzufügen. Durch den Parameter `nCmdShow` wird uns mitgeteilt, wie sich unsere Applikation auf dem Bildschirm darstellen soll (z. B. minimiert oder maximiert). Die anderen beiden Parameter sind für uns nicht von Bedeutung.

Insbesondere hat die `WinMain`-Funktion die Aufgabe, einen so genannten **Callback-Handler** einzurichten. Dabei handelt es sich um eine Funktion, die vom Betriebssystem immer dann aufgerufen wird, wenn Botschaften für unsere Applikation vorliegen. Diesen Callback-Handler müssen wir mit der folgenden Schnittstelle implementieren:



Den Namen der Callback-Funktion können wir frei wählen, die Schnittstelle müssen wir allerdings so akzeptieren, wie sie ist. Durch die Schnittstelle bekommen wir alle Informationen, die wir zur Reaktion auf eine Message benötigen. Ich werde das später anhand konkreter Messages noch einmal erklären.

Mit diesen Vorkenntnissen wollen wir den Windows-Rahmen für unser Spiel implementieren. Die Größe des Hauptfensters legen wir durch die folgenden Konstanten fest:

```
const int ultris_nettobreite = 360;
const int ultris_nettohoehe  = 520;
```

Die beiden Konstanten definieren die nutzbare Größe des Fensters ohne den Rahmen. Die dazu erforderliche Gesamtgröße des Fensters einschließlich des Rahmens werden wir später berechnen und in den beiden folgenden Variablen ablegen:

```
int ultris_bruttobreite;
int ultris_bruttohoehe;
```

Zusätzlich legen wir einige globale Variablen an, in denen wir wichtige Zugriffsinformationen über unsere Applikation speichern werden:

```
HINSTANCE ultris_instance;
HWND ultris_window;
HMENU ultris_menu;
```

Bei allen drei Datentypen handelt es sich um so genannte Handles. Das sind im Prinzip Zeiger, die den Zugriff auf die Instanz (`HINSTANCE`), das Hauptfenster der Instanz (`HWND`) oder das Menü der Instanz (`HMENU`) ermöglichen. Die Variablen werden in der `WinMain`-Funktion gesetzt. Das folgende Bild zeigt den vollständigen Code der `WinMain`-Funktion. Dieser Code mag Sie auf den ersten Blick verwirren, aber ich werde im Folgenden versuchen, den Knoten für Sie zu entwirren:

```
   int APIENTRY WinMain( HINSTANCE hInst, HINSTANCE hPrevInst,
                                 LPSTR pCmdLine, int nCmdShow)
     {
     MSG msg;
     HACCEL acc;
     WNDCLASSEX wcx;
A    wcx.cbSize = sizeof( wcx);
     wcx.lpszClassName = TEXT( "Ultris");
     wcx.lpfnWndProc = ultris_windowhandler;
     wcx.style = CS_VREDRAW | CS_HREDRAW;
     wcx.hInstance = hInst;
     wcx.hIcon = LoadIcon ( hInst, MAKEINTRESOURCE( IDI_MAIN));
     wcx.hIconSm = LoadIcon( hInst, MAKEINTRESOURCE( IDI_MAIN));
     wcx.hCursor = LoadCursor ( NULL, IDC_ARROW);
     wcx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
     wcx.lpszMenuName = MAKEINTRESOURCE( IDR_MENU);
     wcx.cbClsExtra = 0;
     wcx.cbWndExtra = 0;

     if( !RegisterClassEx ( &wcx))
        return 0;
B    acc = LoadAccelerators ( hInst, MAKEINTRESOURCE(IDR_ACCEL));
C    ultris_bruttohoehe = ultris_nettohoehe
                        + 2*GetSystemMetrics( SM_CYSIZEFRAME)
                        + GetSystemMetrics( SM_CYMENU)
                        + GetSystemMetrics( SM_CYCAPTION);
     ultris_bruttobreite = ultris_nettobreite
                        + 2*GetSystemMetrics( SM_CXSIZEFRAME);
D    ultris_window = CreateWindowEx ( 0, TEXT("Ultris"), TEXT("Ultris"),
                         WS_OVERLAPPEDWINDOW & ~WS_MAXIMIZEBOX,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         ultris_bruttobreite, ultris_bruttohoehe,
                                     NULL, NULL, hInst, NULL);
     if( !ultris_window)
        return 0;
     MoveWindow ( ultris_window,
              (GetSystemMetrics(SM_CXSCREEN)-ultris_bruttobreite)/2,
              (GetSystemMetrics(SM_CYSCREEN)-ultris_bruttohoehe)/2,
                   ultris_bruttobreite, ultris_bruttohoehe, TRUE);
     ShowWindow ( ultris_window, nCmdShow);
E    ultris_instance = hInst;
     ultris_menu = GetMenu ( ultris_window);
F    while( TRUE)
        {
        if( PeekMessage ( &msg, NULL, 0, 0, PM_NOREMOVE))
           {
           if( GetMessage ( &msg, NULL, 0, 0 ) == 0)
              return 0; // Message ist WM_QUIT
```

```
            if( TranslateAccelerator ( ultris_window, acc, &msg) == 0)
               {
               TranslateMessage ( &msg);
               DispatchMessage ( &msg);
               }
            }
         else
            {
            // Hier koennen wir uns um das Spiel kuemmern
            }
         }
      }
```

Zur Entwirrung des Knotens werde ich die Bereiche A-F noch einmal im Detail erklären:

A: Hier wird eine Datenstruktur (`wcx`) mit wichtigen Informationen über die Instanz gefüllt. Von besonderer Bedeutung sind hier die Callback-Funktion (`ultris_windowhandler`), die wir noch erstellen müssen, und das Menü. Das Menü wird über eine symbolische Konstante (`IDR_MENU`) identifiziert.[4] Abschließend wird dann die Instanz beim Betriebssystem registriert, indem die in der Datenstruktur gesammelten Daten an die Funktion `RegisterClassEx` übergeben werden. Misslingt die Registrierung, so wird das Programm beendet.

B: Durch Aufruf der Funktion `LoadAccelerators` werden die Akzelleratoren, das sind Tastaturkürzel zum beschleunigten Zugriff auf Menübefehle, geladen. Die Zuordnung erfolgt wieder über eine symbolische Konstante (`IDR_ACCEL`).

C: In diesem Bereich wird die Bruttogröße des Fensters berechnet. Dazu wird zur Nettobreite zweimal die Rahmendicke addiert. Zur Nettohöhe kommen zweimal die Rahmendicke, die Menühöhe und die Dicke der Leiste mit dem Systemmenü hinzu. Wenn man jetzt ein Fenster in dieser Bruttogröße öffnet, so ist sichergestellt, dass es genau die erforderliche Nettogröße von `360x520` Pixeln hat.

D: Hier wird das Hauptfenster für unser Spiel in der unter C berechneten Bruttogröße erzeugt (`CreateWindowEx`), in der Mitte des Bildschirms ausgerichtet (`MoveWindow`) und auf dem Bildschirm angezeigt (`ShowWindow`). Den Handle auf das Fenster merken wir uns in der globalen Variablen `ultris_window`. Schlägt die Erzeugung des Fensters fehl (Returncode `0`), beenden wir das Programm.

E: Jetzt wird der Zugriff auf die Instanz und das Menü in globalen Variablen gesichert.

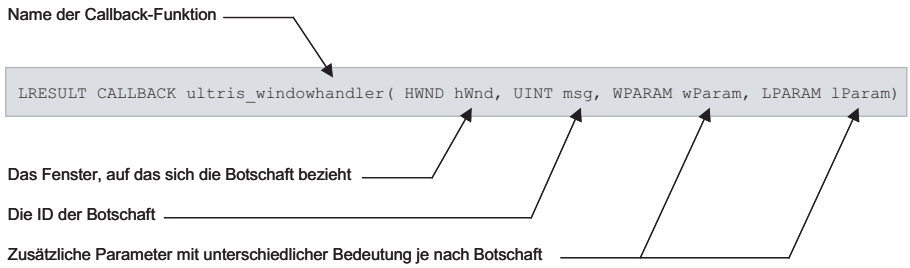
4 Mit dieser Art der Identifikation von Oberflächenelementen werden wir uns noch genauer beschäftigen müssen.

F: Dies ist die Hauptverarbeitungsschleife (Main-Eventloop) unserer Applikation. In einer Endlosschleife (`while(TRUE)`) wird jeweils geprüft, ob eine Botschaft vorliegt (`PeekMessage`), ohne die Botschaft dabei aus der Warteschlange zu entfernen. Liegt eine Botschaft vor, so wird sie aus der Warteschlange gelesen (`GetMessage`). Der Returncode `0` signalisiert in dieser Situation, dass die Applikation beendet werden soll (`WM_QUIT`). Sofern es weitergeht, prüfen wir, ob ein Akzellerator gedrückt wurde, und lassen die zugeordnete Funktion ausführen (`TranslateAccelerator`). Wurde kein Akzellerator betätigt, sorgen wir dafür, dass die Meldung durch Aufruf der Funktionen `TranslateMessage` und `DispatchMessage` dem richtigen Empfänger (Handler) zugestellt wird.

Um eine lauffähige Windows-Applikation zu erhalten, müssen wir noch den Callback-Handler implementieren, dessen Adresse wir bereits oben in der Form

```
wcx.lpfnWndProc = ultris_windowhandler
```

verwendet hatten. Wir erinnern uns dazu noch einmal an die vorgegebene Schnittstelle des Handlers



und diskutieren die Bedeutung der einzelnen Parameter:

`hwnd` ist der Handle des Fensters, auf das sich die Message bezieht. Bei uns wird es sich immer um den Handle des Hauptfensters der Applikation handeln, den wir bereits in der Variablen `ultris_window` gespeichert haben. Insofern ist dieser Parameter für uns nicht sehr aussagekräftig.

`Msg` ist eine Zahl, die uns sagt, um welche Message es sich handelt. Alle vorkommenden Messages sind durch symbolische Konstanten (z. B. `WM_COMMAND`, `WM_PAINT` oder `WM_SIZE`[5]) bezeichnet, sodass uns der konkrete numerische Wert nicht interessiert.

`WParam` ist ein zusätzlicher Parameter, der zusätzliche Informationen über die Message transportiert. Nicht alle Messages verwenden diesen

5 `WM` = Windows-Message

Parameter, und wird er verwendet, so ist seine Bedeutung von Message zu Message verschieden. Im Zusammenhang mit der Message `WM_COMMAND` enthält dieser Parameter zum Beispiel die Information, welches konkrete Kommando gegeben wurde.

`LParam` ist ein weiterer Parameter, der wie `wParam` zusätzliche Informationen über die Message transportiert. Im Zusammenhang mit der Message `WM_COMMAND` enthält dieser Parameter zum Beispiel die Information, auf welches Dialog-Element eines Fensters (z. B. einen Button) sich das Kommando bezieht.

Es ist unmöglich, an dieser Stelle alle Windows-Messages mit ihren Parametern aufzuführen, zumal uns die meisten für unser konkretes Vorhaben nicht interessieren. Ich werde nur auf die Messages eingehen, die im Rahmen unseres Programms verwendet werden. Weitergehende Informationen erhalten Sie aus dem Hilfesystem der Entwicklungsumgebung.

Das folgende Code-Beispiel zeigt den Callback-Handler, soweit wir ihn initial erstellen wollen:

```
   LRESULT CALLBACK ultris_windowhandler( HWND hWnd, UINT msg,
                                  WPARAM wParam, LPARAM lParam)
     {
     switch(msg)
        {
A    case WM_COMMAND:
        switch( LOWORD( wParam))
           {
        case IDM_EXIT:
           PostMessage( hWnd, WM_CLOSE, 0, 0) ;
           return 0;
           }
        break;
B    case WM_GETMINMAXINFO:
        ((MINMAXINFO *)lParam)->ptMinTrackSize.x =
                ((MINMAXINFO *)lParam)->ptMaxTrackSize.x =
                 ultris_bruttobreite;
        ((MINMAXINFO *)lParam)->ptMinTrackSize.y =
                ((MINMAXINFO *)lParam)->ptMaxTrackSize.y =
                 ultris_bruttohoehe;
        return 0;
C    case WM_DESTROY:
        PostQuitMessage( 0) ;
        return 0;
        }
D    return DefWindowProc (hWnd, msg, wParam, lParam);
     }
```

Der Callback-Handler enthält im Wesentlichen eine Sprungleiste, in der entsprechend der Message-Nummer (`switch(msg)`) zu den mit `A`, `B` bzw. `C` bezeichneten Stellen im Code verzweigt wird:

A: Wir haben ein Kommando erhalten. Um welches Kommando es sich handelt, steht im Lower-Word[6] des Parameters `wParam`. Wir eröffnen eine neue Sprungleiste, in der bezüglich des Kommandos verzweigt wird. In dieser Sprungleiste behandeln wir zunächst nur den Fall `IDM_EXIT`. In diesem Fall senden wir eine `WM_CLOSE`-Message, ohne uns Gedanken darüber zu machen, wer diese Message behandelt.

B: Das Window-System will unser Fenster in der Größe verändern und fragt mit der Message `WM_GETMINMAXINFO` an, in welchem Rahmen wir Größenänderungen zulassen. Es schickt uns dabei im Parameter `lParam` einen Zeiger auf eine Datenstruktur vom Typ `MINMAXINFO`, in die wir unsere Minimal- und Maximalwerte eintragen sollen. Dadurch, dass wir die Minimal- und Maximalwerte jeweils gleich halten, erreichen wir, dass das Fenster nicht in der Größe verändert werden kann.

C: Wir werden aufgefordert, unser Fenster zu zerstören, und reagieren darauf durch Aufruf der Funktion `PostQuitMessage`, die die Message `WM_QUIT` verschickt.

D: In allen Fällen, die wir nicht behandeln können oder wollen, rufen wir den vom System bereitgestellten Standard-Handler (`DefWindowProc`) auf.

Beide Sprungleisten spiegeln nur den derzeitigen Entwicklungsstand unseres Programms wieder. Beide Sprungleisten werden wir im Laufe unseres Projekts Schritt für Schritt erweitern.

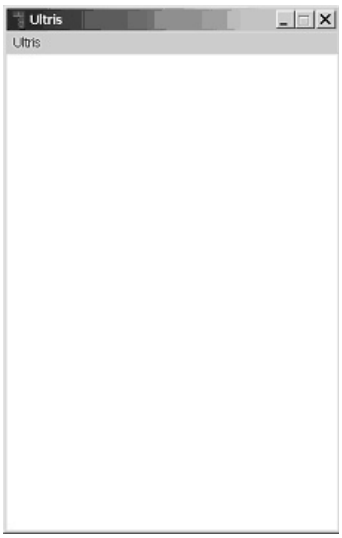
Da es für das Verständnis der Windows-Applikation wichtig ist, fasse ich das noch einmal zusammen:

- Wir erstellen eine Funktion `ultris_windowhandler` mit einer fest vorgegebenen Schnittstelle.
- Diese Funktion melden wir im Hauptprogramm (`WinMain`) beim Betriebssystem als Message-Handler für das Hauptfenster (`ultris_window`) unserer Applikation an.
- Das Betriebssystem ruft diese Funktion immer dann auf, wenn eine Nachricht für das Hauptfenster unserer Applikation vorliegt.

6 Dabei handelt es sich um die beiden niederwertigsten Bytes des 4 Byte großen Parameters, die mit dem Makro `LOWORD` extrahiert werden können.

- Wir reagieren auf die Nachricht, indem wir den jeweils erforderlichen Code ausführen, und geben die Kontrolle (möglichst schnell) wieder an das Betriebssystem zurück.

Wenn Sie jetzt – ausgehend von der von mir bereitgestellten Version `V01` – die Applikation erzeugen, erhalten Sie ein noch weitestgehend funktionsloses Windows-Programm, das Sie starten und beenden können und das ein Fenster in der Größe unseres Spielfeldes in der Bildschirmmitte zeigt:



Es ist durchaus interessant, sich den Kommando-Fluss in dieser einfachen Windows-Anwendung anhand eines Beispiels noch einmal zu verdeutlichen. Wenn wir unser kleines Programm durch Wahl des Menüpunktes *Ende* im Menü *Ultris* beenden wollen, so erhalten wir vom Betriebssystem die Message `WM_COMMAND` mit dem Befehlscode `IDM_EXIT`. Unter Punkt `A` senden wir daraufhin die Message `WM_CLOSE`. Diese Meldung wird uns wieder zugestellt. Da wir diese Meldung aber nicht explizit behandeln, wird die Standardprozedur (`DefWindowProc`) aufgerufen. Diese Standardprozedur erzeugt eine `WM_DESTROY`-Message, mit der wir wiederum aufgefordert werden, unser Fenster zu zerstören. Wir senden daraufhin eine `WM_QUIT`-Message, die in unserer Hauptverarbeitungsschleife entgegengenommen wird und dazu führt, dass das Programm beendet wird.

Auf den ersten Blick mag dieses Vorgehen sehr umständlich erscheinen. Es ist aber sinnvoll, wenn man auf der einen Seite das Window-System mit sehr komplexen Standardabläufen und auf der anderen Seite ein Programm hat, das diese Standardabläufe natürlich nicht neu implementieren, aber doch an gewissen Stellen gezielt in diese Abläufe eingreifen möchte. In dieser Situation kann die Appli-

kation etwa Folgendes festlegen: »Ich möchte das Fenster nicht selbst in der Größe verändern, weil das viel zu kompliziert ist. Aber ich möchte, bevor das Fenster in der Größe geändert wird, gefragt werden, ob ich der Änderung zustimme.« Ich hoffe, dass Sie erkennen, dass man solche Anforderungen am besten in einem Messaging-System umsetzen kann. Botschaften, die mich nicht interessieren, laufen an mir vorbei. Bei Botschaften, die mich interessieren, kann ich mich aber gezielt einschalten.

Wenn Sie unsere Applikation noch einmal betrachten, werden Sie feststellen, dass das Programm bereits ein Menü und einen Icon hat, ohne dass wir das explizit programmiert haben. Diese Beobachtung führt uns zu einem weiteren wichtigen Bestandteil von Windows-Programmen, den so genannten Ressourcen.

Ressource ist ein Sammelbegriff für alle Oberflächen-Elemente einer Windows-Applikation. Ressourcen können zum Beispiel Menüs, Dialoge oder Kontroll-Elemente wie Buttons, Check- oder Listboxen sein. Das folgende Bild zeigt alle Ressourcen im Arbeitsbereich des Startprojektes `V01`:

Jede Ressource hat einen Identifier[7] (`IDR_ACCEL`, `IDI_MAIN`, `IDR_MENU`), den wir ihr selbst zuweisen können. In der Entwicklungsumgebung werden den Identifiern dann konkrete Zahlenwerte zugeordnet, und es wird eine Header-Datei (`Resource.h`) erzeugt, die in unserer Applikation inkludiert wird. Im folgenden Kasten finden Sie einen Auszug aus der Datei `Resource.h`:

```
#define IDI_MAIN                        101
#define IDR_MENU                        102
#define IDR_ACCEL                       103
#define IDM_EXIT                        1001
```

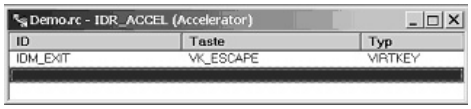
Die Konstanten in dieser Header-Datei sind das Bindeglied zwischen unserem Programm und den Elementen der Benutzeroberfläche. Dabei sind die konkreten numerischen Werte für uns in der Regel nicht von Bedeutung, da wir in unseren Programmen die symbolischen Konstanten verwenden. Sie erinnern sich sicherlich noch an die Funktion `WinMain`, in der unter anderem der folgende Code stand:

7 eindeutige Bezeichnung

```
wcx.hIcon = LoadIcon( hInst, MAKEINTRESOURCE( IDI_MAIN));
wcx.hIconSm = LoadIcon( hInst, MAKEINTRESOURCE( IDI_MAIN));
wcx.hCursor = LoadCursor( NULL, IDC_ARROW);
wcx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wcx.lpszMenuName = MAKEINTRESOURCE( IDR_MENU);
```

Im Code wird hier unter anderem auf das Icon `IDI_MAIN` und das Menü `IDR_MENU` aus der Ressourcen-Datei zugegriffen.

Für alle Ressource-Typen gibt es angepasste Editoren in der Microsoft-Entwicklungsumgebung, mit denen die Ressourcen erstellt und bearbeitet werden können. Durch einen Doppelklick auf die Ressource `IDR_ACCEL` öffnet sich zum Beispiel ein Editor, in dem Sie die Akzelleratorentabelle bearbeiten können:

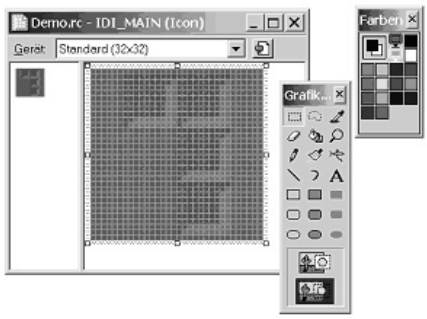


Hier kann man neue Akzelleratoren hinzufügen.

Mit einem Doppelklick auf den Identifier `IDR_MENU` im Arbeitsbereich wird der Menü-Editor geöffnet:

Im Menü-Editor kann man dann das Hauptmenü unserer Applikation um eigene Menüpunkte erweitern. Wir werden später davon Gebrauch machen.

Mit einem Doppelklick auf den Identifier `IDI_MAIN` im Arbeitsbereich wird der Icon-Editor geöffnet:



Im Icon-Editor können Sie die Icons Ihrer Applikation verändern. Beachten Sie, dass es zwei Icons gibt: einen großen im Format 32x32 Pixel und einen im Format 16x16 Pixel. Mit der Combobox im oberen Teil des Icon-Editors können Sie zwischen den beiden Icons hin- und herschalten.

Erstellen Sie jetzt Ihre persönlichen Icons für Ihre Tetris-Variante. Den Umgang mit dem Icon-Editor muss ich Ihnen nicht eigens erklären. Sicherlich haben Sie schon häufiger Pixel- oder Bitmap-Editoren mit vergleichbarer Funktionalität bedient.

Die Ressourcen-Informationen werden übrigens in einer Datei mit der Namenserweiterung *.rc* – in unserem Fall in der Datei *Demo.rc* – gespeichert. Diese Dateien können auch mit einem normalen Texteditor gelesen werden. Der folgende Auszug aus der Datei *Demo.rc* zeigt die Definition des Menüs für unser Spiel:

```
IDR_MENU MENU DISCARDABLE
BEGIN
    POPUP "Ultris"
    BEGIN
        MENUITEM "Ende", IDM_EXIT
    END
END
```

Im Prinzip könnten Sie die Ressourcen also auch mit einem gewöhnlichen Text-Editor erstellen. Aber dazu will ich Ihnen keineswegs raten. Mit einem speziellen Ressourcen-Compiler werden die Ressourcen-Dateien dann in ein Binärformat (*.res*-Dateien) übersetzt. Diese Dateien werden dann ähnlich wie Objekt-Dateien durch den Linker zum Programm hinzugebunden.

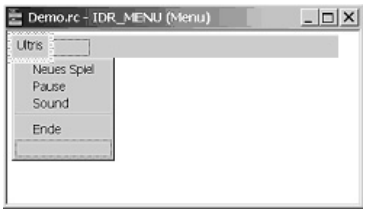
2.4.2 V02 Laden und Initialisieren der Sounds

In dieser Kurseinheit wollen wir die für Ultris benötigten Sounds in unsere Applikation laden. Im Zusammenhang damit wollen wir auch das Menü erweitern und neue Akzelleratoren definieren. Bevor Sie mit den Erweiterungen beginnen, sollten Sie den Ordner V01 duplizieren und das Duplikat in V02 umbenennen. Alle nachfolgend beschriebenen Änderungen nehmen Sie dann nur an dem Projekt im Ordner V02 vor.

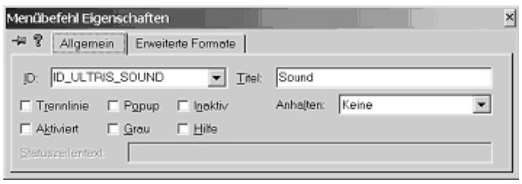
Zur Vorbereitung der weiteren Arbeitsschritte sollten Sie die von Ihnen ausgewählten beziehungsweise erstellten Sound- und Grafikdateien in den Ordner V02 kopieren. Achten Sie dabei darauf, dass die Dateien vollständig sind und die am Ende von Abschnitt 2.3 festgelegten Namen haben.

Wir starten mit der Erweiterung des *Ultris*-Menüs, das ja bisher nur den Menüpunkt *Ende* aufweist:

Starten Sie den Menü-Editor, indem Sie im Arbeitsbereich die Lasche *Ressourcen* wählen und dann im Ordner Menu auf den Identifier `IDR_MENU` doppelklicken. Der Menü-Editor ist intuitiv bedienbar, sodass es Ihnen nicht schwer fallen sollte, die erforderlichen Erweiterungen vorzunehmen:



Neue Menüpunkte geben Sie einfach in das freie Feld am unteren Ende der Liste ein. Anschließend können Sie die Menüpunkte mit der Maus umsortieren. Durch einen Doppelklick auf einen Menüpunkt erhalten Sie einen Dialog, der die Eigenschaften des Menüpunktes anzeigt:

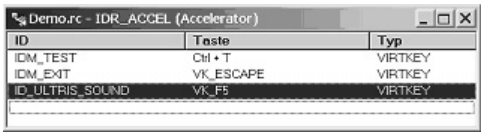


Insbesondere zeigt Ihnen dieser Dialog den vom System für diesen Menüpunkt vergebenen Identifier (im Beispiel oben `ID_ULTRIS_SOUND`).[8] Dieser Identifier stellt die Verbindung zwischen der Ressource und dem Quellcode unserer Applikation dar. In diesem Dialog können Sie auch festlegen, ob es sich bei einem Eintrag nur um eine Trennlinie handeln soll (Checkbox *Trennlinie*).

Nach der Erweiterung des Menüs können Sie die Applikation neu erstellen. Die Applikation hat jetzt das erweiterte Menü, wobei mit den neuen Menüpunkten natürlich noch keine Funktionen verbunden sind. Diese Funktionen müssen wir ja auch erst noch erstellen.

8 Sie könnten den Identifier ändern, aber wir wollen den vorgeschlagenen Namen akzeptieren.

Bevor wir mit dem Erstellen von Funktionen beginnen, wollen wir noch einen Akzellerator hinzufügen. Im Spiel soll der Sound sowohl über den Menüpunkt *Sound* als auch über die Funktionstaste F5 ein- beziehungsweise ausgeschaltet werden. Dazu erweitern wir die Akzelleratorentabelle um den folgenden Eintrag:



Um den Eintrag zu erstellen, doppelklicken Sie auf den freien letzten Eintrag und füllen den nachfolgenden Dialog aus:

Den Eintrag für die Taste erhalten Sie, indem Sie den Button *Nächste Taste* betätigen und auf die folgende Eingabeaufforderung hin die Funktionstaste F5 drücken. Legen Sie auch noch einen Akzellerator (`IDM_TEST`) für die Tastenkombination Ctrl-T (auf der deutschen Tastatur Strg-T) an. Diesen Akzellerator werden wir verwenden, um gewisse Testfunktionen anzustoßen. Zum Beispiel werden wir in dieser Version die Sound-Funktionen über diese Tastenkombination testen.

Die Funktionstaste F5 und der Menüpunkt *Sound* haben jetzt die gleiche Funktion – sie lösen den Befehl `ID_ULTRIS_SOUND` aus. Wir werden im Callback-Handler unserer Applikation auf diesen Befehl reagieren, indem wir den Sound ein- oder ausschalten. Bevor wir das implementieren, müssen wir aber zunächst die Sound-Funktionen erstellen.

Wir werden die acht Sounds in einem Array speichern. Für den Zugriff in diesen Array legen wir einige Konstanten an:[9]

```
const int sound_start = 0; // Sound fuer neues Spiel
const int sound_dreh  = 1; // Sound bei Drehung
const int sound_move  = 2; // Sound bei rechts/links Bewegung
const int sound_down  = 3; // Sound bei Aufprall
const int sound_row1  = 4; // Sound bei Abraeumen einer Reihe
const int sound_row2  = 5; // Sound bei Abraeumen von mehreren Reihen
```

9 »Langweilige« Codepassagen wie diese können Sie durchaus aus der Musterlösung kopieren.

```
const int sound_ende  = 6; // Sound bei Spielende
const int sound_win   = 7; // Sound bei Eintrag in Highscore Tab.
const int anzahl_sounds = 8; // Anzahl Sounds
```

Zusätzlich benötigen wir in unserem Programm die Namen der Sound-Dateien:

```
char *soundfiles[anzahl_sounds] =
    {
    "ul_start.wav",
    "ul_dreh.wav",
    "ul_move.wav",
    "ul_down.wav",
    "ul_row1.wav",
    "ul_row2.wav",
    "ul_ende.wav",
    "ul_win.wav"
    };
```

In den Microsoft-Dateien `Dsutil.h` und `Dsutil.cpp` werden die Klassen `CSoundManager` und `CSound` bereitgestellt. Diese Klassen werden wir verwenden, da sie alle von uns benötigten Sound-Funktionen bereits enthalten. Wir werfen einen kurzen Blick auf die beiden Klassen, wobei ich mich auf deren öffentliche Schnittstelle und dort auf Member-Funktionen beschränkt habe, die wir hier explizit verwenden werden. Darüber hinaus habe ich zur Vereinfachung die Parameter der Member-Funktionen weggelassen.[10]

Die Klasse `CSoundManager` dient zur übergeordneten Organisation – eben zum Management – von Sounds. Uns interessieren hier nur die Funktionen `Initialize` und `Create`:

```
class CSoundManager
    {
    public:
      HRESULT Initialize(...);
      HRESULT Create(...);
      };
```

Mit der Funktion `Initialize` müssen wir den Soundmanager vor seiner Verwendung initialisieren. Beim Initialisieren werden wir technische Parameter wie zum Beispiel die Abtastrate festlegen. Mit der Funktion `Create` können wir einen neuen Sound aus einer Sound-Datei hinzufügen. Im Zusammenhang mit ihrer Verwendung werden wir die Funktionen noch genauer betrachten.

10 Die hier weggelassenen Details finden Sie in `Dsutil.h`.

Die Klasse `CSound` repräsentiert einen konkreten Sound. Das folgende Codefragment zeigt auch wieder nur den uns interessierenden Teil dieser Klasse:

```
class CSound
    {
    public:
        HRESULT Play(...);
        HRESULT Stop();
        HRESULT Reset();
        BOOL IsSoundPlaying();
    };
```

Die Namen der Member-Funktionen sprechen für sich, sodass hier keine weiteren Erklärungen notwendig sind.

Jetzt können wir mit der Implementierung der Sounds für unser Spiel beginnen. Wir legen eine Klasse `sounds` an, die im privaten Bereich einen `CSoundManager` und einen Array (`snd`) mit Zeigern auf die acht erforderlichen Sounds enthält:

```
class sounds
    {
    private:
        CSoundManager smgr;
        CSound *snd[anzahl_sounds];
    public:
        int on;
        sounds();
        int init( HWND wnd);
        void play( int snr);
        ~sounds();
    };
```

Im öffentlichen Bereich der Klasse sind neben dem Konstruktor (`sounds`) und dem Destruktor (`~sounds`) der Ein-/Ausschalter (`on`) und die Methoden `init` und `play` zu finden.

Im Konstruktor initialisieren wir alle Zeiger im Array `snd` mit `0`, um einen konsistenten Initialzustand zu erhalten:

```
sounds::sounds()
    {
    int i;
    for( i = 0; i < anzahl_sounds; i++)
        snd[i] = 0;

    on = 1;
    }
```

Zusätzlich schalten wir den Sound ein (`on = 1`).[11]

Auch Sounds werden einem Fenster (in unserem Fall dem Hauptfenster unserer Applikation) zugeordnet. Sie erkennen dies, wenn Sie etwa, während unsere Applikation einen Sound spielt, ein anderes Fenster in den Vordergrund holen. Das System bringt dann unsere Applikation sofort zum Schweigen. Um diese Zuordnung durchführen zu können, wird bei der Initialisierung der Klasse ein Fenster (`wnd`) als Parameter übergeben. Es wird sich dabei natürlich um das Hauptfenster unserer Applikation (`ultris_window`) handeln:

```
   int sounds::init( HWND wnd)
     {
     HRESULT ret;
     int i;

A    ret = smgr.Initialize( wnd, DSSCL_PRIORITY, 2, 22050, 16);
     if( ret < 0)
        return ret;

B    for( i = 0; i < anzahl_sounds; i++)
        {
        ret = smgr.Create( snd+i, soundfiles[i]);
        if( ret < 0)
           return ret;
        }

     return S_OK;
     }
```

In der Methode `init` initialisieren wir zunächst den `SoundManager` (A). Die zur Initialisierung verwendeten Parameter[12] beziehungsweise Werte sind:

Parameter/Wert	Bedeutung
`wnd`	Dies ist das Fenster, mit dem die Sounds verknüpft werden.
`DSSCL_PRIORITY`	Hier wird festgelegt, wie unsere Applikation mit anderen Applikationen bei konkurrierendem Zugriff auf die Soundkarte kooperiert. Details entnehmen Sie der DirectSound-Dokumentation zum Thema IDirectSound8::SetCooperativeLevel
2	Unser Soundbuffer verwendet zwei Primärkanäle. Das heißt, wir verwenden Stereo-Sound.

11 Wir schalten natürlich nicht den Lautsprecher oder die Soundkarte ein oder aus, sondern merken uns nur in der Member-Variablen `on`, ob Sound-Ausgaben gemacht oder unterdrückt werden sollen.

12 Wenn Sie diese Parameter in ihrer technischen Bedeutung nicht verstehen, so stellt das für die weitere Programmierung kein Problem dar.

Parameter/Wert	Bedeutung
22050	Wir verwenden 22,05 kHz als Sampling-Rate (Abtastfrequenz).
16	Die Anzahl der Bits pro Sample (Abtastung)

Nach der Initialisierung des Sound-Managers werden in einer Schleife die einzelnen Sounds aus ihrer Datei geladen und einem `CSound`-Objekt zugeordnet (B). Einen Zeiger auf das neue Sound-Objekt speichern wir im Array `snd`.

Schlägt irgendeine der Initialisierungen fehl (man erkennt dies am negativen Returncode der gerufenen Funktion), so brechen wir die Funktion vorzeitig ab und reichen den Fehlercode zum rufenden Programm durch. Nur wenn alle Initialisierungen in Ordnung waren, beenden wir die Funktion mit dem Returncode `S_OK`.

Der für die Sounds benötigte Speicher wird in der Funktion `Create` des Sound-Managers dynamisch allokiert. Wir geben diesen Speicher im Destruktor der Klasse `sounds` wieder frei:

```
sounds::~sounds()
    {
    int i;

    for( i = 0; i < anzahl_sounds; i++)
        {
        if( snd[i])
            delete snd[i];
        }
    }
```

Jetzt benötigen wir nur noch eine Funktion, um einen Sound mit einer bestimmten Nummer `i` (`0 ... 7`) zu spielen:

```
    void sounds::play( int i)
       {
A      if( !on)
           return;
B      if( snd[i]->IsSoundPlaying())
           {
           snd[i]->Stop();
           snd[i]->Reset();
           }
       snd[i]->Play( 0, 0);
       }
```

Wichtig ist, dass wir vor dem Abspielen eines Sounds prüfen, ob der Sound für unser Spiel überhaupt eingeschaltet ist (A). Ansonsten verwenden wir hier Funktionen der Klasse `CSound` in nahe liegender Weise (B).

Zum Abschluss legen wir eine Instanz der Klasse `sounds` mit dem Namen `ultris_sounds` an:

```
sounds ultris_sounds;
```

Das Sound-Objekt `ultris_sounds` müssen wir jetzt noch in unsere Windows-Applikation einbetten. Dazu muss Folgendes geschehen:

- Das Sound-Objekt muss bei Applikationsstart initialisiert werden.
- Der Sound muss bei Wahl des Menüpunktes *Sound* oder bei Betätigung der Funktionstaste F5 ein- beziehungsweise ausgeschaltet werden.
- Der Checkmark des Menüpunktes *Sound* muss immer konsistent mit dem Ein-/Ausschalter des Soundobjekts gehalten werden.
- Es sollte eine Funktion erstellt werden, um das Sound-Objekt zu testen.

In der `WinMain`-Funktion nehmen wir dazu die folgenden Erweiterungen vor:[13]

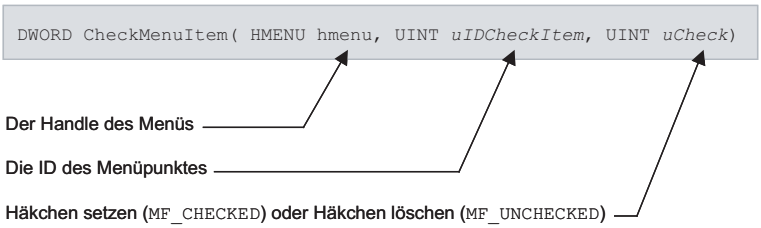
```
   int APIENTRY WinMain(...)
      {
      ...
      ultris_menu = GetMenu( ultris_window);

A     if( ultris_sounds.init( ultris_window) < 0)
         {
         MessageBox( ultris_window,
                     "Fehler beim Initialisieren der Sounds",
                     "Ultris-Fehlermeldung",
                     MB_OK | MB_ICONERROR | MB_SETFOREGROUND);
         return 0;
         }
B     CheckMenuItem( ultris_menu, ID_ULTRIS_SOUND,
                        ultris_sounds.on ?
                                       MF_CHECKED:MF_UNCHECKED);
      ...
      while( TRUE)
         {
         ...
         }
      }
```

13 Der bisherige Code bleibt unverändert und ist aus Gründen der Übersichtlichkeit teilweise nur durch »...« angedeutet.

A: Das Sound-Objekt wird in der `WinMain`-Funktion initialisiert, nachdem `ultris_window` und `ultris_menu` erfolgreich zugewiesen wurden. Schlägt die Initialisierung fehl, wird mit der Windows-Funktion `MessageBox` eine Fehlermeldung auf dem Bildschirm ausgegeben und die Applikation beendet.

B: Je nachdem, ob der Sound eingeschaltet ist oder nicht, wird der Menüpunkt Sound mit einem Checkmark (Häkchen) versehen oder nicht. Die dazu erstmals verwendete Funktion `CheckMenuItem` wird im Folgenden noch einmal genauer betrachtet.

Bei der Auswahl des Menüpunktes *Sound* oder beim Drücken von F5 erhalten wir die Kommando-Message `ID_ULTRIS_SOUND`. Beim Eintreffen dieser Message müssen wir den Ein-/Ausschalter (`on`) des Sound-Objekts umlegen und das Häkchen im Menü Sound anpassen. Um ein Häkchen in einem Menü zu setzen, verwenden wir die Windows-Funktion `CheckMenuItem` mit der folgenden Schnittstelle:



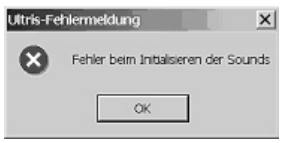
Der folgende Auszug aus der Funktion `ultris_windowhandler` zeigt den dazu erforderlichen Code unter A:

```
   LRESULT CALLBACK ultris_windowhandler(...)
      {
      switch (msg)
          {
      case WM_COMMAND:
          switch( LOWORD( wParam))
              {
          ...
A         case ID_ULTRIS_SOUND:
              ultris_sounds.on = !ultris_sounds.on;
              CheckMenuItem( ultris_menu, ID_ULTRIS_SOUND,
                             ultris_sounds.on ?
                                    MF_CHECKED:MF_UNCHECKED);
              return 0;
B         case IDM_TEST:
              static int testno = 0;
              ultris_sounds.play( testno % anzahl_sounds);
              testno++;
              return 0;
```

```
            }
        break;
    case WM_GETMINMAXINFO:
        ...
        }
    ...
    }
```

Drückt der Benutzer die Tastenkombination Strg-T, so führt dies zum Kommando `IDM_TEST`. In dieser Situation (siehe B im obigen Codefragment) wollen wir einen einfachen Sound-Test durchführen. Mit Hilfe einer statischen Variablen zählen wir alle Sound-Nummern durch und spielen bei jeder Betätigung von Strg-T den jeweils nächsten Sound. Dieser Code wird in der nächsten Version wieder gelöscht, um dort gegebenenfalls eine andere Testfunktion zu implementieren.

Wenn Sie alles richtig implementiert haben, sollten Sie Ihre Sounds jetzt testen können, ohne die folgende Fehlermeldung zu Gesicht zu bekommen:



Testen Sie jetzt die Sounds, das An- und Abschalten des Sounds und die korrekte Verwendung des Häkchens im *Ultris*-Menü.

2.4.3 V03 Laden und Initialisieren der Grafiken

In dieser Kurseinheit wollen wir die Grafiken des Spiels laden und testweise auf dem Bildschirm anzeigen. Bevor Sie damit beginnen, sollten Sie die Version `V03` als Kopie aus der Version `V02` erzeugen. In dieser neuen Version entfernen Sie dann zunächst die `V02`-spezifische Testfunktionalität – also den Code unter `IDM_TEST`.

Analog zu den Sounds werden wir auch bei den Grafiken eine Klasse anlegen, die alle erforderlichen Grafikoperationen enthält. Diese Klasse wollen wir `display` nennen

```
class display
    {
    private:
    ...
    public:
    ...
    };
```

und in mehreren Schritten entwickeln. Bei der Implementierung greifen wir auf die in den Microsoft-Quellen `Ddutil.h` und `Ddutil.cpp` definierten Klassen `CDisplay` und `CSurface` zurück. Die Klasse `CDisplay` stellt eine allgemeine Schnittstelle zum Grafiksystem bereit, während Instanzen der Klasse `CSurface` die einzelnen darzustellenden Bitmaps aufnehmen. Die Elemente der Oberfläche (Hintergrund, Abdeckung, Ziffern und Steine) werden jeweils aus der Bitmap-Datei in eine `CSurface` eingelesen. Die Klasse `CDisplay` dient dazu, die übergreifenden Dienste (zum Beispiel zum Erstellen von Surfaces) bereitzustellen. Wir werfen zunächst einen kurzen Blick auf diese beiden Klassen, wobei wir uns wieder auf die in diesem Abschnitt verwendeten Funktionen beschränken.

Bei der Klasse `CDisplay` interessieren uns die folgenden Funktionen:

```
class CDisplay
    {
     public:
        HRESULT CreateWindowedDisplay(...);
        HRESULT CreateSurfaceFromBitmap(...);
        HRESULT Blt(...);
        HRESULT Present();
        LPDIRECTDRAW7 GetDirectDraw();
        HRESULT UpdateBounds();
    };
```

Im Einzelnen haben diese Funktionen die folgende Bedeutung:

Member-Funktion	Bedeutung
`CreateWindowedDisplay`	Erzeugt ein Display in einem Fenster.
`CreateSurfaceFromBitmap`	Erzeugt eine Surface aus einer Bitmapdatei.
`Blt`	Kopiert eine Surface in das Display.
`Present`	Aktualisiert das Display im Fenster.
`GetDirectDraw`	Gibt einen Zeiger auf das mit dem Display verbundene DirectDraw-Objekt zurück.
`UpdateBounds`	Aktualisiert das Display bei Größenänderung des Fensters.

Von der Klasse `CSurface` verwenden wir nur eine Funktion (`DrawBitmap`):

```
class CSurface
    {
    HRESULT DrawBitmap();
    };
```

Diese Funktion überträgt eine Bitmap aus einer Datei in eine Surface.

Im ersten Schritt versehen wir die Klasse `display` mit dem `CDisplay` und den erforderlichen `CSurfaces`. Gleichzeitig sehen wir den Konstruktor (`display`), den Destruktor (`~display`) und eine allgemeine Freigabefunktion (`free_all`) vor:

```
class display
    {
    private:
        CDisplay dsply;      // das Display
        CSurface *hgrnd;     // der Hintergrund
        CSurface *fldst;     // ein Stein im Feld
        CSurface *fllst;     // ein Stein in einer
                                        herabfallenden Form
        CSurface *prvst;     // ein Stein in der Vorschau
        CSurface *deckel;    // der Deckel
        CSurface *ziff[10];  // die Ziffern 0 - 9
    public:
        display();
        void free_all();
        ~display() {free_all();}
    };
```

Im Konstruktor werden alle Zeiger mit dem Wert `0` initialisiert:

```
display::display()
    {
    int i;

    hgrnd = 0;
    fldst = 0;
    fllst = 0;
    prvst = 0;
    deckel = 0;

    for( i = 0; i < 10; i++)
        ziff[i] = 0;
    }
```

In der `free_all`-Funktion werden alle dynamisch angelegten Objekte wieder beseitigt:

```
void display::free_all()
    {
    int i;

    if( hgrnd)
        delete hgrnd;
    if( fldst)
        delete fldst;
    if( fllst)
        delete fllst;
    if( prvst)
        delete prvst;
    if( deckel)
        delete deckel;

    for( i = 0; i < 10; i++)
        {
        if( ziff[i])
            delete ziff[i];
        }
    }
```

Der Destruktor `~display` ruft lediglich die `free_all`-Funktion auf und ist bereits oben in der Klasse implementiert worden.

Im nächsten Schritt fügen wir zur Klasse `display` eine Initialisierungsmethode hinzu:

```
class display
    {
    private:
        ...
    public:
        ...
        HRESULT init( HWND wnd);
    };
```

In dieser Methode werden das Display und die Surfaces erzeugt. Für die Initialisierung des Displays durch die Funktion `CreateWindowedDisplay`[14] benötigen wir den Handle des umschließenden Fensters (`wnd`) sowie die gewünschten Abmessungen (`ultris_nettobreite`, `ultris_nettohoehe`) des Displays:

14 Alternativ gibt es eine Funktion `CreateFullScreenDisplay`, mit der Sie ein Spiel im Vollbild-Modus erstellen können. Viele Spiele werden im Vollbild-Modus erstellt, um den gesamten Bildschirm exklusiv zu nutzen. Im Vollbild-Modus müssen Sie dann allerdings die gesamte Benutzerführung selbst programmieren, da Sie nicht auf Standardelemente wie Menüs oder Dialoge zurückgreifen können.

```
HRESULT display::init( HWND wnd)
   {
   HRESULT hr;
   int i;
   char fname[20];

   hr = dsply.CreateWindowedDisplay( wnd, ultris_nettobreite,
                                          ultris_nettohoehe );
   if( hr < 0)
      return hr;
   hr = dsply.CreateSurfaceFromBitmap( &hgrnd, "ul_hgrnd.bmp",
                     ultris_nettobreite, ultris_nettohoehe);
   if( hr < 0)
      return hr;
   hr = dsply.CreateSurfaceFromBitmap( &fldst, "ul_feld.bmp",
                                                   20, 20);
   if( hr < 0)
      return hr;
   hr = dsply.CreateSurfaceFromBitmap( &fllst, "ul_stein.bmp",
                                                   20, 20);
   if( hr < 0)
      return hr;
   hr = dsply.CreateSurfaceFromBitmap( &prvst, "ul_prev.bmp",
                                                   15, 15);
   if( hr < 0)
      return hr;
   hr = dsply.CreateSurfaceFromBitmap( &deckel,
                                  "ul_adeck.bmp", 240, 100);
   if( hr < 0)
      return hr;

   for( i = 0; i < 10; i++)
      {
      sprintf( fname, "ul_z%d.bmp", i);
      hr = dsply.CreateSurfaceFromBitmap( &ziff[i], fname, 20,
                                                        40);
      if( hr < 0)
         return hr;
      }
   return S_OK;
   }
```

Die einzelnen Surfaces werden durch Aufruf der Funktion `CreateSurface-FromBitmap` dynamisch aus der jeweiligen Bitmap-Datei erzeugt. Als Parameter werden dabei übergeben:

- die Adresse eines Zeigers, in dem die Referenz auf die dynamisch erzeugte Surface abgelegt wird

- der Name der Bitmap-Datei[15]
- die Abmessungen (Breite und Höhe) der Bitmap

Tritt bei der Erzeugung eines Elements ein Fehler auf, erfolgt ein vorzeitiger Rücksprung mit einem Fehlercode (`hr`) in das aufrufende Programm. Da eine Surface immer genau einer Bitmap entspricht, werde ich im Folgenden nicht immer sauber zwischen diesen Begriffen unterscheiden. Ich hoffe, Sie sehen mir das nach.

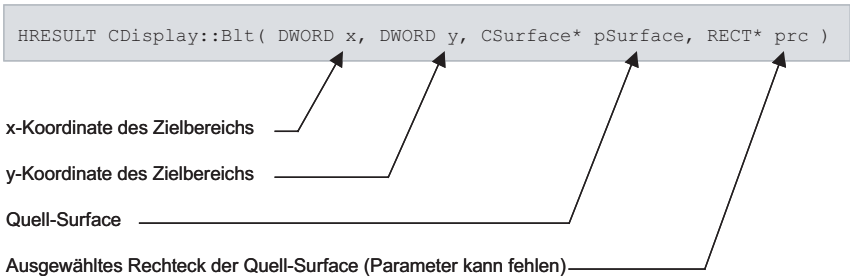
Bevor wir weiteren Code erstellen, sollte ich erwähnen, wie man mit Hilfe eines Displays eine für das menschliche Auge kontinuierlich wirkende Animation auf den Bildschirm bringt. Sie wissen, dass ein Kinofilm aus einer Folge von Standbildern besteht. Wenn man in der Abfolge der Standbilder eine Frequenz von mehr als 20 Bildern (Frames) pro Sekunde erreicht, entsteht für das menschliche Auge der Eindruck einer kontinuierlichen Bewegung. Ganz ähnlich arbeitet unser Display. Es hat einen Front- und einen Backbuffer. Im Frontbuffer ist das Bild, das jeweils angezeigt wird. Im Backbuffer können wir ein neues Bild (den nächsten Frame) aufbauen. Zur Darstellung des nächsten Frames wird dann einfach zwischen Front- und Backbuffer umgeschaltet. Im Prinzip brauchen wir also zwei Funktionen:

- eine Funktion, um Surfaces gezielt in den Backbuffer des Displays zu laden und
- eine Funktion, um zwischen Frontbuffer und Backbuffer des Displays umzuschalten.

Die erste Funktion heißt `Blt`, wobei `Blt` die Abkürzung von Blocktransfer ist. Mit Hilfe dieser Funktion können wir die Bitmap einer Surface (Quell-Surface) gezielt an eine bestimmte Position unseres Displays (Zielbereich) transferieren. Man bezeichnet diesen Vorgang auch mit dem Kunstwort **blitten**. Das folgende Bild zeigt die Schnittstelle der `Blt`-Funktion:[16]

15 Sie können die Bitmap auch als Ressource in Ihr Programm aufnehmen und dann als Ressource laden. Dazu schreiben Sie anstelle des Dateinamens `MAKEINTRESOURCE(IDB_XXX)`, wenn `IDB_XXX` die Ressource-ID der gewünschten Bitmap ist. Der Vorteil ist, dass die Bitmaps in das ausführbare Programm integriert werden und nicht mehr separat gespeichert werden müssen. Das erleichtert die Installation auf anderen Computern. Der Nachteil (oder vielleicht auch Vorteil) ist, dass der Endbenutzer die Bitmaps nicht ändern kann.

16 Wenn Sie sich die Implementierung der `Blt`-Funktion in der Klasse `CDisplay` anschauen, werden Sie feststellen, dass dort die Funktion `BltFast` der DirectDraw-Surface aufgerufen wird. Es gibt eine weitere Blt-Funktion `für` DirectDraw-Surfaces. (`IDirectDrawSurface3::Blt`), die es zusätzlich ermöglicht, eine Bitmap beim Blitten zu rotieren. Für Ultris benötigen wir diese Funktionalität nicht, aber für Ihre eigenen Spiele sollten Sie wissen, dass diese Möglichkeit besteht. Die Details finden Sie im Hilfesystem.



Wird kein Teil-Rechteck (`prc`) der Quell-Surface ausgewählt, so wird die gesamte Bitmap zum Zielbereich geblittet.

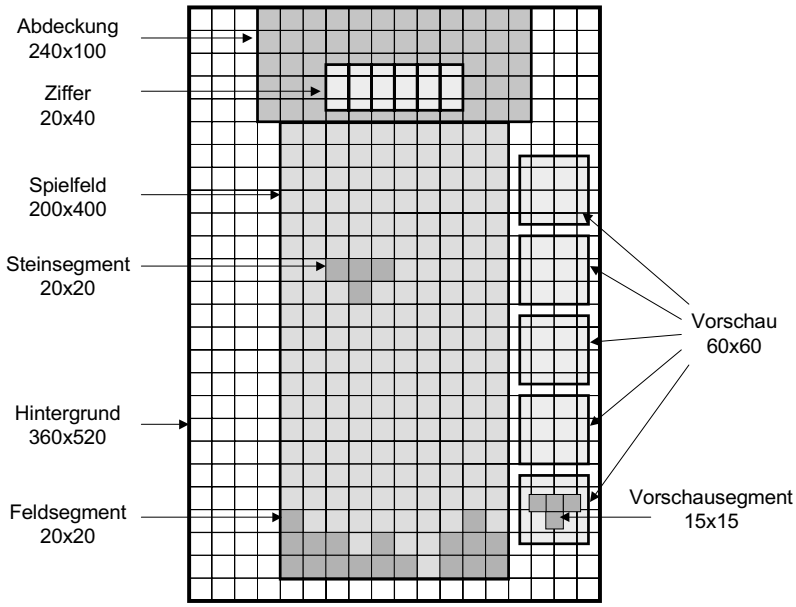
Zum Umschalten zwischen Front- und Backbuffer dient die parameterlose Funktion `Present`:

```
HRESULT CDisplay::Present()
```

Mit der ersten der beiden Funktionen können wir jetzt die erforderlichen Erweiterungen der Klasse `display` zur Darstellung der Bitmaps vornehmen:

```
class display
   {
   private:
      ...
   public:
      ...
      void hintergrund() { dsply.Blt( 0, 0, hgrnd);}
      void abdeckung() { dsply.Blt( 60, 0, deckel);}
      void ziffer(int pos,int val){dsply.Blt(120+pos*20,50,ziff[val]);}
      void feldstein(int z,int s){dsply.Blt(80+s*20,100+z*20,fldst);}
      void fallstein( int z, int s, int offset)
                  { dsply.Blt( 80+s*20,100+z*20+offset, fllst);}
      void prevstein( int p, int z, int s, int b, int h)
       { dsply.Blt(290+s*15+(4-b)*15/2,410-p*70+z*15+(4-h)*15/2,prvst);}
   };
```

Zum besseren Verständnis der Koordinatenberechnungen in diesen Funktionen betrachten wir noch einmal die Skizze der Oberfläche:



Der Hintergrund beginnt in der linken oberen Ecke (`x = 0`, `y = 0`) des Displays und füllt den gesamten Fensterinhalt aus:

```
dsply.Blt( 0, 0, hgrnd)
```

Die linke obere Ecke der Abdeckung liegt im Punkt mit den Koordinaten `x = 60` und `y = 0`:

```
dsply.Blt( 60, 0, deckel)
```

Die sechs Ziffern haben Positionen mit fester y-Koordinate (`y = 50`), die sich in der x-Koordinate (`x = 120`, `140`, `160`, ...) unterscheiden. Die Ziffer mit dem Index pos ist damit an der Stelle mit den Koordinaten `x = 120+pos*20` und `y = 50` zu zeichnen:

```
dsply.Blt(120+pos*20,50,ziff[val])
```

Der Ziffernwert (`val`) entscheidet dabei über die Auswahl der Bitmap.

Ein einzelnes Segment eines gefallenen Steins liegt in einer bestimmten Zeile (`z`) und Spalte (`s`) im Feld. Das Feld, in dem die Steine fallen, beginnt bei `x = 80` und `y = 100`. Die zugehörigen Koordinaten berechnen sich dann bei einer Spalten- und Zeilenbreite von `20` Pixeln wie folgt: `x = 80+s*20` und `y = 100+z*20`:

```
dsply.Blt(80+s*20,100+z*20,fldst)
```

Auch ein fallendes Steinsegment befindet sich immer in einer bestimmten Spalte (s). Es liegt jedoch nicht exakt in einer Zeile, da es ja Pixel für Pixel nach unten fällt. Zu der letzten Zeile (z), in der es sich befunden hat, kommt daher immer noch eine Zugabe (offset), um die das Segment inzwischen weitergerückt ist. In Formeln heißt das x = 80+s*20 und y = 100+z*20+offset:

```
dsply.Blt( 80+s*20, 100+z*20+offset, fllst)
```

Am komplexesten ist die Berechnung der Position der Vorschau-Steine, da diese Steine in ihren Ausgabebereichen zentriert dargestellt werden sollen. Zunächst einmal gibt es bis zu fünf Vorschau-Steine (p = 0,...,4). Die linke obere Ecke des Ausgabebereichs für den Vorschau-Stein p ist durch die Koordinaten x = 290 und y = 410-p*70 gegeben. Wenn man nun das Segment in der Spalte s und in der Zeile z des Vorschau-Steins zeichnen will, so liegt die linke obere Ecke eines solchen Segments wegen der Segmentgröße von 15x15 Pixeln im Punkt mit den Koordinaten x = 290+s*15 und y = 410-p*70+z*15. Kennt man zusätzlich die Gesamtgröße (= Anzahl der Segmente) des Steins in der Höhe (h) und in der Breite (b), so kann man den für die Zentrierung eines Steins im 4x4 Segmente großen Vorschau-Bereich erforderlichen Randausgleich berechnen und erhält die folgenden Koordinaten

x = 290+s*15+(4-b)*15/2 und

y = 410-p*70+z*15+(4-h)*15/2:

```
dsply.Blt(290+s*15+(4-b)*15/2,410-p*70+z*15+(4-h)*15/2,prvst);
```

Mit Hilfe der Funktion Present können wir dann die Grafiken aus dem Backbuffer auf den Bildschirm bringen. Im Zusammenhang mit dieser Funktion müssen wir das wichtige Phänomen der »Lost Devices« diskutieren. Der Zugriff auf Devices wie die Grafikkarte kann »verloren gehen«. Das Device befindet sich dann aus der Sicht unserer Applikation im so genannten »lost state«. Eine Ursache für einen lost state kann zum Beispiel sein, dass sich eine andere Applikation der Grafikkarte im Fullscreen-Modus bemächtigt hat. Ist das Device im lost state, so werden Grafikoperationen, wie zum Beispiel das Blitten von Bitmaps in den Backbuffer, die nur auf den internen Speicher zugreifen, ohne Fehler durchgeführt. Erst beim Umschalten von Front- und Backbuffer (Present) wird der lost state kritisch, da jetzt auf die Grafikkarte zugegriffen werden muss. Als Returncode erhalten wir in dieser Situation die Fehlermeldung DDERR_SURFACELOST. Wir müssen darauf reagieren, indem wir alle verlorenen Daten auf der Grafikkarte restaurieren. Zusammen mit der Funktion present fügen wir daher eine Funktion restore in unsere Klasse display ein:

```
class display
    {
    private:
        ...
    public:
        ...
        HRESULT restore();
        HRESULT present();
    };
```

In der Funktion `restore` restaurieren wir alle Surfaces unseres Displays und zeichnen anschließend alle Bitmaps in allen Surfaces neu. Der folgende Kasten zeigt den dafür erforderlichen Code:

```
HRESULT display::restore()
    {
    HRESULT hr;
    int i;
    char fname[20];

    hr = dsply.GetDirectDraw()->RestoreAllSurfaces();
    if( hr < 0)
        return hr;
    hr = hgrnd->DrawBitmap( "ul_hgrnd.bmp",
                    ultris_nettobreite, ultris_nettohoehe);
    if( hr < 0)
        return hr;
    hr = fldst->DrawBitmap( "ul_feld.bmp", 20, 20);
    if( hr < 0)
        return hr;
    hr = fllst->DrawBitmap( "ul_stein.bmp", 20, 20);
    if( hr < 0)
        return hr;
    hr = prvst->DrawBitmap( "ul_prev.bmp", 15, 15);
    if( hr < 0)
        return hr;
    hr = deckel->DrawBitmap( "ul_adeck.bmp", 240, 100);
    if( hr < 0)
        return hr;
    for( i = 0; i < 10; i++)
        {
        sprintf( fname, "ul_z%d.bmp", i);
        hr = ziff[i]->DrawBitmap( fname, 20, 40);
        if( hr < 0)
            return hr;
        }
    return S_OK;
    }
```

In der present-Funktion versuchen wir zunächst die korrespondierende Present-Funktion von CDisplay auszuführen. Misslingt das wegen DDERR_SURFACELOST, versuchen wir nur die Surfaces zu restaurieren. Beim nächsten Aufruf von present sind dann die Surfaces (hoffentlich) wieder da:

```
HRESULT display::present()
    {
    HRESULT hr;

    hr = dsply.Present();
    if( hr == DDERR_SURFACELOST)
        return restore();
    return hr;
    }
```

Zum Abschluss vervollständigen wir die Klasse display durch zwei weitere Member-Funktionen, die wir wegen ihres geringen Codeumfangs direkt in der Klasse display implementieren können:

```
class display
   {
   private:
      ...
   public:
      ...
      void update(){ dsply.UpdateBounds();}
      HRESULT cooperative()
         {return dsply.GetDirectDraw()->TestCooperativeLevel();}
   };
```

Die erste Member-Funktion (update) müssen wir immer dann aufrufen, wenn sich die Lage unseres Hauptfensters auf dem Bildschirm verändert hat. Durch Aufruf der Funktion UpdateBounds passen wir dann das Display an die neue Situation an.

Die Funktion cooperative dient dazu, die »Kooperativität« des Displays festzustellen. Was es damit auf sich hat, werde ich Ihnen am Ende dieser Kurseinheit sagen.

Die Klasse display ist jetzt vollständig implementiert, und wir legen eine Instanz dieser Klasse mit dem Namen ultris_display an:

```
display ultris_display;
```

Dieses Objekt müssen wir zum Abschluss dieser Kurseinheit in die Windows-Applikation integrieren. Dazu initialisieren wir unser Display in der `WinMain`-Funktion, bevor das Fenster mit `ShowWindow` dargestellt wird:

```
int APIENTRY WinMain(...)
    {
    ...
    if( ultris_display.init( ultris_window) < 0)
        {
        MessageBox( ultris_window,
                    "Fehler beim Initialisieren der Grafik",
                    "Ultris-Fehlermeldung",
                    MB_OK | MB_ICONERROR | MB_SETFOREGROUND);
        return 0;
        }

    ShowWindow( ultris_window, nCmdShow);

    while( TRUE)
        {
        ...
        }
    }
```

Falls die Initialisierung fehlschlägt, brechen wir die Applikation mit einem Fehlerdialog (`MessageBox`) ab. Ansonsten geht es mit der Anzeige des Fensters (`ShowWindow`) und dem Main-Event-Loop (`while(TRUE)`) weiter.

Auch im Callback-Handler unseres Hauptfensters (`ultris_windowhandler`) werden wir einige wenige Erweiterungen vornehmen, um wieder zu einer testfähigen Version zu kommen. Immer wenn das Fenster bewegt oder in seiner Größe verändert wurde, erhalten wir eine Benachrichtigung durch die Message `WM_MOVE`. Als Reaktion darauf rufen wir die `update`-Funktion unseres Displays auf (A):

```
   LRESULT CALLBACK ultris_windowhandler(...)
      {
      switch (msg)
          {
          ...
A     case WM_MOVE:
          ultris_display.update();
          return 0;
B     case WM_PAINT:
          int i;
          ultris_display.hintergrund();
          ultris_display.abdeckung();
```

```
        for( i = 0; i < 6; i++)
            ultris_display.ziffer( i, i+1);
        for( i = 0; i < 10; i++)
            ultris_display.feldstein( 19-i, i);
        for( i = 0; i < 10; i++)
            ultris_display.fallstein( 1, i, 2*i);
        for( i = 0; i < 4; i++)
            ultris_display.prevstein( 3, 0, i, 4, 1);
        ultris_display.present();
        break;
        }
    ...
    }
```

Über die message-orientierte Architektur des Windows-Systems hatte ich Ihnen bereits einiges erzählt. Insbesondere bedingt diese Architektur, dass wir unsere Fensterausgaben nicht spontan machen dürfen. Wir müssen immer warten, bis wir vom System dazu aufgefordert werden. Das System schickt uns als Aufforderung zum Neuzeichnen unseres Hauptfensters die Message `WM_PAINT`. Wenn wir diese Message erkennen (B im obigen Code-Fragment), machen wir einige Testausgaben. Wir geben mit Hilfe der in diesem Abschnitt erstellten Klasse `ultris_display` den Hintergrund, die Abdeckung sowie einige Ziffern und Steine auf dem Bildschirm aus.

In der Funktion `WinMain` hatten wir bereits eine Stelle lokalisiert, an der wir regelmäßig wiederkehrende, das Spiel betreffende Aktivitäten einbauen können:

```
int APIENTRY WinMain(...)
    {
    ...
    while( TRUE)
        {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE))
            {
            ...
            }
        else
            {
            // Hier koennen wir uns um das Spiel kuemmern
            }
        }
    }
```

An diese Stelle gelangen wir in unserer Hauptverarbeitungsschleife immer dann, wenn keine aktuell zu verarbeitende Windows-Message vorliegt. An dieser Stelle können wir dann den Spielverlauf programmieren (z. B. einen fallenden Stein ein

Stück weiterbewegen). Zunächst nutzen wir diese Stelle aber, um regelmäßig den Zustand unseres Devices zu kontrollieren, indem wir die Kooperativität prüfen:

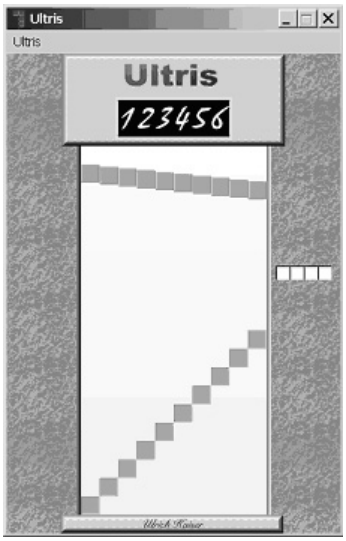
```
int APIENTRY WinMain(...)
    {
    ...
    while( TRUE)
        {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE))
            {
            ...
            }
        else
            {
            HRESULT hr;
            hr = ultris_display.cooperative();
            if( hr < 0)
                {
                switch( hr )
                   {
                case DDERR_EXCLUSIVEMODEALREADYSET:
                   Sleep(10);
                   break;
                case DDERR_WRONGMODE:
                   ultris_display.free_all();
                   ultris_display.init( ultris_window);
                   PostMessage( ultris_window, WM_PAINT, 0, 0);
                   break;
                   }
                }
            else
                {
                // Hier koennen wir uns um das Spiel kuemmern
                }
            }
        }
    }
```

Wenn der Aufruf der Funktion `ultris_display.cooperative()` ein negatives Ergebnis liefert, so liegt ein Problem vor. Zwei mögliche Probleme interessieren uns an dieser Stelle:

- `DDERR_EXCLUSIVEMODEALREADYSET`
  Dies bedeutet, dass eine andere Applikation sich das Grafik-Device exklusiv gesichert hat. Wir können in dieser Situation nichts anderes tun, als uns eine gewisse Zeit schlafen zu legen und auf bessere Zeiten zu hoffen.

- `DDERR_WRONGMODE`
  Irgendjemand hat den Grafikmodus (zum Beispiel die Auflösung) geändert. Wir müssen in dieser Situation alles noch einmal neu initialisieren. Zuvor geben wir die von uns belegten Ressourcen frei, anschließend schicken wir uns selbst die Windows-Message `WM_PAINT`, die uns veranlasst, das Grafikfenster neu zu zeichnen.[17]

Wenn Sie das Programm übersetzen und starten, sollten Sie den folgenden Bildschirm sehen:



Tritt ein Fehler bei der Initialisierung der Grafik auf, erhalten Sie dagegen die folgende Fehlermeldung:

Das Programm wird nach Quittierung der Fehlermeldung beendet.

Die Test-Ausgaben unter `WM_PAINT` werden wir in der nächsten Version wieder beseitigen. Ich empfehle Ihnen an dieser Stelle, zunächst noch weitere Test-Ausgaben zu programmieren, bis Sie sattelfest in der Verwendung des Displays sind.

17 Es ist nicht unüblich, sich selbst eine Message zu schicken. Es ist sogar sehr elegant, weil man dadurch eine Funktion nicht direkt ausführt, sondern in den normalen Bearbeitungszyklus einer Windows-Anwendung einpasst.

Testen Sie das Programm jetzt auch in verschiedenen Bildschirmauflösungen. Schalten Sie dabei die Bildschirmauflösung auch bei laufendem Programm um.

Es gibt im Übrigen weitere Meldungen im Zusammenhang mit der Grafikkarte, auf die man reagieren könnte und vielleicht auch sollte. Ist das System beispielsweise im 8-Bit-Grafikmodus (256 Farben), so arbeitet die Grafikkarte mit einer Farbpalette, und wir werden unter Umständen aufgefordert, unsere Farbpalette neu zu initialisieren, weil sie durch eine andere Applikation überschrieben wurde. Wir erhalten in dieser Situation die Message `WM_QUERYNEWPALETTE`. Als Reaktion müssten wir die gewünschte Palette erstellen und beim System anmelden. Da ich aber für *Ultris* Bitmaps mit mehr als 256 Farben verwende, und stillschweigend voraussetze, dass das System im hochauflösenden Farbmodus ist, gehe ich auf diese Thematik nicht ein. Es ist inzwischen auch sehr unwahrscheinlich, dass auf einem für Computerspiele verwendeten PC der 8-Bit-Farbmodus verwendet wird.

In der Rückschau auf diesen Abschnitt werden Sie vielleicht einwenden, dass es offensichtlich nur möglich ist, rechteckig berandete Objekte auf den Bildschirm zu blitten. In Ihren Spielen wollen Sie aber runde Objekte (z. B. eine Kugel) oder sogar beliebig berandete Figuren (gegebenenfalls sogar mit »Löchern«) vor einem Hintergrund bewegen. Auch das ist natürlich möglich. Man verwendet dazu das so genannte **Color-Keying**. Dazu wird eine in der darzustellenden Bitmap nicht vorkommende Farbe als **Color-Key** für die Surface ausgewählt. Alles, was dann in der Farbe des Color-Keys eingefärbt wird, wird beim Blitten der Surface nicht transferiert und ist somit transparent. Das ist so wie eine Blue-Box, die Sie vielleicht vom Fernsehen her kennen. Die auszustanzende Farbe können wir dabei beliebig festlegen.

Als Beispiel stellen Sie sich vor, dass die gewünschte Bitmap in einer Datei vorliegt und alles, was nicht gezeichnet werden soll, schwarz ist. Sie legen dann wie gewohnt eine Surface (`CSurface *s`) an und initialisieren diese mit der Funktion `CreateSurfaceFromBitmap`. Danach setzen Sie den Color-Key der Surface durch den Funktionsaufruf:

```
s->SetColorKey( RGB(0, 0, 0));
```

Der Makro `RGB` stellt dabei aus dem Rot-, Grün- und Blau-Anteil einer Farbe den zugehörigen Farbcode zusammen. In diesem Fall handelt es sich um Schwarz. Bei nachfolgenden Blt-Operationen werden bei dieser Surface schwarze Bildpunkte nicht transferiert, und der Hintergrund scheint an schwarz gefärbten Stellen durch. Wenn Sie andere Farben als Color-Key verwenden wollen, wählen Sie entsprechend andere RGB-Werte. Wie sich Farben aus RGB-Werten zusammensetzen, werde ich später – allerdings erst im zweiten Projekt – noch ausführlich erklären.

Vieles in diesem Abschnitt hatte weniger mit unserem konkreten Spiel, sondern mit der notwendigen, aber leider auch sehr technischen Initialisierung der Grafikumgebung zu tun. Ich habe Verständnis dafür, wenn Sie sich nicht übermäßig dafür interessieren, was zu tun ist, wenn jemand bei laufendem Programm den Grafikmodus umschaltet. Aber wir müssen uns damit beschäftigen, da ein Programm robust gegenüber solchen Eingriffen sein muss. Ich kann Ihnen aber versprechen, dass wir in den folgenden Abschnitten näher an der eigentlichen Spielidee arbeiten werden und auch sehr schnell sichtbare Fortschritte erzielen werden.

2.4.4 V04 Definition und Implementierung der Formen

Erstellen Sie zunächst als Ausgangspunkt für die Version `V04` eine Kopie der Version `V03`, aus der Sie alle auf die Version `V03` bezogenen Testfunktionen wieder entfernen.

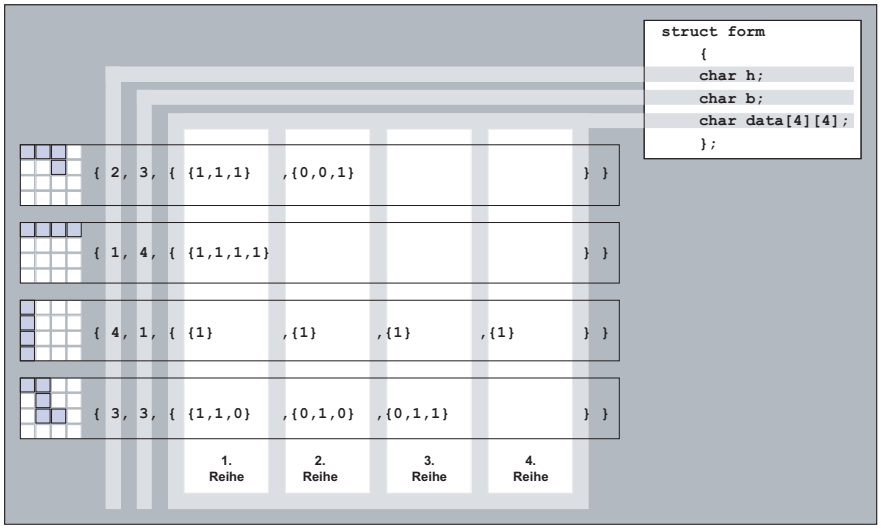
Der Spieler kann bei *Ultris* aus einer Gesamtzahl von 35 Steinen seine Spielsteine auswählen. Es macht bei einer solchen Anforderung keinen Sinn, alle möglichen Spielsteine individuell zu programmieren. Wir benötigen eine einheitliche Datenstruktur, in der wir alle benötigten Daten über die Spielsteine ablegen. Unter »Daten« wird dabei natürlich die äußere Form der Steine verstanden. Um Begriffsverwirrungen zu vermeiden, werde ich in diesem Zusammenhang immer von *Formen* sprechen – den Begriff Stein haben wir ja bereits durch die einzelnen Segmente, aus denen sich die Formen zusammensetzen, belegt. Beachten Sie daher die folgende Klarstellung der Begriffe:

	Ein Spielstein oder besser eine **Form**
	Ein Stein oder besser ein **Segment** einer Form

Zu den Daten einer Form gehören ihre Höhe und die Breite sowie Informationen über die vorhandenen Segmente. Zur Speicherung dieser Daten wählen wir die folgende Struktur, in der wir beliebige Figuren mit bis zu `4x4` Segmenten ablegen können:

```
struct form
    {
    char h;           // Hoehe
    char b;           // Breite
    char data[4][4]; // Vorhandene Segmente
    };
```

Die folgende Grafik zeigt anhand einiger Beispiele, wie diese Datenstruktur zur Abbildung von Formen mit Werten gefüllt werden muss:



Wir werden eine solche Struktur für alle 35 Formen in allen Drehvarianten[18] anlegen. Das sind 35*4 = 140 Datensätze. Ich zeige Ihnen hier nur die ersten drei Formen:[19]

```
// Form 1 in allen Drehvarianten
//
// *** *   *   *
//  *  ** *** **
//     *       *
const form s_01_1 = {2, 3, {{1,1,1}, {0,1,0}}};
const form s_01_2 = {3, 2, {{1,0}, {1,1}, {1,0}}};
const form s_01_3 = {2, 3, {{0,1,0}, {1,1,1}}};
const form s_01_4 = {3, 2, {{0,1}, {1,1}, {0,1}}};

// Form 2 in allen Drehvarianten
//
// **    * **   *
//  **  **  ** **
//      *      *
const form s_02_1 = {2, 3, {{1,1,0}, {0,1,1}}};
const form s_02_2 = {3, 2, {{0,1}, {1,1}, {1,0}}};
const form s_02_3 = {2, 3, {{1,1,0}, {0,1,1}}};
const form s_02_4 = {3, 2, {{0,1}, {1,1}, {1,0}}};
```

18 Die Drehvarianten könnte man natürlich auch berechnen.

19 Bei gewissen Formen sind manche Drehvarianten gleich. Ich habe allerdings auch in diesen Fällen aus Gründen der Systematik verschiedene Datenstrukturen angelegt.

```
// Form 3 in allen Drehvarianten
//
//  ** *   ** *
// **  ** **  **
//      *      *
const form s_03_1 = {2, 3, {{0,1,1}, {1,1,0}}};
const form s_03_2 = {3, 2, {{1,0}, {1,1}, {0,1}}};
const form s_03_3 = {2, 3, {{0,1,1}, {1,1,0}}};
const form s_03_4 = {3, 2, {{1,0}, {1,1}, {0,1}}};
```

Wichtig ist dabei, dass die Drehvarianten in der Systematik so angeordnet sind, dass ihre Abfolge einer Linksdrehung der Form entspricht.

Die komplette Formensammlung finden Sie in der Musterlösung zur Version 4. Wenn Sie die Formen selbst erstellen, sollten Sie darauf achten, dass die ersten sieben Formen die Standardformen aus dem originalen Tetris-Spiel sind. Ansonsten können Sie auch andere Formen als die von mir gewählten verwenden.

Alle Formen mit ihren Drehvarianten sammeln wir in einem zweidimensionalen Array:

```
const int anzahl_formen = 35;

const form *ultris_form[anzahl_formen][4] =
    {
        { &s_01_1, &s_01_2, &s_01_3, &s_01_4},
        { &s_02_1, &s_02_2, &s_02_3, &s_02_4},
        { &s_03_1, &s_03_2, &s_03_3, &s_03_4},
        ...
    };
```

In der ersten Dimension finden wir die Form, in der zweiten ihre Drehvariante. Konkret heißt das, dass wir auf die dritte Form in der zweiten Drehvariante (also auf `s_03_2`) in der folgenden Weise zugreifen können:[20]

```
ultris_form[2][1]
```

Als Ergebnis des Zugriffs erhalten wir einen Zeiger auf die gewünschte Form, sodass wir zum Zugriff auf die Daten dereferenzieren müssen:

```
ultris_form[2][1]->h
ultris_form[2][1]->b
ultris_form[2][1]->data[2][1]
```

20 Beachten Sie, dass in C die Zählung von Array-Elementen immer bei 0 beginnt.

Obwohl wir in dieser Version keinen Code, sondern nur Daten implementiert haben, ist es sehr sinnvoll, wieder eine Testfunktion zu erstellen. Schließlich müssen wir überprüfen, ob wir die Formen korrekt und im richtigen Drehsinn eingegeben haben.

Zum Test erweitern wir den Callback-Handler des Hauptfensters wie folgt:

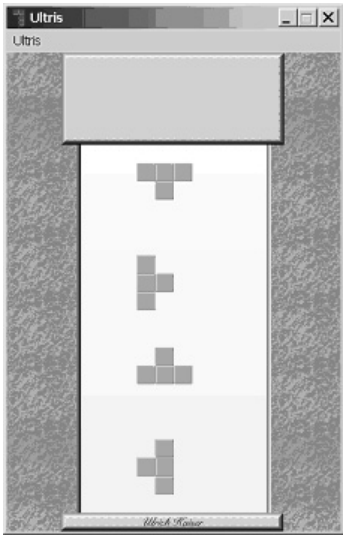
```
A  int testform = 0;

   LRESULT CALLBACK ultris_windowhandler(...)
      {
      switch(msg)
          {
      case WM_COMMAND:
          switch( LOWORD( wParam))
              {
              ...
B         case IDM_TEST: // Testcode
              testform = (testform + 1) % anzahl_formen;
              PostMessage( hWnd, WM_PAINT, 0, 0);
              return 0;
              }
          ...
C     case WM_PAINT:
          int variante;
          int z, s, x, y;
          const form *f;

          ultris_display.hintergrund();
          for( variante = 0; variante < 4; variante++)
              {
              z = 1 + 5*variante;
              s = 3;
              f = ultris_form[testform][variante];
              for( x = 0; x < f->b; x++)
                  {
                  for( y = 0; y < f->h; y++)
                      {
                      if( f->data[y][x])
                         ultris_display.fallstein(z+y,s+x,0);
                      }
                  }
              }
          ultris_display.present();
          break;
          }
      ...
      }
```

A: In einer globalen Variablen (`testform`) speichern wir die aktuell anzuzeigende Form. Initial ist das die Form `0`.

B: Immer wenn der Benutzer die Test-Taste (Strg-T) betätigt, gehen wir zur nächsten Form. Beim Erreichen der `35` (`anzahl_formen`) geht es wieder von vorn los. Anschließend veranlassen wir, dass unser Fenster neu gezeichnet wird. Dazu schicken wir die Message `WM_PAINT` an uns selbst. Danach geben wir die Kontrolle an das Betriebssystem zurück. Wir können aber sicher sein, dass der Window-Handler mit dem Kommando `WM_PAINT` erneut gerufen wird.

C: Wir werden aufgefordert, unser Fenster neu zu zeichnen. Dazu zeichnen wir zunächst den Hintergrund. Dann holen wir uns zu jeder Drehvariante die Daten und zeichnen die Form, indem wir in einer Doppelschleife alle vorhandenen Segmente in das Display blitten.

Kompilieren, linken und testen Sie jetzt diese Version. Beim Programmstart sollten Sie das folgende Bild sehen:



Gehen Sie mit der Test-Taste (Strg-T) Schritt für Schritt durch alle Formen, und überprüfen Sie, ob die vier Drehvarianten jeweils korrekt und in der richtigen Reihenfolge (Linksdrehung) angezeigt werden. Jetzt kann man schon erkennen, was es einmal werden soll.

Diesmal haben wir einen kleinen, aber wichtigen Entwicklungsschritt gemacht. Grundsätzlich sollten Sie immer in überschaubaren Entwicklungsschritten vorgehen, an deren Ende jeweils eine testfähige Version steht. Wenn Sie zu viele Ent-

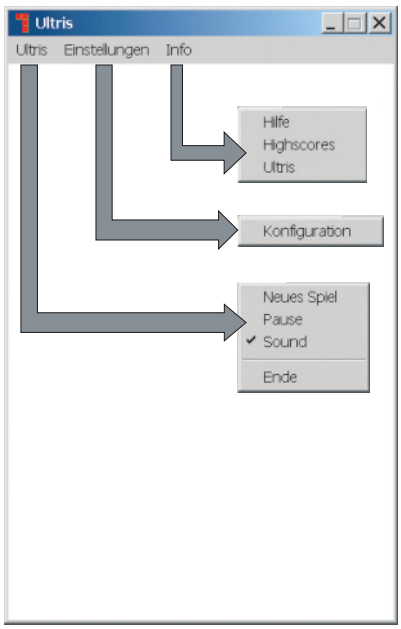
wicklungsschritte durchführen, bevor Sie wieder in eine Testphase gehen, kann es passieren, dass Sie neu auftretende Fehler nur sehr schwer lokalisieren können und eine gegebenenfalls eingeleitete Fehlentwicklung mit viel Aufwand zurückspulen müssen. Es ist durchaus sinnvoll, auch bei Projekten dieser Größenordnung ein Quellcode-Management-System wie *CVS* oder *SourceSafe* einzusetzen. Ein solches System hilft Ihnen dabei, Ihren Quellcode versionsbezogen zu verwalten, sodass Sie im Falle einer Fehlentwicklung leicht Wiederaufsetzpunkte in früheren Versionen rekonstruieren können.

2.4.5 V05 Das Menü und die einfachen Dialoge

Zu unserem Spiel müssen wir einen recht komplexen Konfigurationsdialog erstellen. Da ich nicht voraussetze, dass Sie die Windows-Programmierung beherrschen, muss ich Ihnen erkären, wie man einen Dialog erstellt und in eine Windows-Applikation einbaut. Um dies zunächst einmal anhand einfacher Beispiele zu üben, starten wir in dieser Kurseinheit mit zwei leicht zu erstellenden Dialogen aus dem *Info*-Menü. Zum eigentlichen Spiel tragen diese Dialoge wenig bei. Sie sind jedoch nützlicher Übungsstoff zum Thema Windows-Programmierung.

Bevor es losgeht, erstellen Sie wie üblich eine bereinigte neue Version (V05).

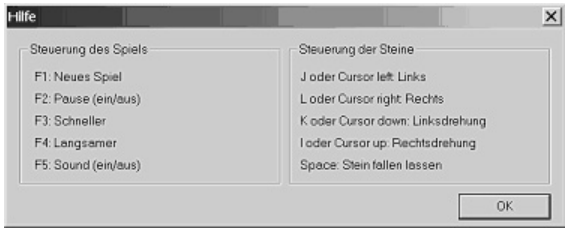
In einem ersten Schritt erweitern wir das *Ultris*-Menü um die erforderlichen Menüpunkte. Laden Sie dazu die Menü-Ressource in den Ressourcen-Editor, und komplettieren Sie das Menü entsprechend der folgenden Grafik:



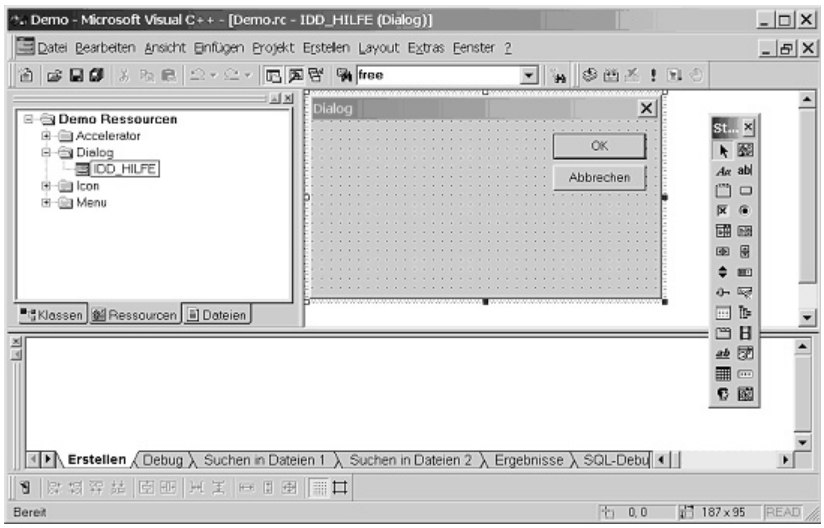

```
ID_INFO_HILFE
ID_INFO_HIGHSCORES
ID_INFO_ULTRIS

ID_EINSTELLUNGEN_KONFIGURATION

ID_ULTRIS_NEUESSPIEL
ID_ULTRIS_PAUSE
ID_ULTRIS_SOUND

IDM_EXIT
```

Das System generiert die den Menüpunkten zugeordneten Identifier aus den Bezeichnungen in den Menüs. Da wir die Identifier zum Zugriff im C-Code verwenden, sollten Sie prüfen, ob auch bei Ihnen die oben stehenden Identifier gewählt wurden. Bei Abweichungen sollten Sie die Bezeichnungen entsprechend ändern.

Wir starten mit der Erstellung des *Hilfe*-Dialogs, der die Tastenbelegung unseres Spiels anzeigt.



Die zugehörige Ressource können Sie fast ohne meine Hilfe erstellen. Zunächst müssen Sie einen neuen, leeren Dialog erstellen. Dazu wählen Sie im Menü *Einfügen* den Menüpunkt *Ressource* und selektieren im dann erscheinenden Unterdialog den Ressourcentyp *Dialog*. Wenn Sie anschließend den Button *Neu* betätigen, wird Ihrem Projekt eine neue Dialog-Ressource hinzugefügt. Als Identifier generiert Ihnen das System einen Namen wie `IDD_DIALOG1`. Bei einem Doppelklick auf den Dialog öffnet sich ein Eigenschaftsfenster, in dem Sie den Identifier aussagekräftiger in `IDD_HILFE` ändern sollten. Wenn alles geklappt hat, sollten Sie jetzt den neuen Dialog im Dialog-Editor sehen:

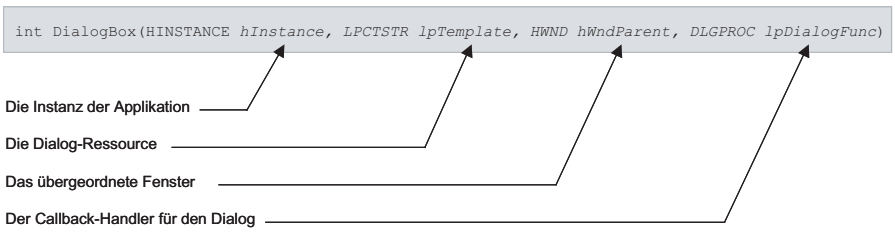


Überarbeiten Sie den Dialog entsprechend der Vorlage, indem Sie nicht benötigte Elemente löschen und neue Elemente von der Werkzeugleiste auf den Dialog ziehen. Testen Sie das Erscheinungsbild Ihres Dialogs über den Menüpunkt *Testen* im Menü *Layout*.

Wenn Sie die Dialog-Ressource fertig gestellt haben, können Sie sich wieder dem Programmcode zuwenden. Den Hilfe-Dialog müssen wir öffnen, wenn der Menüpunkt `ID_INFO_HILFE` gewählt wurde. Dazu erweitern wir den Callback-Handler des Hauptfensters in der folgenden Weise:

```
LRESULT CALLBACK ultris_windowhandler(...)
    {
    switch(msg)
        {
    case WM_COMMAND:
        switch( LOWORD( wParam))
            {
        ...
        case ID_INFO_HILFE:
            DialogBox( ultris_instance,
                       MAKEINTRESOURCE( IDD_HILFE),
                       ultris_window, hilfedialog);
            return 0;
        ...
            }
        break;
    ...
        }
    ...
    }
```

Der Aufruf der Funktion `DialogBox`



liest die zum Identifier `IDD_HILFE` gehörende Dialog-Ressource, erzeugt daraus den zugehörigen Dialog und bringt diesen auf den Bildschirm. Zur Bearbeitung der Benutzerinteraktion im Dialog übergeben wir die Adresse eines Callback-Handlers (`hilfedialog`), den wir allerdings noch implementieren müssen. Der Callback-Handler des Hilfedialogs hat die gleiche Schnittstelle wie der Callback-

Handler des Hauptfensters und wird immer dann gerufen, wenn auf eine Message reagiert werden muss, die den Hilfedialog betrifft. Da innerhalb des Dialogs aber nicht viel Interaktion stattfinden kann, sind dies nur Meldungen zum Schließen des Dialogs über *OK* beziehungsweise *Cancel*. Entsprechend einfach gestaltet sich der Code des Callback-Handlers:[21]

```
BOOL CALLBACK hilfedialog( HWND hwndDlg, UINT uMsg,
                           WPARAM wParam, LPARAM lParam)
    {
    switch (uMsg)
        {
        case WM_COMMAND:
            if((wParam == IDOK)||(wParam == IDCANCEL))
                {
                EndDialog(hwndDlg, wParam);
                return TRUE;
                }
        }
    return FALSE;
    }
```

Der Rückgabewert des Callback-Handlers sollte `TRUE` (oder `1`) sein, wenn die Message behandelt wurde. Wurde sie nicht behandelt, muss der Returnwert `FALSE` (oder `0`) sein. Insbesondere die letzte Anweisung, `return FALSE`, ist besonders wichtig, weil wir damit dem System mitteilen, welche Kommandos wir nicht behandelt haben. Das System führt dann die üblichen Standardbehandlungen durch. Würde an dieser Stelle `return TRUE` stehen, so würden unter Umständen wichtige Standardfunktionen im Dialog nicht ausgeführt werden, weil das System dann annehmen würde, dass wir diese Dinge bereits in unserem Callback-Handler erledigt haben.

Der zweite Dialog dieses Abschnitts ist der *Ultris*-Dialog aus dem Menü *Info*:

21 Eine Sprungleiste mit nur einem Sprungziel wirkt vielleicht etwas befremdlich, aber ich habe das bewusst mit Blick auf eine mögliche Erweiterung des Dialogs so angelegt.

Erstellen Sie für diesen Dialog eine neue Dialog-Ressource mit `IDD_ULTRIS` als Identifier. Neu ist hier, dass der Dialog eine Bitmap und einen zusätzlichen Button (*Weitere Informationen*) enthält. Die Bitmap[22] können Sie im Ressourcen-Editor erstellen oder auch aus einer externen Bitmap-Datei laden. Das schaffen Sie ohne weitere Unterstützung. Zur Darstellung der Bitmap-Ressource ziehen Sie dann einen Bildrahmen von der Werkzeugleiste auf den Ultris-Dialog und weisen dem Rahmen in seinem Eigenschaftsdialog die Bitmap zu.

Auch die Ressource für den Button *Weitere Informationen* ziehen Sie von der Werkzeugleiste auf den Dialog. Anschließend richten Sie ihn nach Ihren Wünschen aus. Wichtig ist, dass Sie dem Button über seinen Eigenschafts-Dialog den Identifier `IDC_WEITERE_INFO` zuordnen, damit die Einbindung im C-Code reibungslos funktioniert.

Den Dialog bringen Sie aus dem Windows-Handler `ultris_windowhandler` heraus mit dem bereits bekannten Aufruf der Funktion `DialogBox` als Reaktion auf das Kommando `ID_INFO_ULTRIS` zur Anzeige:

```
DialogBox( ultris_instance, MAKEINTRESOURCE( IDD_ULTRIS),
                                ultris_window, ultrisdialog);
```

Als Callback-Handler übergeben Sie die Funktion `ultrisdialog`, die Sie wie folgt ausprogrammieren:

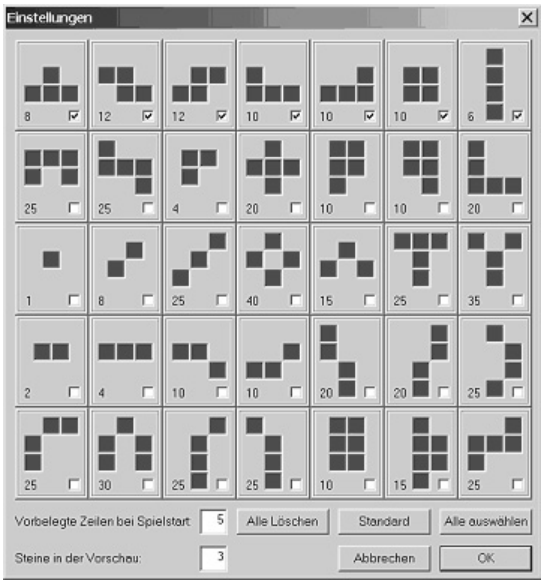
```
BOOL CALLBACK ultrisdialog( HWND hwndDlg, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam)
   {
   switch (uMsg)
      {
      case WM_COMMAND:
         if( wParam == IDC_WEITERE_INFO)
            {
            ShellExecute(NULL, "open",
                     "http://www-et.bocholt.fh-gelsenkirchen.de",
                                       NULL, NULL, SW_SHOWNORMAL);
            return TRUE;
            }
         else if((wParam == IDOK)||(wParam == IDCANCEL))
            {
            EndDialog(hwndDlg, wParam);
            return TRUE;
            }
      }
   return FALSE;
   }
```

22 Sie erstellen hier natürlich eine individuelle Bitmap und müssen nicht meine Bitmap nachzeichnen.

Neu ist hier nur die Behandlung des Buttons `IDC_WEITERE_INFO`. Wird dieser Button gedrückt, so starten wir über die Funktion `ShellExecute` den Internet-Browser auf dem Zielrechner und verzweigen auf die Homepage der Fachhochschule in Bocholt. Sie setzen hier natürlich die Adresse Ihrer eigenen Homepage ein, auf der der Benutzer des Spiels weitere Informationen zum Spiel erhält oder ein Update herunterladen kann.

2.4.6 V06 Der Konfigurationsdialog

Die letzte Kurseinheit hat uns in Bezug auf das eigentliche Spiel nicht viel weiter gebracht. Sie hat uns aber das Rüstzeug vermittelt, um einen wesentlich komplexeren Dialog in Angriff nehmen zu können – den Konfigurationsdialog:



Bevor wir diesen Dialog erstellen, müssen wir das Spiel zumindest so weit implementieren, dass die Daten dieses Dialogs vollständig in das Spiel übernommen werden können. Dazu erstellen wir zunächst einen Array (`aktive_formen`), in dem wir festhalten, welche der `35` Formen für das Spiel aktiviert sind.

```
char aktive_formen[anzahl_formen] =
    {
    1,1,1,1,1,1,1,
    0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,
    0,0,0,0,0,0,0
    };
```

Wir initialisieren diesen Array so, dass beim Programmstart die sieben Standard-Tetris-Formen aktiviert sind. Der Inhalt dieses Arrays kann dann im Laufe des Spiels über den Konfigurationsdialog geändert werden.

In einer zusätzlichen globalen Variablen (`anz_aktive_formen`) merken wir uns, wie viele Formen aktiv sind:

```
int anz_aktive_formen = 7;
```

Zu Spielbeginn sind das natürlich die sieben Standard-Tetris-Formen.

Wenn wir auf die Erstellung des Spiels zurückblicken, so haben wir bisher immer nur wichtige Grunddaten bereitgestellt. Wir haben die Sounds geladen und eine Schnittstelle zum Abspielen der Sounds programmiert. Wir haben die Grafiken geladen und Funktionen zur Darstellung der Grafiken erstellt. Wir haben die Daten für die Formen bereitgestellt. Jetzt beginnen wir, die Fäden zusammenzuknüpfen und das eigentliche Spiel zu erstellen. Dazu definieren wir eine neue, zunächst noch leere Klasse `ultris`

```
class ultris
    {
    private:

    public:

    };
```

und eine Instanz dieser Klasse – das Spiel:

```
ultris mein_spiel;
```

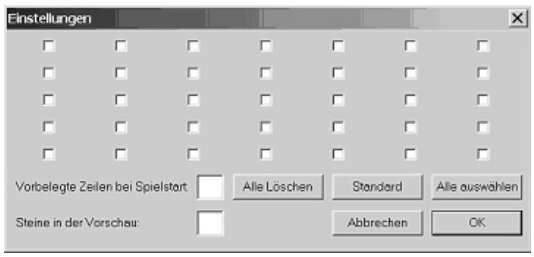
Im Folgenden werden wir uns überwiegend mit dieser Klasse beschäftigen und sie systematisch mit Funktionen und Daten anreichern, bis am Ende alle Spielfunktionen abgedeckt sind. Parallel dazu werden wir die neuen Funktionen immer in die Windows-Applikation integrieren.

Zu den Daten des Spiels gehören die Grundeinstellungen (Anzahl der vorbelegten Reihen, Anzahl der Formen in der Vorschau). Diese Daten müssen gelesen und im Rahmen der zulässigen Werte auch geschrieben werden können. Dadurch ergeben sich die folgenden Erweiterungen der Klasse `ultris`:

```
class ultris
    {
    private:
        int vorbelegung;
        int vorschau;
    public:
        ultris(){ vorbelegung = 0; vorschau = 1;}
        int get_vorbelegung() { return vorbelegung;}
        void set_vorbelegung( int v)
                {if( v < 0) v = 0; if( v > 20) v = 20;
                                              vorbelegung = v;}
        int get_vorschau() { return vorschau;}
        void set_vorschau( int v)
                    {if( v < 0) v = 0; if(v > 5) v = 5;
                                                vorschau = v;}
    };
```

Der Konstruktor erzeugt sinnvolle Initialwerte für die Grundeinstellungen. Über die `get`-Funktionen können die Grundeinstellungen gelesen, über die `set`-Funktionen können sie im Rahmen der zulässigen Grenzen geändert werden.

Die Erstellung der Dialog-Ressource für den Konfigurationsdialog ist zwar anspruchsvoller als die Erstellung aller bisher fertig gestellten Dialoge, aber mit etwas Übung bekommen Sie das schon hin. Sie können ja zunächst einen schlichten, aber funktionell ausreichenden Prototyp erstellen. Die Feinarbeit am Layout können Sie später durchführen. Durch die Verwendung von Ressourcen erzielen wir eine konsequente und saubere Trennung von Form (Ressource) und Funktionalität (Code) eines Dialogs. Insbesondere bedeutet dies, dass man das äußere Erscheinungsbild eines Dialogs nachträglich noch ändern kann, ohne in den Programmcode eingreifen zu müssen. Ein nüchternes, ausschließlich auf die Funktionalität ausgerichtetes Design des Konfigurationsdialoges könnte etwa wie folgt aussehen:

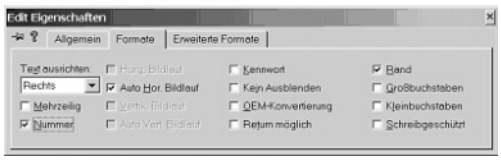


Wichtig für die Anbindung an den weiter unten beschriebenen Code ist, dass Sie die folgenden Identifier verwenden:

Ressource	Identifier	Wert
Konfigurationsdialog	`IDD_KONFIGURATION`	-
Button *Alle Löschen*	`IDC_LOESCHEN`	-
Button *Standard*	`IDC_STANDARD`	-
Button *Alle Auswählen*	`IDC_ALLE`	-
Eingabe Vorbelegung	`IDC_VORBELEGUNG`	-
Eingabe Vorschau	`IDC_VORSCHAU`	-
Checkbox Form 1	`IDC_CHECK1`	5000
Checkbox Form 2	`IDC_CHECK2`	5001
…	`...`	…
Checkbox Form 35	`IDC_CHECK35`	5034

Festgelegte Identifier benötigen wir natürlich nur für die Dialogelemente, die wir vom Programm aus bedienen wollen. Eine Besonderheit ist allerdings, dass wir für die Checkboxen fortlaufende Identifier-Werte benötigen, um sie im Programm durch einfache Verarbeitungsschleifen bearbeiten zu können. Man sollte in einer solchen Situation nicht die vom System generierten Werte, sondern eigene Werte für die Identifier verwenden. Man erreicht das, indem man im Eigenschaftsdialog im Feld *ID* den gewünschten Wert (z. B.: `IDC_CHECK1=5000`) zusätzlich einträgt:

Bei den Eingabefeldern sollten Sie als Format *Nummer* mit der Ausrichtung *Rechts* wählen:



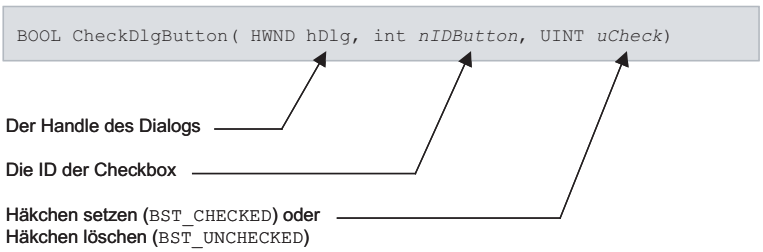
Sobald die Dialog-Ressource fertig ist, können wir mit der Implementierung der Funktionen beginnen. Aufgerufen wird der Konfigurationsdialog aus dem `ultris_windowhandler`, wenn das Kommando `ID_EINSTELLUNGEN_-KONFIGURATION` empfangen wird. Dazu fügen Sie im `ultris_windowhandler` an passender Stelle die folgenden, grau hinterlegten Zeilen ein:

```
LRESULT CALLBACK ultris_windowhandler(...)
    {
    switch(msg)
        {
    case WM_COMMAND:
        switch( LOWORD( wParam))
            {
        ...
        case ID_EINSTELLUNGEN_KONFIGURATION:
            DialogBox( ultris_instance,
                       MAKEINTRESOURCE( IDD_KONFIGURATION),
                       ultris_window, konfigurationsdialog);
            return 0;
        ...
            }
        break;
    ...
    }
```

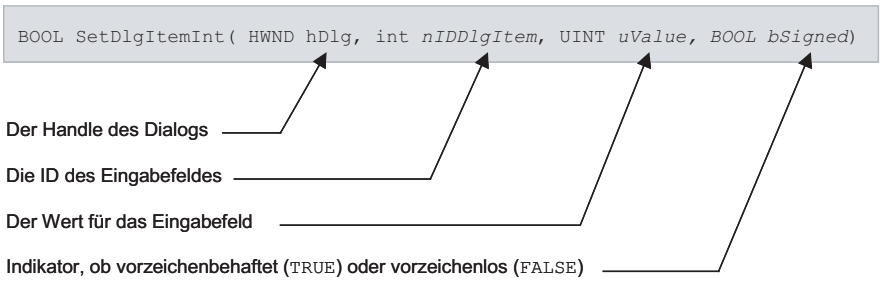
Den hier im Vorgriff bereits verwendeten Dialog-Handler `konfigurationsdialog` müssen wir noch mit der Ihnen bereits vertrauten Schnittstelle erstellen:

```
BOOL CALLBACK konfigurationsdialog( HWND hwndDlg, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam)
    {
    switch (uMsg)
        {
    case WM_INITDIALOG:
        ...
    case WM_COMMAND:
        ...
        }
    return FALSE;
    }
```

Die im Handler bereits vorgesehenen Fälle `WM_INITDIALOG` und `WM_COMMAND` müssen wir noch ausprogrammieren. Bei der Initialisierung des Dialogs setzen wir die Checkboxen mit der Windows-Funktion `CheckDlgButton`



entsprechend den Aktivierungsdaten im Array `aktive_formen`. Die Eingabefelder `IDC_VORBELEGUNG` und `IDC_VORSCHAU` werden mit Hilfe der Funktion `SetDlgItemInt`



initialisiert. Im Code sieht das dann wie folgt aus:

```
BOOL CALLBACK konfigurationsdialog(...)
    {
    int i;

    switch (uMsg)
        {
    case WM_INITDIALOG:
        for( i = 0; i < anzahl_formen; i++)
             CheckDlgButton( hwndDlg, 5000+i,
                          aktive_formen[i] ? BST_CHECKED :
                                             BST_UNCHECKED);
        SetDlgItemInt( hwndDlg, IDC_VORBELEGUNG,
                       mein_spiel.get_vorbelegung(), FALSE);
        SetDlgItemInt( hwndDlg, IDC_VORSCHAU,
                          mein_spiel.get_vorschau(),FALSE);
        return TRUE;
    case WM_COMMAND:
        ...
        }
    return FALSE;
    }
```

Im Falle eines Kommandos müssen wir drei Fälle unterscheiden:

- Der Benutzer hat den *OK*-Button betätigt.
- Der Benutzer hat den *Abbrechen*-Button betätigt.
- Der Benutzer hat einen der drei anderen Buttons betätigt.

Alle anderen Aktivitäten im Dialog interessieren uns nicht, obwohl wir uns auch darüber informieren lassen könnten. Warum sollten wir aber reagieren, wenn der Benutzer Checkboxen markiert oder Daten in die Eingabefelder schreibt, solange

unklar ist, ob er seine Eingaben am Ende mit *OK* bestätigt. Erst wenn der Benutzer den Dialog über den *OK*-Button verlässt, interessieren wir uns für die von ihm vorgenommenen Einstellungen.

Im Fall von `WM_COMMAND` implementieren wir daher die folgende Fallunterscheidung:

```
BOOL CALLBACK konfigurationsdialog(...)
    {
    ...
    switch (uMsg)
        {
    ...
    case WM_COMMAND:
        if(wParam == IDOK)
            {
            ...
            }
        if(wParam == IDCANCEL)
            {
            ...
            }
        if (HIWORD(wParam) == BN_CLICKED)
            {
            ...
            }
        }
    return FALSE;
    }
```

Die einzelnen Fälle müssen natürlich noch ausprogrammiert werden.

Im ersten Fall (`IDOK`) müssen wir die Einstellungen aus den Checkboxen herauslesen und im Array `aktive_formen` ablegen. Die Anzahl der aktiven Formen müssen wir dabei mitzählen (A):

```
A   for( i = 0, anz_aktive_formen = 0; i < anzahl_formen; i++)
      {
      aktive_formen[i] = 0;
      if(IsDlgButtonChecked( hwndDlg, 5000+i) == BST_CHECKED)
          {
          aktive_formen[i] = 1;
          anz_aktive_formen++;
          }
      }
B   if( anz_aktive_formen == 0)
      {
      aktive_formen[0] = 1;
```

```
   anz_aktive_formen = 1;
   }
C  mein_spiel.set_vorbelegung( GetDlgItemInt( hwndDlg,
                              IDC_VORBELEGUNG,NULL, FALSE));
   mein_spiel.set_vorschau( GetDlgItemInt( hwndDlg,
                                IDC_VORSCHAU,NULL, FALSE));
D  EndDialog(hwndDlg, wParam);
   return TRUE;
```

Hat der Benutzer keine Form ausgewählt, so wählen wir zwangsweise die erste Form (B). Anschließend werden die Daten für die Vorbelegung und die Vorschau in die internen Daten des Spiels übertragen (C). Dann wird der Dialog beendet und geschlossen (D).

Im zweiten Fall (`IDCANCEL`) müssen wir nur den Dialog schließen. Eine Übernahme der Daten kommt in dieser Situation nicht in Frage:

```
EndDialog(hwndDlg, wParam);
return TRUE;
```

Ist einer der Buttons *Alle Löschen*, *Standard* oder *Alle Auswählen* angeklickt worden (das ist der dritte Fall: `BN_CLICKED`), so müssen wir zunächst in einer weiteren Fallunterscheidung feststellen, welcher der drei Buttons gedrückt wurde. Je nach Button werden dann die erforderlichen Operationen auf den Checkboxen ausgeführt. Die Information, welcher der Buttons gedrückt wurde, finden wir im Lower-Word[23] des Parameters `wParam`. Bezüglich dieses Wertes treffen wir also eine Fallunterscheidung:

```
switch( LOWORD(wParam))
    {
case IDC_LOESCHEN:
    for( i = 0; i < anzahl_formen; i++)
        CheckDlgButton( hwndDlg, 5000+i, BST_UNCHECKED);
    return TRUE;
case IDC_ALLE:
    for( i = 0; i < anzahl_formen; i++)
        CheckDlgButton( hwndDlg, 5000+i, BST_CHECKED);
    return TRUE;
case IDC_STANDARD:
    for( i = 0; i < anzahl_formen; i++)
         CheckDlgButton( hwndDlg, 5000+i,
                         i < 7 ? BST_CHECKED:BST_UNCHECKED);
    return TRUE;
    }
```

23 Das sind die niederwertigen 16 Bit des insgesamt 32 Bit großen Parameters.

Wenn Sie die Einzelteile jetzt richtig zusammenfügen und anschließend kompilieren und linken, sollten Sie ein Spiel mit arbeitsfähigem Konfigurationsdialog erhalten. Anfangs sind die Default-Werte eingestellt. Benutzeränderungen werden gespeichert und beim nächsten Aufruf des Dialogs wieder angezeigt.

Falls dies noch nicht geschehen ist, sollten Sie dem Konfigurationsdialog jetzt auch optisch den letzten Schliff geben.

Wenn bei Ihnen jetzt die Vorstellung aufkommt, dass Sie es hier mehr mit einem Buch über Windows- als über Spieleprogrammierung zu tun haben, so kann ich Sie beruhigen. Natürlich ist und bleibt die Windows-Programmierung ein wichtiger Aspekt, aber bis auf den Highscore-Dialog sind jetzt alle Dialoge das Spiels fertig, und wir werden uns in den nächsten Kapiteln ganz auf den Spielablauf konzentrieren.

2.4.7 V07 Vorbesetzen des Spielfeldes

In dieser Kurseinheit beschäftigen wir uns mit den statischen Aspekten des Spiels, also mit den Dingen, die vor dem Spielstart zu tun sind, um das Spiel aufzubauen. Dynamische Aspekte, wie das Bewegen der Steine sind hier noch kein Thema.

Im folgenden Code-Fenster sehen Sie die Erweiterungen, die wir in dieser Kurseinheit an der Klasse `ultris` vornehmen wollen:

```
class ultris
    {
    private:
        ...
        int spielfeld[20][10];
        int fuellstand[20];
        int punktestand;
        void initialisieren();
        void vorbelegen();
        void display_spielfeld();
        void display_punktestand();
    public:
        ...
        int spiel_laeuft;
        ultris(){ srand( GetTickCount()); initialisieren();
                              vorbelegung = 0; vorschau = 1;}
        void start();
        void display();
    };
```

Zunächst sehen Sie hier eine Reihe von zusätzlichen, zumeist privaten Daten-Membern, die wir benötigen, um den Spielstand jederzeit konsistent und vollständig erfassen zu können:

Daten-Member	Bedeutung
`spielfeld`	Dies ist ein zweidimensionaler Array, der das Ultris-Spielfeld von 20x10 Feldern abbildet. Im Array-Element `spielfeld[z][s]` steht eine `1`, wenn in der Zeile `z` und der Spalte `s` ein Feldstein ist. Ansonsten steht dort eine `0`.[24]
`fuellstand`	Diese Zahl gibt den Füllstand in einer Zeile (Anzahl der Feldsteine in einer Zeile) an. Dieser Wert ist im Prinzip überflüssig, da er aus den Daten in `spielfeld` jederzeit berechnet werden kann. Er vereinfacht jedoch die Entscheidung, ob eine Reihe abgeräumt werden kann oder nicht.
`punktestand`	Dies ist der aktuelle Punktestand.
`spiel_laeuft`	Dies ist ein öffentlich zugängliches Daten-Member, über das das Spiel angehalten werden kann.

Die folgende Tabelle zeigt die neuen beziehungsweise geänderten Funktions-Member:

Funktions-Member	Bedeutung
`initialisieren`	Diese Funktion bringt das Spiel bei einem Neustart in einen konsistenten Initialzustand.
`vorbelegen`	In dieser Funktion wird das Spielfeld falls erforderlich mit Feldsteinen zufällig vorbelegt.
`display_spielfeld`	Anzeige des Spielfeldes im Grafikfenster
`display_punktestand`	Anzeige des Punktestandes im Grafikfenster
`ultris`	Der Konstruktor von `ultris` wird gegenüber V06 um die Initialisierung des Zufallszahlengenerators und um den Aufruf der Funktion `initialisieren` erweitert.
`start`	Start des Spiels
`display`	Anzeige des kompletten Spiels auf dem Bildschirm

Natürlich schauen wir uns auch noch den Code der einzelnen Funktionen an. Sie werden sehen, dass sich die Funktionen sehr einfach implementieren lassen, da wir mit den bereitgestellten Hilfsfunktionen gut vorgearbeitet haben.

Die Funktion `initialisieren` erzeugt ein leeres Spielfeld (alle Felder und Füllstände werden auf `0` gesetzt) und setzt zusätzlich die Daten-Member `spiel_laeuft` und `punktestand` auf sinnvolle Startwerte:

24 Beachten Sie, dass der fallende Stein hier nicht erfasst wird. Dieser gehört zu den dynamischen Aspekten des Spiels und befindet sich im Gegensatz zu den Feldsteinen ja auch nicht immer exakt in einer Zeile.

```
void ultris::initialisieren()
    {
    int z, s;

    for( z = 0; z < 20; z++)
        {
        fuellstand[z] = 0;
        for( s = 0; s < 10; s++)
            spielfeld[z][s] = 0;
        }
    spiel_laeuft = 0;
    punktestand = 0;
    }
```

In der Funktion `vorbelegen` werden zufällig Zeilen und Spalten im vorzubelegenden Bereich gewählt und die zugehörigen Felder mit einem Feldstein belegt. Der Füllstand der Zeilen wird dabei mitgezählt. Abschließend bekommt der Spieler einen Bonus von `vorbelegung*2500` Punkten:

```
void ultris::vorbelegen()
    {
    int z, s, i;

    for( i = 0; i < vorbelegung*5; i++)
        {
        z = 19-rand()%vorbelegung;
        s = rand()%10;
        if( !spielfeld[z][s])
            {
            spielfeld[z][s] = 1;
            fuellstand[z]++;
            }
        }
    punktestand = vorbelegung * 2500;
    }
```

Es kann bei dieser Vorgehensweise durchaus vorkommen, dass versucht wird, eine Position mehrmals zu belegen. Das soll uns aber nicht weiter stören. Wichtig ist nur, dass in dieser Situation der Füllstand der Zeile nicht hochgezählt wird. Nach dem Aufruf dieser Funktion sind maximal 50% der Felder in dem vorzubelegenden Teil des Spielfeldes mit Feldsteinen belegt. Wie viele Felder genau belegt sind, hängt von der Anzahl der Kollisionen ab, die bei den Belegungsversuchen aufgetreten sind.

Die `start`-Funktion kann noch nicht vollständig implementiert werden. Vorerst werden hier nur der Start-Sound gespielt und die bisher implementierten Vorbelegungen durchgeführt:

```
void ultris::start()
    {
    ultris_sounds.play( sound_start);
    initialisieren();
    vorbelegen();
    spiel_laeuft = 1;
    }
```

In den nachfolgenden Projektstufen werden wir diese Funktion noch weiter ausbauen.

Um das Spielfeld darzustellen, gehen wir durch alle Zeilen und Spalten. Finden wir einen Feldstein, so geben wir ihn über das `ultris_display` aus:

```
void ultris::display_spielfeld()
    {
    int z, s;

    for( z = 0; z < 20; z++)
        {
        for( s = 0; s < 10; s++)
            {
            if( spielfeld[z][s])
                ultris_display.feldstein( z, s);
            }
        }
    }
```

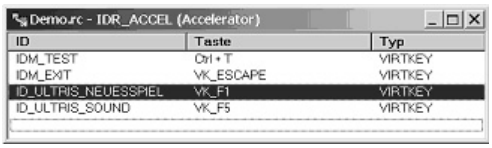
Zur Darstellung des Punktestandes auf der Abdeckung des Spiels zerlegen wir den Punktestand in einzelne Ziffern `z` und geben diese mit der vorbereiteten Funktion `ultris_display.ziffer` aus:

```
void ultris::display_punktestand()
    {
    int i, z, p;

    for( i = 5, p = punktestand; i >= 0; i--, p/= 10)
        {
        z = p % 10;
        ultris_display.ziffer( i, z);
        }
    }
```

Auch die Funktion `display` kann in dieser Projektphase noch nicht vollständig erstellt werden, da wir bisher nur die statischen Spielelemente (Hintergrund, Spielfeld mit Feldsteinen, Abdeckung und Punktestand) ausgeben können:

```
void ultris::display()
    {
    ultris_display.hintergrund();
    display_spielfeld();
    ultris_display.abdeckung();
    display_punktestand();
    ultris_display.present();
    }
```

Damit sind alle Erweiterungen der Klasse `ultris`, die wir uns für diese Projektstufe vorgenommen hatten, implementiert. Die Erweiterungen müssen wir jetzt noch in die Windows-Applikation integrieren. Dazu benötigen wir zunächst einen Akzellerator, über den wir ein neues Spiel (aus dem Menü *Ultris* oder über die Funktionstaste F1) starten können. Im Ressourcen-Editor für die Akzelleratoren-Tabelle fügen wir dazu die folgende Zeile ein:



`ID_ULTRIS_NEUESSPIEL` ist dabei der Identifier, den Sie seinerzeit dem Menüpunkt *Neues Spiel* im *Ultris*-Menü zugewiesen hatten. Diesem Menüpunkt ordnen wir jetzt zusätzlich den virtuellen Key `VK_F1` zu.

Wenn wir das Kommando `ID_ULTRIS_NEUESSPIEL` im `ultris_windowhandler` empfangen, können wir nicht unterscheiden, ob der Benutzer das Kommando über die Taste F1 oder den Menüpunkt *Neues Spiel* ausgelöst hat. Aber diese Unterscheidung ist für die Spielsteuerung auch irrelevant. Wichtig ist, dass wir im Callback-Handler auf das Kommando reagieren und das Spiel starten. Dazu müssen wir den folgenden Code hinzufügen:

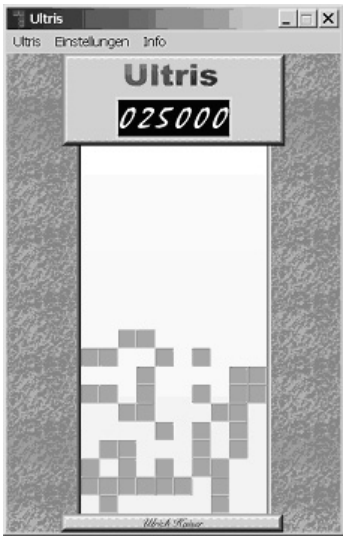
```
   LRESULT CALLBACK ultris_windowhandler( ...)
      {
      switch(msg)
          {
      case WM_COMMAND:
          switch( LOWORD( wParam))
              {
          ...
A         case ID_ULTRIS_NEUESSPIEL:
              mein_spiel.start();
              CheckMenuItem( ultris_menu, ID_ULTRIS_PAUSE,
                                                  MF_UNCHECKED);
              PostMessage( ultris_window, WM_PAINT, 0, 0);
              break;
```

```
        ...
            }
    ...
    case WM_PAINT:
B       mein_spiel.display();
        break;
        }
    ...
    }
```

Bei der Anforderung, ein neues Spiel zu starten (A), rufen wir die soeben erstellte Startfunktion (`mein_spiel.start`), löschen ein gegebenenfalls vorhandenes Pause-Häkchen im *Ultris*-Menü und schicken uns selbst eine `WM_PAINT`-Message. Auf diese Message reagieren wir dann an der bereits vorgesehenen Stelle (B) mit dem Neuzeichnen der Spielgrafik.

Das war schon alles, was wir zur Integration zu tun hatten. Sie können jetzt wieder kompilieren, linken und testen.

Immerhin erkennt man jetzt schon, was es einmal werden soll. Wenn Sie das Programm starten, im Konfigurationsdialog beispielweise `10` vorzubelegende Zeilen einstellen und anschließend das Spiel über F1 oder den Menüpunkt *Neues Spiel* starten, so sollten Sie in etwa das folgende Spielfeld sehen:



Die genaue Belegung der Felder hängt natürlich von den Werten ab, die der Zufallszahlengenerator liefert. Sie ist von Spiel zu Spiel verschieden.

2.4.8 V08 Erzeugen der Formen und Anzeigen der Vorschau

Jetzt geht es an die Erzeugung der fallenden Formen. Jede Form hat einen bestimmten Wert, der beim Erzeugen der Form dem Punktekonto des Spielers gutgeschrieben wird. Dazu benötigen wir die Information über den Punktwert der einzelnen Formen, die wir in dem folgenden Array ablegen:

```
const int formwert[anzahl_formen] =
    {
     8,12,12,10,10,10, 6,
    25,25, 4,20,10,10,20,
     1, 8,25,40,15,25,35,
     2, 4,10,10,20,20,25,
    25,30,25,25,10,15,25
    };
```

Da das Spiel eine Vorschau auf bis zu fünf Formen haben soll, müssen wir zusätzlich zur aktuellen Form immer die nächsten fünf Formen im Voraus berechnen und abspeichern. Dazu legen wir einen Array für die Daten von sechs Formen an. Der erste Eintrag bezieht sich auf die aktuelle Form, die fünf anderen beziehen sich auf die nachfolgenden Formen. Für jede Form müssen wir ihre laufende Nummer (Index) und ihre Drehvariante in diesem Array speichern. Wir erweitern die Klasse `ultris` um eine Datenstruktur (A) und den erforderlichen Array (B):

```
   class ultris
      {
A     struct formwahl
          {
          int ix; // Index der Form
          int dv; // Drehvariante der Form
          };
      private:
          ...
B         formwahl formen[6]; // 0 aktuelle Form, 1-5 Vorschau
C         int zeile;
          int spalte;
          int offset;
D         int sichtbar;
E         int zeige_dyn;
      public:
          ...
      };
```

Für die aktuell fallende Form benötigen wir zusätzlich die Zeile und die Spalte, in der sie sich jeweils befindet und das Offset in Pixeln, um das sie sich bereits weiterbewegt hat. Diese Daten finden Sie oben unter Punkt C. Solange die fallende

Form durch die Abdeckung verdeckt ist, zeigen wir die Form und bis zu vier weitere Formen in der Vorschau an. Sobald die fallende Form zum ersten Mal vollständig sichtbar ist, schalten wir die Vorschau um und zeigen die auf die Form folgenden Steine in der Vorschau. Zum Umschalten der Vorschau verwenden wir die Member-Variable `sichtbar` (D), die anfangs den Wert `0` hat und auf `1` gesetzt wird, sobald die fallende Form voll sichtbar ist. Wenn am Anfang kein Spiel gestartet ist oder der Spieler sich zwischen zwei Spielen befindet, sollen die dynamischen Daten (fallende Form und Vorschau) nicht angezeigt werden. Wir steuern dies über die Member-Variable `zeige_dyn` (E).

Neben den oben diskutierten Daten benötigen wir eine Reihe von neuen Member-Funktionen:

```
class ultris
    {
    ...
    private:
        ...
        void display_vorschau();
        void display_form();
        void naechste_form();
        const form *aktuelle_form()
                        {return ultris_form[formen[0].ix]
                                           [formen[0].dv];}
    public:
        ...
        void neue_form();
    };
```

Diese Funktionen haben im Einzelnen die folgende Bedeutung:

Member-Funktion	Bedeutung
`display_vorschau`	Anzeige der Vorschau am rechten Spielfeldrand
`display_form`	Anzeige der fallenden Form
`naechste_form`	Bereitstellen der nächsten Form
`aktuelle_form`	Liefert einen Zeiger auf die aktuell fallende Form
`neue_form`	Bereitstellen einer neuen Form mit Initialisierung ihrer Position[25]

25 Diese Funktion liegt nur deshalb im öffentlichen Bereich, damit sie für Tests zugänglich ist. Nachdem Sie die Tests in diesem Abschnitt abgeschlossen haben, können Sie die Funktion in den privaten Bereich verlegen.

Einige der bereits vorhandenen Funktionen müssen in diesem Abschnitt geringfügig erweitert werden. In der Funktion `initialisieren` fügen Sie die Zeile

```
zeige_dyn = 0;
```

ein, damit die dynamischen Elemente zunächst nicht angezeigt werden.

Jetzt implementieren wir die Funktion `naechste_form`:

```
    void ultris::naechste_form()
       {
       int z, i, ix;

A      for( i = 0; i < 5; i++)
           formen[i] = formen[i+1];
B      z = rand()%anz_aktive_formen + 1;
       for( ix = -1; z; z--)
           {
           while( !aktive_formen[++ix])
               ;
           }
       formen[5].ix = ix;
C      formen[5].dv = rand()%4;
       }
```

In dieser Funktion lassen wir zunächst alle Formen im Array `formen` um einen Platz aufrücken (A). Die erste Vorschauform wird damit zur aktuellen Form, die zweite Vorschauform zur ersten usw. Dann berechnen wir eine Zufallszahl `z` zwischen `1` und der Anzahl der aktiven Formen (B). Anschließend durchlaufen wir den Array der Formen, um die `z`-te aktive Form zu ermitteln. Der Index dieser Form steht nach dem Durchlaufen der Schleife in der Variablen `ix` und wird in die Datenstruktur für die fünfte, unter A frei gewordene Vorschauform geschrieben. In Punkt C wählen wir dann noch zufallsgesteuert eine Drehvariante für die Vorschauform aus.

Die Funktion `naechste_form` aktualisiert nur den Vorschaubereich, ohne die neue, aktive Form in ihre Startposition zu bringen. Das holen wir in der Funktion `neue_form` nach:

```
    void ultris::neue_form()
       {
A      naechste_form();
B      zeile = -aktuelle_form()->h;
       offset = 0;
       spalte = 5-(aktuelle_form()->b)/2;
       sichtbar = 0;
C      punktestand += formwert[formen[0].ix];
       }
```

Nachdem die nächste Form worden erzeugt ist (A), wird die neue Form so positioniert, dass sie mit der Unterkante noch so eben durch die Abdeckung verborgen wird

```
zeile = -aktuelle_form()->h
```
und
```
offset = 0
```

und in etwa mittig

```
spalte = 5-(aktuelle_form()->b)/2)
```

ausgerichtet ist (B). Der Punktestand erhöht sich bei jedem neuen Stein um den Wert der Form (C).

Die Trennung der Funktionen `naechste_form` und `neue_form` habe ich durchgeführt, damit man beim Start des Spiels zunächst eine Warteschlange von fünf Formen in der Vorschau aufbauen kann (A), bevor man dann die erste Form für das Spiel bereitstellt (B):

```
   void ultris::start()
      {
      ...
A     naechste_form();
      naechste_form();
      naechste_form();
      naechste_form();
      naechste_form();
B     neue_form();
      zeige_dyn = 1;
      }
```

Jetzt müssen wir die bereitgestellten Formen noch auf den Bildschirm bringen. Dazu dienen die Funktionen `display_vorschau` und `display_form`. Bevor wir die Vorschau ausgeben, prüfen wir zunächst, ob die Form bereits einmal vollständig sichtbar war.[26] War sie bereits sichtbar, so geben wir in der Vorschau die auf die aktuelle Form folgenden Steine aus. War das noch nicht der Fall, beginnt die Vorschau mit der aktuellen Form:

26 Diese Formulierung mag Ihnen merkwürdig erscheinen, da die Form doch stetig nach unten fällt. Es ist aber möglich, dass die Form vollständig sichtbar ist und durch eine (noch nicht implementierte) Drehung wieder teilweise unter der Abdeckung verschwindet. Dann fahren wir natürlich nicht die Vorschau zurück. Entscheidend ist also, ob die Form bereits einmal vollständig sichtbar war.

```
void ultris::display_vorschau()
    {
    int z, s, p, y;
    const form *f;

    if( !sichtbar)
        sichtbar = (zeile >= 0);

    for( y = 0, p = sichtbar; y < vorschau; p++, y++)
        {
        f = ultris_form[formen[p].ix][formen[p].dv];
        for( z = 0; z < f->h; z++)
            {
            for( s = 0; s < f->b; s++)
                {
                if( f->data[z][s])
                    ultris_display.prevstein(y, z, s, f->b,
                                                    f->h);
                }
            }
        }
    }
```

Um die fallende Form auf den Bildschirm zu bringen, müssen wir die aktuelle Form Segment für Segment in das Display blitten:

```
void ultris::display_form()
    {
    int z, s;

    for( z = 0; z < aktuelle_form()->h; z++)
        {
        for( s = 0; s < aktuelle_form()->b; s++)
            {
            if( aktuelle_form()->data[z][s])
                {
                ultris_display.fallstein( zeile+z, spalte+s,
                                                    offset);
                }
            }
        }
    }
```

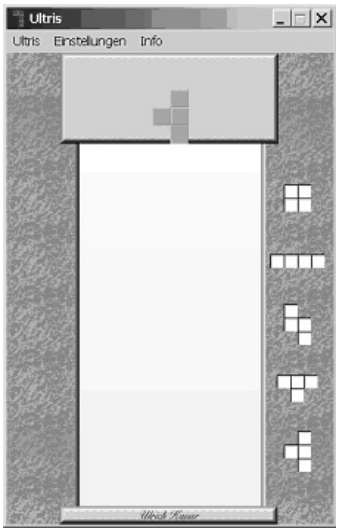
Falls die dynamischen Elemente angezeigt werden sollen, rufen wir die entsprechenden `display`-Funktionen auf:

```
void ultris::display()
    {
    ultris_display.hintergrund();
    display_spielfeld();
    if( zeige_dyn)
        {
        display_vorschau();
        display_form();
        }
//    ultris_display.abdeckung();
//    display_punktestand();
    ultris_display.present();
    }
```

Ich habe in dieser Funktion die Aufrufe zum Zeichnen der Abdeckung und des Punktestandes zu Testzwecken deaktiviert. Ohne diese Deaktivierung wären die Formen in ihrer Ausgangsstellung von der Abdeckung und dem Zählwerk verdeckt. Da wir aber die korrekte Erzeugung der Formen in Verbindung mit der Vorschau testen wollen, habe ich durch die Kommentierungen sozusagen das Gehäuse des Spiels geöffnet, damit wir hineinsehen können. Nach Abschluss der Tests sollten Sie das Gehäuse wieder »zuschrauben«. Für die Tests erweitern wir den `ultris_windowhandler` so, dass beim Drücken der Test-Taste (Strg-T) eine neue Form erzeugt wird:

```
LRESULT CALLBACK ultris_windowhandler(...)
    {
    switch(msg)
        {
    case WM_COMMAND:
        switch( LOWORD( wParam))
            {
        ...
        case IDM_TEST: // Testcode
            mein_spiel.neue_form();
            PostMessage( ultris_window, WM_PAINT, 0, 0);
            return 0;
            }
        break;
     ...
        }
    ...
    }
```

Jetzt können wir wieder testen. Bei einer Einstellung von fünf Formen in der Vorschau sollte das Spiel jetzt so aussehen:



Mit jeder Betätigung von Strg-T verschwindet die unterste Form aus der Vorschau, und die zweite von unten tritt an ihre Stelle und geht gleichzeitig an den Start. Von oben rückt in der Vorschau jeweils eine neue Form nach.

Vergessen Sie nicht, nach Abschluss der Tests das »Gehäuse wieder zu schließen«.

2.4.9 V09 Fallende Formen, Geschwindigkeitsregelung und Timing

In dieser Projektphase kommt endlich Bewegung ins Spiel. Wir werden die im letzten Abschnitt erzeugten Formen jetzt herunterfallen lassen, bis sie auf ein Hindernis stoßen. Geschwindigkeit und Zeit sind jetzt ein wichtiges Thema. Wir erweitern die Klasse `ultris` daher um zwei weitere private Daten-Member und sieben neue Funktions-Member:

```
class ultris
    {
    ...
    private:
        ...
        int speed;
        DWORD time;
        int onestep();
        int blockiert();
        ...
    public:
        ...
        int step();
        void down();
        void reset_timer(){time = timeGetTime();}
```

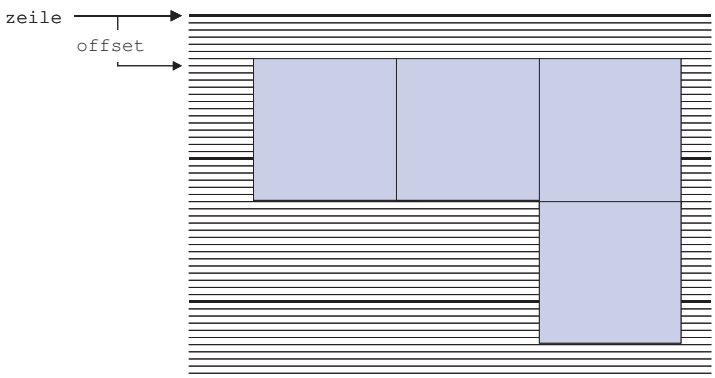
```
        void schneller() { speed += 50;}
        void langsamer() { if( speed > 50) speed -= 50;}
        ...
    };
```

In der Variablen `speed` speichern wir die aktuelle Fallgeschwindigkeit der Formen. Die Variable `time` verwenden wir, um uns den Zeitpunkt der letzten Aktualisierung des Spiels zu merken. Anhand der seit der letzten Aktualisierung verflossenen Zeit und der Geschwindigkeit können wir dann ausrechnen, um wie viele Pixel eine Form vorangerückt werden muss. Gleichzeitig habe ich die Funktion `reset_timer` eingefügt, die die interne Variable `time` auf die aktuelle Zeit (Systemzeit in Millisekunden) setzt. Die ebenfalls neuen Funktionen `schneller` und `langsamer` erklären sich von selbst. Die Funktionen `onestep`, `blockiert`, `down` und `step` werden weiter unten besprochen.

Beim Start des Spiels initialisieren wir die Zeitreferenz mit der Funktion `reset_timer` und setzen gleichzeitig die Anfangsgeschwindigkeit auf `100`:

```
void ultris::start()
    {
    ...
    reset_timer();
    speed = 100;
    }
```

Die aktuell fallende Form befindet sich immer in einer Zeile und fällt Pixel für Pixel von oben herab. Dazu wird ein so genanntes Offset zum Zeilenzähler hinzugezählt. Da eine Zeile `20` Pixel hoch ist, kann das Offset einen Wert zwischen `0` und `19` haben. Ist das Offset `19`, so wird im nächsten Schritt der Zeilenzähler um eins erhöht und das Offset wieder auf `0` gesetzt. Immer wenn das Offset den Wert 0 hat, befinden wir uns also exakt im Raster des Spiels. In allen anderen Fällen liegt eine Verschiebung gegenüber dem Raster vor:



In der Funktion `onestep` versuchen wir die Form ein Pixel nach unten voranzuschieben und melden im Returncode, ob dies geklappt hat (`1`) oder nicht (`0`):

```
    int ultris::onestep()
       {
       if( offset)
           {
A          if( offset < 19)
               offset++;
           else
               {
               offset = 0;
               zeile++;
               }
           return 1;
           }
       else
           {
B          if( blockiert())
               {
               ultris_sounds.play( sound_down); // vorlaeufig
               neue_form();                     // vorlaeufig
               return 0;
               }
           offset++;
           return 1;
           }
       }
```

Wenn das `offset` nicht `0` ist (A), so befinden wir uns zwischen zwei Zeilen. Ist das `offset` dabei kleiner als `19`, so können wir es bedenkenlos weiter erhöhen. Hat das `offset` den Wert `19`, so gehen wir in die nächste Zeile. Das `offset` ist dann wieder `0`. Ist das `offset` andererseits beim Eintritt in diese Funktion `0`, so müssen wir prüfen, ob unterhalb unserer Form Platz ist, um noch eine Zeile weiterzugehen. Wir testen dies mit der Funktion `blockiert`, die wir noch implementieren müssen. Nur wenn Platz ist, rücken wir weiter voran, ansonsten spielen wir den Sound der aufprallenden Form und initialisieren die nächste Form. Die beiden letzten Befehle sind nur vorläufig an dieser Stelle eingesetzt. Später müssen wir an dieser Stelle die Form zu Feldsteinen auflösen und die dabei gegebenenfalls entstehenden vollständigen Reihen abräumen. Im Moment lassen wir aber die Formen, wenn sie nicht mehr weiterkommen, einfach verschwinden und starten mit einer neuen Form.

In der Funktion `blockiert` testen wir, ob unterhalb der einzelnen Segmente der aktuellen Form noch Platz ist:

```
int ultris::blockiert()
    {
    int z, s, zz;

    for( z = 0; z < aktuelle_form()->h; z++)
        {
        for( s = 0; s < aktuelle_form()->b; s++)
            {
            if( aktuelle_form()->data[z][s])
                {
                zz = zeile + z + 1;
                if( (zz >= 20) ||
                                ((zz >= 0) &&(spielfeld[zz]
                                        [spalte+s])))
                    return 1;
                }
            }
        }
    return 0;
    }
```

Ist die Form nach unten blockiert, so ist der Returnwert `1`, andernfalls `0`. Aufbauend auf der Hilfsfunktion `onestep` implementieren wir zwei weitere Funktionen (`down` und `step`).

In der Funktion `down` lassen wir einen Stein herunterfallen, indem wir so lange `onestep` aufrufen, wie die Form nicht blockiert ist:

```
void ultris::down()
    {
    while( onestep())
        ;
    }
```

Diese Funktion werden wir auslösen, wenn der Benutzer die Leertaste betätigt.

Um das Spiel ohne Benutzereingriffe ablaufen zu lassen, werden wir die nachfolgend beschriebene Funktion `step` regelmäßig aus der Hauptverarbeitungsschleife aufrufen:

```
    int ultris::step()
       {
       DWORD now;
       int pixel;
       int diff;
A      now = timeGetTime();
       diff = (now-time)*speed;
       pixel = diff/5000;
       if( !pixel)
           return 0;
B      time = now - diff%5000/speed;
C      for( ; pixel && onestep(); pixel--)
           ;
       return 1;
      }
```

Zunächst (A) berechnen wir in dieser Funktion aus der abgelaufenen Zeit (`now-time`) und der Geschwindigkeit, um wie viele Pixel (`pixel`) die Form nach unten fallen muss. Kommt bei dieser Berechnung `0` heraus, so melden wir an das rufende Programm zurück, dass kein Schritt ausgeführt werden musste. Unter Punkt B setzen wir dann die Zeitreferenz auf die aktuelle Zeit und setzen dann aber noch die Zeit um die Milisekunden zurück, die wegen der obigen Integer-Division durch `5000` als Divisionsrest unberücksichtigt geblieben sind. Im Punkt C führen wir dann die zuvor berechnete Anzahl von Schritten aus, sofern unsere Form nicht vorher blockiert. Abschließend melden wir zurück, dass Schritte durchgeführt wurden (Returncode `1`).

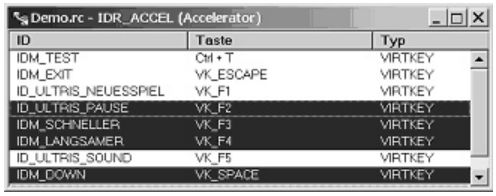
Im Callback-Handler unseres Fensters nehmen wir einige Erweiterungen vor, um zu verhindern, dass das Spiel, während wir Menüs oder Dialoge bedienen, einfach weiterläuft. Dazu müssen wir immer nur, wenn wir aus einem Dialog oder Menü zurückkommen, den Timer zurücksetzen, um dem Spiel vorzugaukeln, dass keine Zeit vergangen ist. Wir machen dies an den nachfolgend markierten Stellen:

```
LRESULT CALLBACK ultris_windowhandler(…)
    {
    switch(msg)
        {
    case WM_COMMAND:
        switch( LOWORD( wParam))
            {
            ...
        case ID_INFO_HILFE:
            DialogBox( ...);
            mein_spiel.reset_timer();
            return 0;
```

```
        case ID_INFO_ULTRIS:
            DialogBox( ...);
            mein_spiel.reset_timer();
            return 0;
        case ID_EINSTELLUNGEN_KONFIGURATION:
            DialogBox( ...);
            mein_spiel.reset_timer();
            return 0;
        ...
            }
        break;
    ...
    case WM_EXITMENULOOP:
    case WM_EXITSIZEMOVE:
        mein_spiel.reset_timer();
        break;
    ...
        }
    ...
    }
```

Neu sind hier die Einsprungpunkte `EXITMENULOOP` und `EXITSIZEMOVE`, die immer angesprungen werden, wenn wir aus einem Menü oder einer Verschiebeoperation des Hauptfensters zurückkommen. Da solche Operationen aus der Sicht des Spiels ebenfalls zeitlos sein sollen, stellen wir auch hier die Uhr zurück.

Die Funktionen `schneller`, `langsamer` und `down` wollen wir über Funktionstasten beziehungsweise die Leertaste ansprechen. Dazu müssen wir zunächst wieder Akzelleratoren einrichten:



Gleichzeitig habe ich auch einen Akzellerator für den Pause-Befehl (Menü oder F2) eingerichtet. Zu diesen Akzelleratoren müssen wir noch den entsprechenden Code erstellen:

```
LRESULT CALLBACK ultris_windowhandler(...)
   {
   switch(msg)
      {
   case WM_COMMAND:
      switch( LOWORD( wParam))
         {
         ...
      case ID_ULTRIS_PAUSE:
         mein_spiel.spiel_laeuft = !mein_spiel.spiel_laeuft;
         CheckMenuItem( ultris_menu, ID_ULTRIS_PAUSE,
                mein_spiel.spiel_laeuft ?
                           MF_UNCHECKED:MF_CHECKED);
         return 0;
      case IDM_DOWN:
         if( mein_spiel.spiel_laeuft)
            mein_spiel.down();
         return 0;
      case IDM_SCHNELLER:
         mein_spiel.schneller();
         return 0;
      case IDM_LANGSAMER:
         mein_spiel.langsamer();
         return 0;
      ...
         }
      break;
   ...
      }
   ...
   }
```

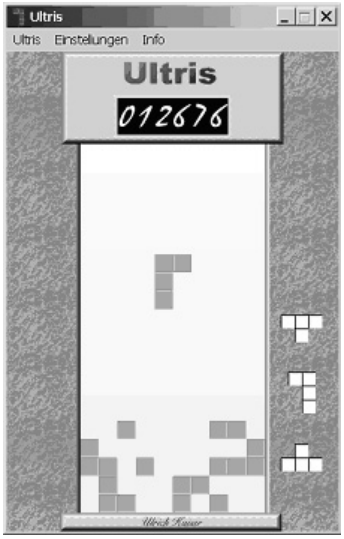
Diese Erweiterungen verstehen Sie sicherlich auch ohne weitere Hinweise.

Trotz aller bisher angesprochenen Erweiterungen »läuft« das Spiel bisher noch nicht selbstständig, sondern reagiert immer nur auf Benutzereingaben. Um das Spiel zum Laufen zu bringen, müssen wir aus der Hauptverarbeitungsschleife regelmäßig ohne vorhergehende Benutzereingaben die Funktion `step` aufrufen:

```
   int APIENTRY WinMain( ...)
      {
      ...
      while( TRUE)
          {
          if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE))
              ...
          else
              {
              HRESULT hr;
              hr = ultris_display.cooperative();
              if( hr < 0)
                  ...
              else
                  {
A                 if( mein_spiel.spiel_laeuft)
                      {
                       if( mein_spiel.step())
                          mein_spiel.display();
                      }
                  else
                      {
B                     WaitMessage();
                      mein_spiel.reset_timer();
                      }
                  }
              }
          }
      }
```

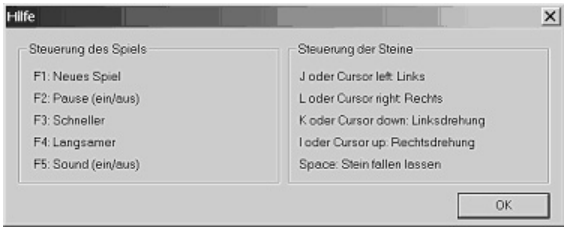
Wenn das Spiel läuft (A), führen wir einen Schritt aus und aktualisieren bei Bedarf das Display. Wenn das Spiel nicht läuft, haben wir nichts zu tun, da ja in dieser Situation auch keine Message vorliegt. Wir legen uns dann schlafen, bis für uns wieder eine Message vorliegt (`WaitMessage`).

Jetzt können Sie wieder ein lauffähiges Programm erzeugen. Wenn Sie das Spiel starten, kommen ständig neue Formen unter der Abdeckung hervor und fallen herab, bis sie auf ein Hindernis treffen. Dort verschwinden sie spurlos, und die nächste Form fällt herab:

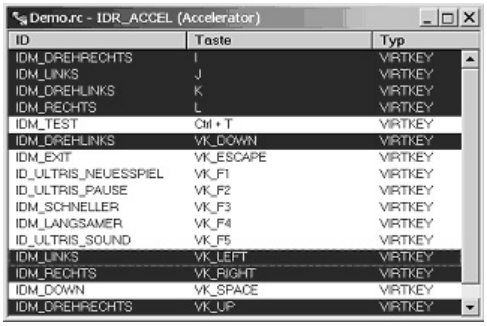


2.4.10 V10 Bewegen der Formen und Kollisionserkennung

In diesem Abschnitt geht es um die Steuerung der Formen mit den Cursortasten beziehungsweise mit den Tasten J, K, L und I. Zur Erinnerung sehen Sie hier noch einmal die Übersicht über alle Steuerbefehle:



Zur Steuerung der Steine erstellen wir mit dem Ressourcen-Editor acht weitere Akzelleratoren:



Jeweils zwei Akzelleratoren haben den gleichen Identifier, weil sie ja auch die gleiche Funktion auslösen sollen.

Die Klasse `ultris` erweitern wir um Methoden zum Rechts-Links-Bewegen beziehungsweise zum Rechts-Links-Drehen:

```
class ultris
    {
    ...
    private:
        ...
    public:
        ...
        void bewegen( int dir);
        void drehen(int dir);
    };
```

Die Dreh- beziehungsweise Bewegungsrichtung steuern wir in beiden Funktionen über den Parameter `dir`. Ein positiver Wert `1` bedeutet eine Drehung beziehungsweise Bewegung nach rechts, ein negativer Wert `–1` gibt die umgekehrte Richtung an.

Drehen ist eine etwas anspruchsvollere Aufgabe als Bewegen. Wir implementieren daher zunächst die Bewegung:

```
    void ultris::bewegen( int dir) //-1 eins nach links +1
    eins nach rechts
       {
       int z, s, neue_spalte;
       const form *f;

A      f = aktuelle_form();
       neue_spalte = spalte + dir;
B      if( neue_spalte < 0)
           return;
       if( neue_spalte + f->b > 10)
           return;
C      for( z = 0; z < f->h; z++)
           {
D          if( zeile+z < 0)
               continue;
E          for( s = 0; s < f->b; s++)
               {
F              if( f->data[z][s])
                   {
G                  if((offset < 16) && spielfeld[zeile+z]
                                                [neue_spalte+s])
                       return;
```

```
H                   if((offset > 0) && spielfeld[zeile+z+1]
                                                [neue_spalte+s])
                        return;
                    }
                }
            }
I       spalte = neue_spalte;
        ultris_sounds.play( sound_move);
        }
```

A: Wir holen uns die aktuelle Form und berechnen anhand der Richtung (`dir`) die Spalte, in die die Form bewegt werden soll (`neue_spalte`).

B: Würde die Bewegung dazu führen, dass die Form links oder rechts aus dem Spielfeld gerät, so brechen wir die Funktion ab, ohne eine Bewegung ausgeführt zu haben.

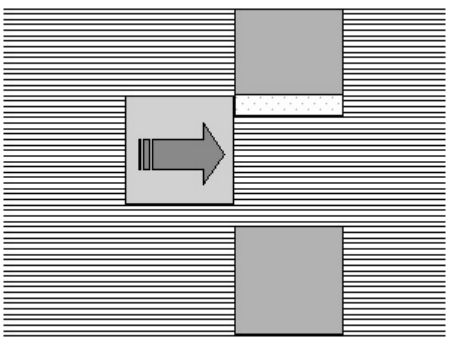
C: Wir führen jetzt für jede Zeile der Form die Unterpunkte D bis H durch. Dabei handelt es sich um eine Reihe von Prüfungen. Wenn alle Prüfungen erfolgreich absolviert sind, geht es anschließend bei I weiter. Schlägt dagegen eine der Prüfungen fehl, so erfolgt ein vorzeitiger Rücksprung.

D: Wenn die zu untersuchende Zeile sich noch unterhalb der Abdeckung befindet, finden für diese Zeile keine weiteren Prüfungen statt.

E: Jetzt werden innerhalb der betrachteten Zeile alle Spalten der Form untersucht.

F: Befindet sich in der Zeile `z` und der Spalte `s` der Form ein Segment, so finden die Prüfungen G und H statt.

G: Wenn das Offset kleiner als `16` ist, so muss die Spalte neben dem betrachteten Segment frei sein. Ansonsten erfolgt ein vorzeitiger Rücksprung. Streng genommen muss die Spalte immer frei sein. Der Verzicht auf eine Prüfung für Offsets größer als `16` schafft ein ausreichend großes Zeitfenster, um Segmente von der Seite her in Lücken einzuschieben:

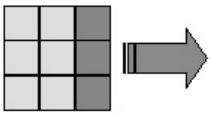


Dabei wird sozusagen der helle Bereich des rechten oberen Segments als »nicht vorhanden« betrachtet. Wichtig ist natürlich, dass der darunter liegende Bereich frei ist. Das wird aber unter H geprüft.

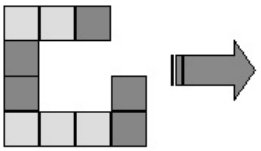
H: Ist das Offset größer als `0`, so muss auch die nächste Zeile der benachbarten Spalte frei sein. Ansonsten erfolgt ein vorzeitiger Rücksprung.

I: Sind alle Tests überstanden, kann die Bewegung ausgeführt werden. Dazu wird die `neue_spalte` als `spalte` übernommen.

Beachten Sie, dass bei der Verschiebung alle Segmente der Form betrachtet werden. Streng genommen würde es ausreichen, nur die Segmente zu betrachten, die sich bei der Verschiebung in »vorderster Front« befinden.



Segmente, die sich im Schlepptau von Segmenten der vordersten Front befinden, müssen nicht überprüft werden. Da wir hier allerdings beliebige 4x4-Formen zulassen, ist die Frontlinie nicht immer leicht auszumachen, wie das folgende Beispiel zeigt:



Die Berechnung der Frontlinie wäre letztlich so aufwändig, dass man in gleicher Zeit auch alle Segmente mit einem weitaus einfacheren Algorithmus prüfen kann.

Nach der Rechts-Links-Bewegung implementieren wir jetzt die Rechts-Links-Drehung:

```
    void ultris::drehen(int dir)
       {
       int sv, sp, zl;
       int b1, b2, h1, h2, db, dh;
       int maxb, maxh, minz, mins;
       int i, j;

A      sv = (formen[0].dv + 4 - dir)%4;
B      b1 = aktuelle_form()->b;
       b2 = ultris_form[formen[0].ix][sv]->b;
       h1 = aktuelle_form()->h;
       h2 = ultris_form[formen[0].ix][sv]->h;
```

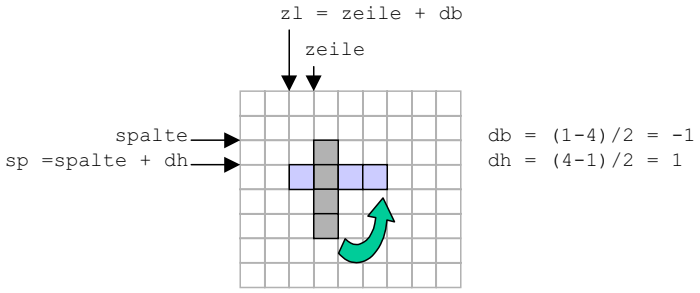
```
C     db = (b1-b2)/2;
      dh = (h1-h2)/2;
      sp = spalte+db;
      zl = zeile+dh;
D     if( sp < 0)
          sp = 0;
      if( sp+b2 >= 10)
          sp = 10-b2;
E     if( zl+h2 >= 20)
          return;
F     mins = spalte < sp ? spalte : sp;
      minz = zeile < zl ? zeile : zl;
      maxb = b1 > b2 ? b1 : b2;
      maxh = h1 > h2 ? h1 : h2;
      if( offset)
          maxh++;
G     for( i = minz; i < minz+maxh; i++)
          {
          for( j = mins; j < mins+maxb; j++)
              {
              if( (i>=0)&&(i<20)&&(j>=0)&&(j<10)
                                          &&spielfeld[i][j])
                  return;
              }
          }
H     formen[0].dv = sv;
      spalte = sp;
      zeile = zl;
      ultris_sounds.play( sound_dreh);
      }
```

Auch dazu sind sicherlich einige Erklärungen notwendig:

A: Zuerst wird der Index der in der gewünschten Drehrichtung (`dir`) folgenden Drehvariante bestimmt.

B: Dann werden die Breite und die Höhe der aktuellen Form und ihrer Drehvariante ermittelt.

C: Anhand der halben Höhen- und Breitendifferenz (`db` und `dh`) werden die Zeile (`zl`) und die Spalte (`sp`) für die gedrehte Form so berechnet, dass sich der Eindruck einer Drehung um den Mittelpunkt ergibt:

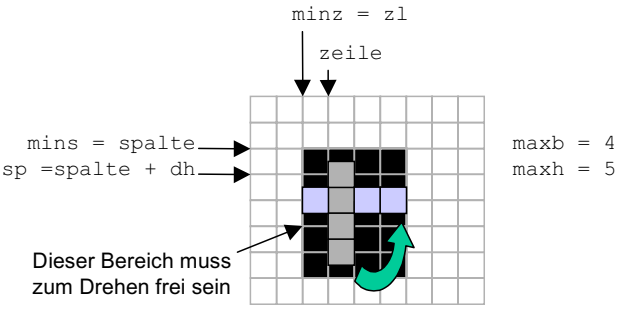


Diese Berechnung ist für Formen wichtig, die in Höhe und Breite stark voneinander abweichen.

D: Der neu berechnete Wert für die Spalte (`sp`) wird so adjustiert, dass die gedrehte Form nicht außerhalb des Spielfeldes liegt.

E: Ragt die gedrehte Figur unten aus dem Spielfeld heraus, wird nicht gedreht. Gerät die Figur beim Drehen nach oben unter die Abdeckung, so stellt das kein Problem dar.

F: Hier wird der Bereich bestimmt, der für eine Drehung der Form frei sein muss. Die Berechnung erfolgt gemäß der folgenden Skizze:



G: Hier wird überprüft, ob der zum Drehen benötigte Raum auf dem Spielfeld frei ist. Wird im Drehbereich ein Feldstein gefunden, erfolgt der vorzeitige Rücksprung.

H: Alle Prüfungen sind durchgeführt. Durch Zuweisung der Drehvariante wird die Drehung ausgeführt. Die neue Zeile und Spalte werden gesetzt, und der Drehsound wird gespielt.

Zum Abschluss binden wir die Bewegungen noch in die Benutzerschnittstelle ein. Die dazu erforderlichen Akzelleratoren haben wir ja bereits zu Beginn dieses Abschnitts eingefügt:

```
LRESULT CALLBACK ultris_windowhandler(...)
    {
    switch(msg)
        {
    case WM_COMMAND:
        switch( LOWORD( wParam))
            {
        ...
        case IDM_DREHRECHTS:
            mein_spiel.drehen(1);
            return 0;
        case IDM_DREHLINKS:
            mein_spiel.drehen(-1);
            return 0;
        case IDM_RECHTS:
            mein_spiel.bewegen(1);
            return 0;
        case IDM_LINKS:
            mein_spiel.bewegen(-1);
            return 0;
        ...
            }
        break;
        ...
        }
    ...
    }
```

Mit den Erweiterungen dieses Abschnitts können Sie bereits halbwegs Ultris spielen. Die Formen können jetzt bewegt und gedreht werden, verschwinden aber immer noch, sobald sie unten anschlagen. Wir müssen jetzt noch dafür sorgen, dass die fallenden Formen zu Feldsteinen umgewandelt werden, sobald sie blockiert sind. Danach müssen wir die durch die Umwandlung gegebenenfalls aufgefüllten Reihen abräumen.

2.4.11 V11 Auflösen der Formen und Abräumen der Reihen

In diesem Abschnitt werden wir erstmals ein vollständig funktionierendes Spiel als Ergebnis erhalten. Dazu müssen wir nur noch die Formen, sobald sie nicht mehr weiter fallen können, in Feldsteine auflösen (`aufloesen`) sowie die dadurch gegebenenfalls entstehenden vollständigen Reihen löschen und die über den gelöschten Reihen liegenden Reihen aufrücken (`aufruecken`). Zunächst erweitern wir die Klasse `ultris` um die zum Auflösen und Aufrücken erforderlichen Member-Funktionen:

```
class ultris
    {
    private:
        ...
        void aufloesen();
        void aufruecken();
    public:
        ...
    };
```

Das Auflösen von Formen ist denkbar einfach. Wir gehen durch alle Zeilen und Spalten der aktuellen Form und erzeugen überall dort einen Feldstein, wo die Form ein innerhalb des Spielfeldes liegendes Segment hat:

```
void ultris::aufloesen()
    {
    int z, s;
    int zz, ss;

    for( z = 0; z < aktuelle_form()->h; z++)
        {
        zz = zeile+z;
        if((zz >=0) && (zz<20))
            {
            for( s = 0; s < aktuelle_form()->b; s++)
                {
                ss = spalte + s;
                if( aktuelle_form()->data[z][s] && (ss>=0)
                                                && (ss<10))
                    {
                    spielfeld[zz][ss] = 1;
                    fuellstand[zz]++;
                    }
                }
            }
        }
    }
```

Wichtig ist, dass wir den Füllstandszähler der jeweiligen Reihen mitführen, damit wir erkennen können, wann eine Reihe komplett ist und abgeräumt werden muss.

Damit sind wir schon bei der Implementierung des Abräumens und Aufrückens. Beim Abräumen gehen wir von unten nach oben durch die Zeilen. Wenn wir dabei auf eine vollständig gefüllte Zeile treffen, räumen wir diese ab und lassen die darüber liegenden Zeilen aufrücken:

```
   void ultris::aufruecken()
      {
      int z, zz, s;
      int a;

A     for( z = 19, a = 0; z >= 0; )
          {
B         if( fuellstand[z] == 10)
              {
C             a++;
              punktestand += speed;
              speed += 5;
D             for( zz = z-1; zz >= 0; zz--)
                  {
                  for( s = 0; s < 10; s++)
                      spielfeld[zz+1][s] = spielfeld[zz][s];
                  fuellstand[zz+1] = fuellstand[zz];
                  }
E             for( s = 0; s < 10; s++)
                  spielfeld[0][s] = 0;
              fuellstand[0] = 0;
              }
          else
F             z--;
          }
G     if( a == 0)
          ultris_sounds.play( sound_down);
      else if( a == 1)
          ultris_sounds.play( sound_row1);
      else
          ultris_sounds.play( sound_row2);
      }
```

A: Wir gehen von unten her durch die Zeilen (`z`) des Spielfeldes. In der Variablen `a`, die hier zunächst auf `0` gesetzt wird, zählen wir dabei die abgeräumten Reihen.

B: Wir erkennen am Füllstand, dass wir eine gefüllte Zeile gefunden haben. Es geht weiter mit C, D und E.

C: Wir zählen die gefüllte Zeile (`a++`) und erhöhen den Punktestand und die Geschwindigkeit. Die beim Abräumen einer Zeile erzielte Punktzahl ist proportional zum Schwierigkeitsgrad, also proportional zur Geschwindigkeit.

D: Wir rücken alle Zeilen (`zz`) inklusive der Füllstandanzeige oberhalb der betrachteten Zeile (`z`) um eine Reihe nach unten, um die abgeräumte Reihe wieder aufzufüllen.

E: Die oberste Zeile wird freigemacht, da sie nach unten gerückt wurde. Nach der Ausführung von C bis E geht es in der gleichen Zeile (`z`) weiter, da neue Steine in diese Zeile aufgerückt sind. Der Zeilenzähler `z` wird daher in dieser Situation nicht dekrementiert.

F: Die betrachtete Zeile ist noch unvollständig (Alternative zu B) und kann nicht gelöscht werden. Also geht es in der Zeile darüber (`z--`) weiter.

G: Je nach Anzahl (`a`) der abgeräumten Reihen (`0`, `1` oder mehr als `1`) wird ein entsprechender Sound gespielt.

Die beiden neuen Funktionen müssen wir jetzt noch in der Funktion `onestep` aufrufen. Wir entfernen den dort vorläufig eingetragenen Code und programmieren den Fall der blockierenden Form wie folgt aus:

```
   int ultris::onestep()
      {
      if( offset)
          {
          ...
          }
      else
          {
          if( blockiert())
              {
A             aufloesen();
              aufruecken();
B             spiel_laeuft = (zeile >= 0);
              if( spiel_laeuft)
                  neue_form();
              else
                  {
                  zeige_dyn = 0;
                  display();
                  ultris_sounds.play( sound_ende);
                  }
              return 0;
              }
          ...
          }
      }
```

Zunächst lösen wir die blockierte Form auf und rücken gegebenenfalls Spielsteine im Spielfeld auf (A). Dann (B) stellen wir fest, ob das Spiel noch weiterlaufen soll. Das Spiel soll weiterlaufen, wenn die letzte Form noch vollständig unter der Abdeckung hervorgekommen ist (`zeile >= 0`). Wenn das Spiel weiterläuft, holen wir eine neue Form. Ansonsten schalten wir die Anzeige der dynamischen Spielelemente aus und spielen den Sound für das Spielende ab.

Jetzt können Sie erstmals mit Ihrem eigenen Spiel *Ultris* spielen. Bevor Sie jedoch endgültig der Spielsucht verfallen, wollen wir das Spiel noch mit einer Highscore-Liste versehen.

2.4.12 V12 Highscores

Das Spiel ist praktisch fertig. Zur Abrundung erstellen wir noch eine Highscore-Tabelle und den zugehörigen Dialog zur Anzeige von Highscores.

Da die Highscores in einer Datei gespeichert werden sollen, müssen wir das Format festlegen, in dem die Daten in der Datei abgelegt werden. Wir wählen ein ganz einfaches Format, bei dem die Daten für jedermann lesbar gespeichert werden:

```
567890,Phil Marlowe
456789,Sam Spade
345678,Lew Archer
234567,Inspector Columbo
123456,Ulrich Kaiser
```

Die Highscore-Tabelle wird als eigenständige Klasse modelliert. Für die Einträge erstellen wir eine Datenstruktur (`score`), die jeweils die Punktzahl und den Namen aufnehmen kann. Die Kapazität der Tabelle (`high`) wird auf fünf Einträge ausgelegt:

```
class highscore
    {
    struct score
        {
        int punkte;
        char name[40];
        };
    private:
        score high[5];
    public:
        highscore();
        ~highscore();
        int get_score( int i){ return high[i].punkte;}
        const char *get_name( int i){ return high[i].name;}
        void newscore( int pkt);
    };
```

Neben dem Konstruktor (`highscore`) und dem Destruktor (`~highscore`) gibt es Funktionen, um die Punktzahl (`get_score`) beziehungsweise einen Namen (`get_name`) aus der Tabelle zu lesen, und schließlich noch eine Funktion, um einen neuen Highscore einzutragen (`newscore`).

Im Konstruktor lesen wir die gespeicherten Highscores aus der Datei `ul_high.dat`:

```
highscore::highscore()
    {
    int i;
    FILE *pf;

    for( i = 0; i < 5; i++)
        {
        high[i].punkte = 0;
        *(high[i].name) = 0;
        }
    pf = fopen( "ul_high.dat", "r");
    if( !pf)
        return;
    for( i = 0; i < 5; i++)
        {
        fscanf( pf, "%d,", &high[i].punkte);
        fgets( high[i].name, 40, pf);
        high[i].name[ strlen(high[i].name)-1] = 0;
        }
    fclose( pf);
    }
```

Falls die Datei nicht vorhanden ist, beenden wir die Funktion vorzeitig. Das ist kein Problem, da zuvor alle Punkte und Namen mit `0` beziehungsweise als leere Strings initialisiert wurden.

Im Destruktor werden die unter Umständen geänderten Daten wieder in die Datei zurückgeschrieben:

```
highscore::~highscore()
    {
    int i;
    FILE *pf;

    pf = fopen( "ul_high.dat", "w");
    for( i = 0; i < 5; i++)
        fprintf( pf, "%d,%s\n", high[i].punkte, high[i].name);
    fclose( pf);
    }
```

Bis auf die Funktion `newscore` ist die Highscore-Klasse damit fertig. Wir legen noch ein Objekt der Klasse `highscore`

```
highscore ultris_highscores;
```

an und kümmern uns dann zunächst einmal um die zugehörige Dialog-Ressource.

Für den Dialog erstellen Sie eine Ressource mit den folgenden Identifiern:

Achten Sie dabei darauf, dass alle Felder das Attribut *Schreibgeschützt* erhalten, damit der Benutzer die Werte nicht ändern kann:

Da die Daten im Dialog nur angezeigt und nicht bearbeitet werden, ist der zum Dialog gehörende Callback-Handler ganz einfach zu programmieren. Im Rahmen der Initialisierung (`WM_INITDIALOG`) werden die Daten aus der Highscore-Tabelle (`ultris_highscores`) gelesen und in den Feldern des Dialogs angezeigt:

```
BOOL CALLBACK highscoredialog( HWND hwndDlg, UINT uMsg,
                               WPARAM wParam, LPARAM lParam)
   {
   switch (uMsg)
      {
      case WM_INITDIALOG:
         SetDlgItemInt( hwndDlg, IDC_SCORE1,
                        ultris_highscores.get_score(0), FALSE);
         SetDlgItemInt( hwndDlg, IDC_SCORE2,
                        ultris_highscores.get_score(1), FALSE);
         SetDlgItemInt( hwndDlg, IDC_SCORE3,
                        ultris_highscores.get_score(2), FALSE);
         SetDlgItemInt( hwndDlg, IDC_SCORE4,
                        ultris_highscores.get_score(3), FALSE);
         SetDlgItemInt( hwndDlg, IDC_SCORE5,
                        ultris_highscores.get_score(4), FALSE);
         SetDlgItemText( hwndDlg, IDC_NAME1,
                        ultris_highscores.get_name(0));
         SetDlgItemText( hwndDlg, IDC_NAME2,
                        ultris_highscores.get_name(1));
```

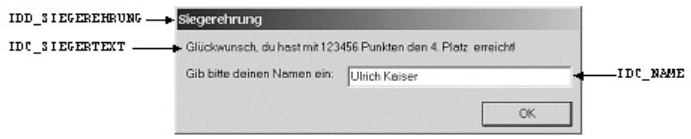
```
         SetDlgItemText( hwndDlg, IDC_NAME3,
                         ultris_highscores.get_name(2));
         SetDlgItemText( hwndDlg, IDC_NAME4,
                         ultris_highscores.get_name(3));
         SetDlgItemText( hwndDlg, IDC_NAME5,
                         ultris_highscores.get_name(4));
         return TRUE;
      case WM_COMMAND:
         if((wParam == IDOK)||(wParam == IDCANCEL))
            EndDialog(hwndDlg, wParam);
         break;
      }
   return FALSE;
   }
```

Jetzt müssen wir den Dialog nur noch mit der Funktion `DialogBox` aufrufen, wenn Menüpunkt `ID_INFO_HIGHSCORES` ausgewählt wurde:

```
LRESULT CALLBACK ultris_windowhandler(...)
   {
   switch(msg)
      {
   case WM_COMMAND:
      switch( LOWORD( wParam))
         {
      ...
      case ID_INFO_HIGHSCORES:
         DialogBox( ultris_instance, MAKEINTRESOURCE( IDD_HIGHSCORE),
                              ultris_window, highscoredialog);
         mein_spiel.reset_timer();
         return 0;
      ...
      break;
   ...
      }
   ...
   }
```

Nach der Abwicklung des Dialogs wird wie üblich der Timer zurückgesetzt, um dem Spiel vorzugaukeln, es wäre keine Zeit vergangen.

Zum Eintragen neuer Highscores benötigen wir noch einen kleinen Zwischendialog, mit dem wir den Spieler auf seinen Erfolg hinweisen und nach seinem Namen fragen. Erstellen Sie diesen Dialog mit den folgenden Identifiern:



Es ist egal, welchen Text Sie bei `IDC_SIEGERTEXT` angeben. Wir werden diesen Text bei der Siegerehrung überschreiben.

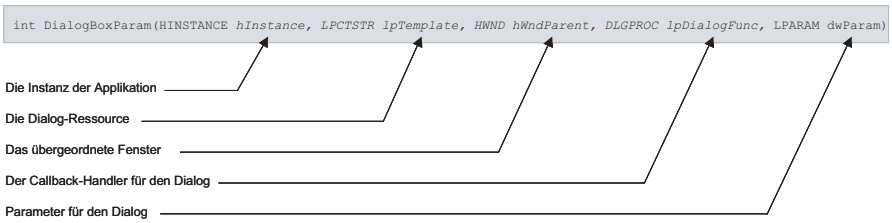
Wenn wir in der Funktion `onestep` erkennen, dass das Spiel zu Ende ist, rufen wir die Funktion `ultris_highscores.newscore` auf und übergeben dieser Funktion die erreichte Punktzahl:

```
int ultris::onestep()
    {
    if( offset)
        {
        ...
        }
    else
        {
        if( blockiert())
            {
            ...
            if( spiel_laeuft)
                ...;
            else
                {
                ...
                ultris_highscores.newscore( punktestand);
                }
            ...
            }
        ...
        }
    }
```

In der Funktion `newscore` wird zunächst geprüft, ob die Punktzahl für einen Platz in der Highscore-Tabelle qualifiziert (A). Wenn die Punktzahl für einen der fünf ersten Plätze (nummeriert von `0` bis `4`) reicht (`pos < 5`), fallen alle Einträge in der Highscore-Tabelle mit einer geringeren Punktzahl um einen Platz zurück (B). Auf dem erreichten Platz wird die Punktzahl eingetragen, das Feld für den Namen wird vorläufig noch leer gelassen (C). Anschließend erklingt die Siegesfanfare (D):

```
    void highscore::newscore( int pkt)
      {
      int pos, i;
      char buf[256];

A     for( pos = 5; pos && (high[pos-1].punkte < pkt); pos--)
        ;
      if( pos < 5)
        {
B       for( i = 4; i > pos; i--)
          high[i] = high[i-1];
C       high[pos].punkte = pkt;
        *(high[i].name) = 0;
D       ultris_sounds.play( sound_win);
E       sprintf( buf,
        "Glückwunsch, du hast mit %d Punkten den %d. Platz erreicht!",
                                                pkt, pos+1);
F       DialogBoxParam( ultris_instance,
                  MAKEINTRESOURCE( IDD_SIEGEREHRUNG),
                       ultris_window, siegerehrung, (LPARAM)buf);
G       buf[38] = 0;
        strcpy( high[pos].name, buf);
H       PostMessage( ultris_window, WM_COMMAND,
                                              ID_INFO_HIGHSCORES, 0);
        }
      }
```

Im Zeichenpuffer `buf` wird dann der Ausgabetext für die Siegerehrung bereitgestellt (E). Dieser Puffer wird beim Starten des Dialogs zur Siegerehrung durch die Funktion `DialogBoxParam` als letzter Parameter übergeben (F). Die Funktion `DialogBoxParam` ist weitgehend identisch mit der bereits des Öfteren verwendeten Funktion `DialogBox`:



Der einzige Unterschied besteht in dem letzten, zusätzlichen Parameter. Dieser wird zur Initialisierung (`WM_INITDIALOG`) an den Callback-Handler des Dialogs (`siegerehrung`) weitergereicht. Wir werden diesen Parameter im Callback-Handler des Dialogs entgegennehmen und über diesen Parameter auch den im Dialog eingegebenen Spielernamen zurückmelden. Nach dem Aufruf der Funk-

tion DialogBoxParam, das heißt nach der Abwicklung des Dialogs, steht der Spielername im Zeichenpuffer buf.[27] Wir schneiden den Namen gegebenenfalls ab, damit er in die Highscore-Tabelle passt, und kopieren ihn anschließend (G) in den Array zur Aufnahme der Highscore-Daten (high) an die vom Spieler erreichte Position (pos). Abschließend schicken wir uns selbst das Windows-Kommando ID_INFO_HIGHSCORES, was dazu führt, dass der Highscore-Dialog angezeigt wird.

Jetzt müssen wir nur noch den Callback-Handler des Siegerehrung-Dialogs programmieren:

```
   BOOL CALLBACK siegerehrung( HWND hwndDlg, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam)
     {
     static char *parameter;

     switch (uMsg)
        {
        case WM_INITDIALOG:
A          parameter = (char *)lParam;
           SetDlgItemText( hwndDlg, IDC_SIEGERTEXT, parameter);
           return TRUE;
        case WM_COMMAND:
           if(wParam == IDOK)
              {
B             GetDlgItemText( hwndDlg, IDC_NAME, parameter, 256);
              EndDialog(hwndDlg, wParam);
              }
           break;
        }
     return FALSE;
     }
```

Im Rahmen der Initialisierung speichern wir den Zeiger auf den von der Funktion newscore übergebenen Puffer in einer statischen Variablen. Den im Puffer übergebenen Text zeigen wir im Textfeld IDC_SIEGERTEXT des Dialogs an (A). Beendet der Benutzer den Dialog mit *OK* (anders geht es nicht), so wird der eingegebene Name in den Zeichenpuffer geschrieben und damit an die aufrufende Funktion zurückgemeldet (B).

Geschafft! Jetzt sollten Sie erst einmal spielen, um die Früchte Ihrer Arbeit zu genießen.

27 Das müssen wir im Dialoghandler siegerehrung natürlich noch implementieren (siehe weiter unten).

Wenn Sie zur Genüge gespielt haben, sollten Sie nicht sofort mit der 3D-Spieleprogrammierung anfangen, sondern zunächst einmal ein eigenes 2D-Spiel realisieren. Anregungen dafür gibt es genug. Es kann etwas Einfaches wie Pacman oder etwas Anspruchsvolles wie ein vollwertiger Flipper sein. Geeignet ist im Prinzip jedes Spiel, dass Sie vom Gameboy oder PocketPC her kennen. Sie können natürlich auch eine eigene Spielidee entwickeln.

Wenn Sie Ihr eigenes Spieleprojekt fertig gestellt haben, treffen wir uns im nächsten Abschnitt wieder zur 3D-Spieleprogrammierung.