MATURAARBEIT:
WIE PROGRAMMIERE ICH EINE RTS-ENGINE?

von

Stephan Vock und Raphael Ochsenbein

Klasse Ic

Andreas Locher

Gymnasium Oberaargau

2005

„Strategie ist wenn man keine Munition mehr hat, aber trotzdem weiterschiesst, damit der Feind davon nichts merkt.“ - Unbekannt

Inhaltsverzeichnis

1. Einleitung

1.1 Vorwort

Seit dem Beginn der Menschheit führt sie untereinander Kriege. Im Laufe der Zeit wurde mehr Geld für die Entwicklung und Produktion von Waffen als für die Entwicklung von Medizin und Nahrung ausgegeben. Wenn man sich das überlegt, ist dies eigentlich sehr traurig. Nicht nur, dass der Krieg sehr selten von einer Mehrheit befürwortet wird. Nein, auch dass meistens diejenigen die Folgen eines Krieges tragen müssen, die eigentlich gar nichts damit zu tun haben wollen.
Durch die grossen Investitionen hat sich die Technologie immer verbessert, und es ergab sich sogar, dass Projekte, die nur für das Militär gedacht waren, auch für einen breiten Teil der Bevölkerung verwendet werden konnten. Das beste Beispiel dafür ist das Internet.
Alle diese Verbesserungen der kriegstechnischen Möglichkeiten führten dazu, dass die Schlachten nicht mehr nur auf dem Feld geschlagen wurden, sondern schon vorher überlegen sich unzählige Genies, wie man den Krieg am schnellsten und möglichst ohne grosse Verluste gewinnen kann. Schon zu Zeiten Napoleons wurden Kriege zuerst auf dem Papier geführt. Heute würde man wohl eher sagen, am Computer. Seitdem haben sich die Möglichkeiten einer Simulation verbessert. Die Menschen wurden grösstenteils durch Computer ersetzt. Durch diesen Gewinn an Rechengeschwindigkeit kamen die Berechnungen der Realität einen grossen Schritt näher. Ist es da nicht nahe liegend, dass die Unterhaltungsindustrie auch diesen Lebensteil umsetzt? Denn wer wollte nicht auch in seiner Kindheit einmal General spielen? Wollte nicht jeder König einen Krieg ohne Verluste gewinnen?
Wir haben diesen Zweig der Unterhaltungsbranche für uns entdeckt. Nicht weil wir Kriege befürworten würden. Im Gegenteil: Wir lehnen Kriege strickte ab. Doch steckt in solch einem Spiel neben Unterhaltung, nicht auch Bildung? Lernt man in einem solchen Spiel nicht ungeheuer viel? Man lernt die Namen von Kaisern, Generälen und

anderen Helden kennen. Man schult sein logisches Denken und seine Fähigkeit, Probleme und Konflikte zu lösen.
Wir sind fasziniert davon, wie man aus Einsen und Nullen eine atemberaubende Grafik und eine überaus realistische Handlung eines Krieges simulieren kann. Und vielleicht werden sich all die Regierungshäupter und Präsidenten in ihren so genannten Demokratien ein Beispiel nehmen und ihre Kriege nicht mehr durch einfache Mensche führen lassen, sondern sich selber hinter einen Computer setzen und so ihre Freizeit verbringen (natürlich nur während der Zeit, in der sie sonst Krieg führen würden, sie müssen das eigene Land ja auch noch regieren). Vielleicht geht es ja noch weiter, und sie können ihren Hass gegen andere Menschen so ausleben und dafür in der Realität voller Liebe und Hilfsbereitschaft dem Wohle der Menschheit dienen.

1.2 Zielsetzung

Wir beide wussten eigentlich von Anfang an, dass wir unsere Maturaarbeit im Bereich Informatik vollbringen werden, dachten aber eigentlich nie an eine gemeinsame Arbeit. Als Raphael dann eines Tages die Idee hatte, ein eigenes Strategiespiel zu programmieren, sahen wird dieser Herausforderung mit einer gewissen Skepsis, wie natürlich auch mit einer grossen Freude entgegen.
Für uns war jedoch schon nach kurzer Recherche klar, dass ein ganzes Spiel den Rahmen einer einfachen Arbeit bei weitem sprengen würde. So entschieden wir uns, nicht ein ganzes RTS zu programmieren, sondern nur Teile einer Game-Engine in Angriff zu nehmen. Unser Ziel war es, bis zum 1. November zwei unabhängig voneinander laufende Programme zu schreiben und als Sahnehäubchen dazu noch einen schriftlichen Teil, der unsere Überlegungen für den Leser doch möglichst logisch erklären sollte.

1.3 Umsetzung in Visual C++ 6.0

Wir haben unsere zwei Teile einer RTS-Einige in C++ geschrieben. Dies aus dem einfachen Grund, weil C++, die wohl am häufigsten benutzte Sprache für Computerspiele ist. Im Gegensatz zu Visual Basic muss man bei C++ alles selber programmieren. Das heisst, das Fenster, in dem das Programm läuft, jedes Textfeld und jeder Button muss selber erstellt werden. Dies hat natürlich seine Vorteile: Man kann den einzelnen Objekten verschiedene Parameter zuweisen und sich so das Gewünschte nicht einfacher, aber doch effizienter gestalten.
Da wir uns noch nie an ein grösseres Projekt in C++ gewagt hatten, mussten wir zuerst einige Grundlagen des Programmierens in C++ erlernen. Wie stelle ich ein leeres Fenster dar? Wie einen Punkt in diesem Fenster? Wie stelle ich Grafiken dar? - Dies waren, die Fragen, um welche sich unsere Arbeit zu Beginn drehte.
Dies kam uns am Anfang etwas deprimierend vor, da wir nach einigen Monaten immer noch keine Erfolge für unsere Arbeit sehen konnten. Doch die ganze Geduld hat sich gelohnt. C++ ist im Gegensatz zu Visual Basic um einiges schneller. Und ohne dieses Geschwindigkeit wären unsere Programme sicher untauglich gewesen, weil sie einfach zu langsam wären.
So wird in unserer Arbeit auch nirgends der vollständige Code erscheinen, sondern nur Codeausschnitte, damit wir mit Hilfe von diesen, darstellen können, wie man eine Pfadfindung und ein Kampfscript programmieren könnte. Wie gesagt könnte, denn es gibt Dutzende anderer und sicher auch besserer Möglichkeiten, ein Projekt anzugehen und umzusetzen. Aber diese würden den Rahmen unserer Arbeit bei weitem sprengen.

2. Begriffe

2.1.1RTS

Hinter der für viele Leute unbekannten Abkürzung verbirgt sich der Begriff „Real Time Strategy“, zu Deutsch „Echtzeitstrategie“. Hierbei ist das Wort Echtzeit möglicherweise ein wenig irreführend. Es bedeutet hier nicht, dass eine Aktion im Spiel gleich lange benötigt, wie wenn sie in der realen Welt ausgeführt würde. Es bezieht sich in diesem Sinne auf die Gleichzeitigkeit, dass mehrere Spieler mehrere Aktionen zur selben Zeit ausführen können und diese vom Computer auch gleich behandelt werden. Das Spiel läuft demnach einfach um einen Faktor x gegenüber der Erdzeit beschleunigt. Diese Beschleunigung führt dazu, dass der Anfang beinahe jedes Strategiespieles, (Aufbau einer Basis und Suchen nach Rohstoffen) doch auf eine angebrachte Länge gekürzt werden kann. Der Spieler kann sich dann beinahe ausschliesslich mit dem Ausbauen seiner Basis und dem Herstellen von Einheiten beschäftigen. Um das Ende des Spieles zu erreichen, muss der Spieler in die Schlacht ziehen. Durch seine taktische oder zahlenmässige Überlegenheit kann er seinen virtuellen Gegner vernichtend besiegen.

2.1.2 Die Geschichte der Echtzeitstrategiespiele

Die Geschichte der Echtzeitstrategiespiele begann mit dem Spiel „Herzog Zwei“, welches im Jahre 1989 auf der Konsole „Sega Mega Drive“ erschien. Als einer der Pioniere gilt aber „Dune II“, erschienen 1992. Es verfügte als erstes Spiel über die typischen Merkmale eines RTS(-spiels); gerade diese bescherte den Entwicklern eine immer grösser werdende Fangemeinde. Richtig bekannt wurden die RTS jedoch erst in den Jahren 1994 und 95, als der 1. Teil der „Warcraft“- Serie erschien und eine zweite sehr bekannte Reihe ihren Anfang feierte - die „Command & Conquer“- Serie war

geboren. Seit diesem Zeitpunkt steigt der Bekanntheitsgrad dieser Spiele unaufhaltsam, und vielleicht werden sie schon in naher Zukunft einen ernst zu nehmenden Konkurrenten für die „Ego-Shooter“. Der stetig wachsende Bekanntheitsgrad hat sicher damit zu tun, dass in Strategiespielen ein unglaubliches Potenzial steckt. Um nur ein Beispiel zu nennen: Bis jetzt hatte jedes Objekt nur eine Animation, wie es aus dem Spielgeschehen ausschied. Dies wurde jetzt geändert. Am 4. November erscheint im deutschsprachigen Raum ein Spiel, das einen neuen Meilenstein legen wird. Das erste Mal wird, die vor allem aus „Ego-Shootern“ bekannte „Havok-Physik-Engine“, in einem Strategiespiel eingebaut sein. Der dritte Teil der „Age of Empires“-Reihe wird das bis jetzt beste Spiel „Command & Conquer: Generals“ höchstwahrscheinlich ablösen. Nicht nur die Physik-Engine wird den Spielspass erheblich verbessern, auch Grafik, Sound und alle anderen typischen Merkmale werden sicher nicht zu kurz kommen. Nicht umsonst sind seit dem 2. Teil mehr als fünf Jahre vergangen. Der Spieler darf sich auf jeden Fall schon jetzt freuen.

2.2.2 Game- Engine

Eine Game-Engine bildet das Grundgerüst eines jeden Computerspieles. Sie besteht aus mehreren Teilen. Darunter Bibliotheken für das Soundsystem, die Garfik-Engine, die künstliche Intelligenz, computergesteuerter Objekte, die Steuerung, und - wenn das Spiel über einen Mehrspielermodus verfügt, auch einen Netzwerkcode. Mit der Game-Engine alleine hat man vom Aussehen her noch kein Spiel. Sie ist eine Ansammlung von verschiedenen Codeteilen und wird vom Programmierer erstellt. Jedoch ist dieser Teil, der Wichtigste; nicht umsonst wird er auch als „Core“ (Kern) bezeichnet.
Eine Engine ist für mehrere Spiele, wo möglich auch für verschiedene Genres, nutzbar. Deshalb kann eine Firma, die eine wirklich gute Engine geschrieben hat, diese dann auch teuer verkaufen. Hat man dazu noch ein bekanntes Spiel, in welchem die Engine eingebaut ist, so kann dies durchaus noch ein positiver Vermarktungseffekt sein.

„Der Weg ist das Ziel“ - Konfuzius

3. Pfadfindung

3.1 Was ist eine Pfadfindung?

Haben Sie sich schon einmal überlegt, was es alles braucht, um von einem Punkt A zu einem Punkt B zu kommen? Haben Sie sich schon einmal überlegt, wie viel es braucht, um am Morgen aufzustehen und nur gerade ins Badezimmer zu gehen? Eigentlich nur Überwindung, denkt man sich. So meinen auch viele Benutzer von Computerspielen, dass die Figuren einfach so in ihrer Umgebung umherlaufen. Wieso auch nicht? Sie sind dort ja zu Hause, kennen jeden Winkel ihrer Umgebung, den ihnen die

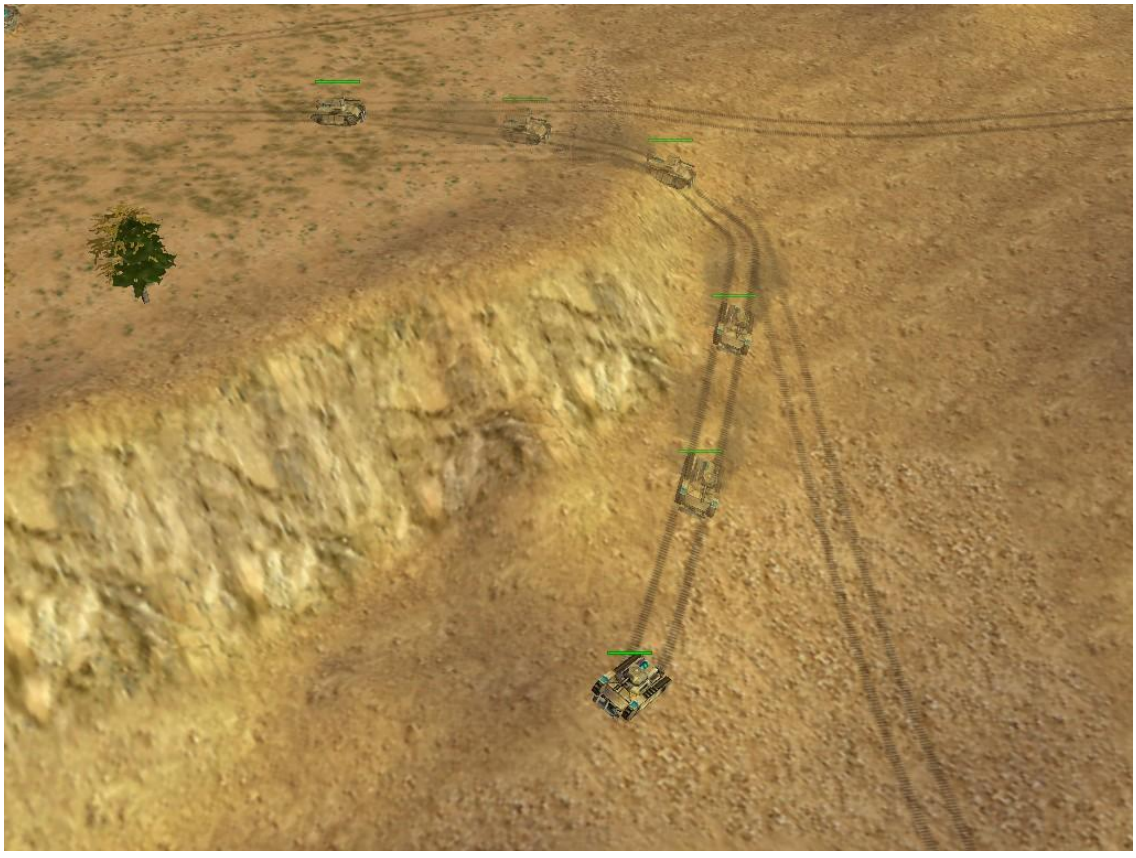


Abb. 3.1.1: Man sieht wie ein Panzer einen Weg abfährt. Obwohl dieser Weg nicht der Kürzeste ist, so ist er doch der Schnellste; Screenshot aus C&C Generals

Programmierer vorher mühsam eingetrichtert haben. Doch, ist das wirklich realistisch?

Für einige Leute ist es vielleicht unvorstellbar, aber die Figuren haben keine Ahnung, wie ihre Umgebung aussieht, sie wissen nichts davon. Man kann es vielleicht mit uns vergleichen, wenn wir am Morgen früh aufstehen müssen. Wir sehen unseren Weg noch nicht, aber trotzdem finden wir ihn. Dies, weil das Gehirn den Weg weiss und uns zum gewünschten Ziel führt. So ist es auch bei den Figuren im Spiel. Sie selbst wissen den Weg nicht, aber der Computer berechnet für sie einen Weg, welcher sie zum Ziel führen sollte.

Wenn man sich jetzt an die alten Strategiespiele erinnert, dann sieht man eine stark vereinfachte Landschaft von oben. Nehmen wir an, wir haben ein Objekt, welches von einem Anfangs- zu einem Endpunkt möchte.

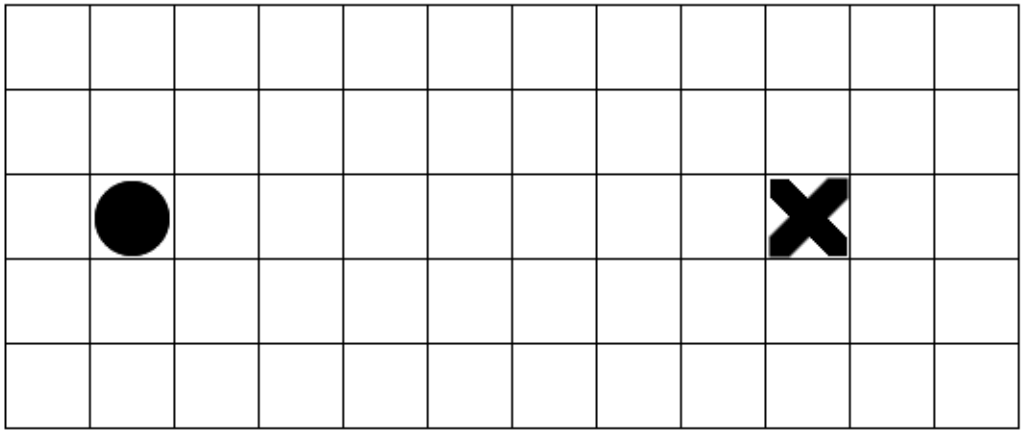


Abb. 3.1.2: Ein zweidimensionales Koordinatensystem, in welchem ein Start- (Kreis) und ein Zielfeld (Kreuz) eingesetzt sind; Grafik mit Fireworks 8 erstellt

Der einfachste Code könnte demnach so aussehen:

```
while (_x.objekt!=_x.end && _y.objekt!=_y.end)
{
        if (_x.objekt>_x.end) _x.objekt--;
        else if (_x.objekt<_x.end) _x.objekt++;
        else if (_y.objekt>_y.end) _y.objekt--;
        else if (_y.objekt<_y.end) _y.objekt++;
```

```
}
```

Im oben stehenden Code wird das Objekt so lange in Richtung des Endpunktes bewegt, bis die x- und die y-Koordinaten des Objektes mit dem Endpunkt übereinstimmen.

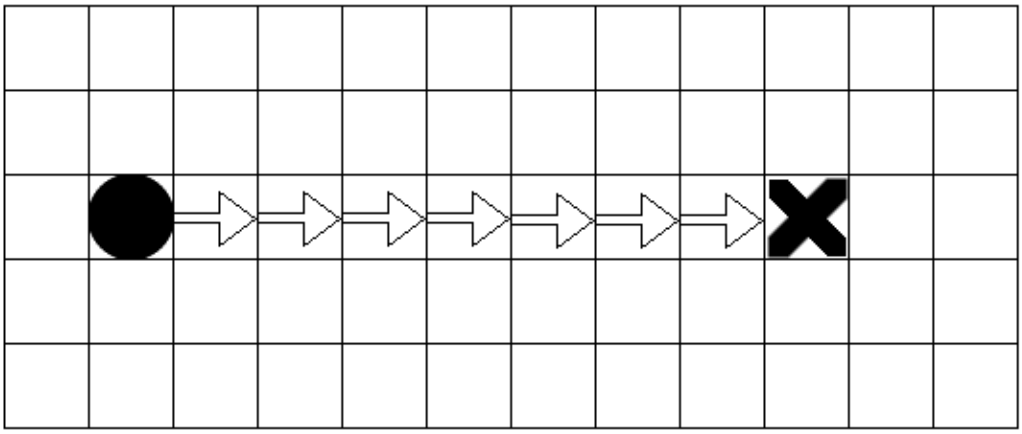


Abb. 3.1.3: Das Objekt im zweidimensionalen Koordinatesystem bewegt sich solange in Richtung, des Zielfeldes, bis es dieses erreicht hat; Grafik erstellt mit Fireworks 8

Dies funktioniert jedoch nur auf einfachster Basis, zum Beispiel auf einem Fussballfeld, begrenzt durch die Aussenlinien. Gehen wir jedoch eine Ebene höher wie in Abbildung 3.1.4, so stösst unser Code schon relativ schnell an seine Grenzen. Er würde einfach

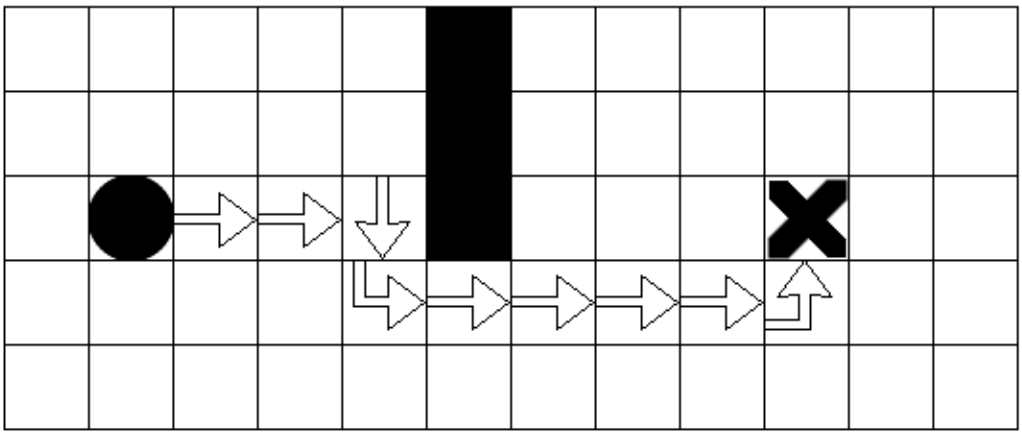


Abb. 3.1.4: Das Hindernis wird erkannt und umgangen; Grafik erstellt mit Fireworks 8

durch das Hindernis hindurch gehen. Was ja nicht die Aufgabe unseres Beispielprogrammes ist. Wir müssen den Code um eine Anweisung erweitern, damit er ein Hindernis erkennt und umgeht.

```
while (_x.objekt!=_x.end && _y.objekt!=_y.end)
{
                                                    if (_x.objekt>_x.end) _x.objekt--;
        else if (_x.objekt<_x.end) _x.objekt++;
        else if (_y.objekt>_y.end) _y.objekt--;
        else if (_y.objekt<_y.end) _y.objekt++;
        if (blockiert) zufall();
}
```

Wir haben jetzt dem ursprünglichen Code eine weitere Bedingung hinzugefügt, die überprüft, ob der Weg blockiert ist. Wenn dies zutrifft, führt es eine Funktion mit dem Namen zufall() aus, welche dann einen zufälligen möglichen Weg geht.

3.2 Die A-Star Pfadlösung

Eine der wichtigsten Eigenschaften einer Pfadfindung ist, nach dem Suchen und Finden des Pfades, auch ihre Ausgewogenheit zwischen Geschwindigkeit und Ressourecenauslastung. Es hat keinen Sinn, wenn eine Pfadfindung unglaublich schnell ist und im Gegenzug während der Suche das ganze Spiel lahm legt. Viele Entwickler haben sich schon mit der Entwicklung und Programmierung einer Pfadfindung auseinander gesetzt. Dabei entstanden verschiedenste Lösungsansätze. Einer dieser Ansätze wurde weiterverfolgt, und aus ihm folgte die so genannte A-Star oder Kurz A* Pfadlösung.
Es ist ganz normal, dass Ideen und Projekte immer weiterverfolgt und ausgearbeitet werden. So hat es die A-Star Pfadlösung von einer zweidimensionalen Umgebung, mit welcher ich gearbeitet habe, auch in eine dreidimensionale Welt geschafft.

3.2.1 Aufbau einer Welt

Damit man versteht, wie diese Pfadlösung arbeitet, ist es wichtig, sich zuerst mit dem Aufbau, der Spielwelt, der so genannten „Map“, auseinander zu setzten.
Jede Map ist aus verschiedenen kleineren Vierecken, genannt „Nodes“ oder auch „Tiles“, aufgebaut. Jedes hat dabei die gleiche Grösse, bei mir 32x32 Pixel, dies kann je nach Programm variieren. Je kleiner die Felder, desto genauer die Pfadfindung, aber auch desto langsamer das Programm.
Die Karte hat dadurch schon eine fest vorgegebene Grösse. Je nach Anzahl Nodes zum Beispiel 640x480 Pixel. Diese Grenzen kann die Figur nicht verlassen, da sich ausserhalb keine Felder befinden, auf welchen sie sich bewegen könnte.
Da wir jetzt wissen wie die Welt in etwa aussieht, fragt sich sicher der eine oder andere: Warum in aller Welt muss man die ganze Karte in kleinere Felder aufteilen? Man könnte ja auch ein grosses Bild als Hintergrund hinlegen. Das wäre schon richtig, aber diese Felder haben eine bestimmte Aufgabe. Denn jedes erhält einen gewissen Wert, dieser kann überall gleich sein oder überall verschieden, das ist eigentlich nicht wichtig. Er sollte nur nicht 0 sein, da wenn kein Feld vorhanden ist, dieser Wert zurückgegeben wird. Man gibt beispielsweise jedem grünen Feld den Wert 2 und jedem blauen den Wert 3, Felder die blockiert sind erhalten -1. Blockierte haben eigentlich nur einen noch nicht verwendeten Wert, damit man sie von den anderen Feldern unterscheiden kann; dies ist sehr praktisch, am einfachsten ist es, man gibt allen begehbaren Feldern einen Wert grösser Null und den Blockierten einen Negativen.

3.2.2 Der Pfad

Da die Welt jetzt steht, kann man sich hinter den Pfad machen. Dabei ist es wichtig, dass man sich zuerst einen Start- und einen Zielpunkt setzt, denn ohne Start und Ziel gibt es auch keinen Weg.
Wenn wir das haben, werden zwei Listen eingeführt, je eine Open und eine Closed List. (Ich hab den Namen in Englisch gelassen, weil mir wirklich kein passender Name

auf Deutsch in den Sinn kam.) Open bedeutet offen, hier im Programm stellt es einen möglichen nächsten Punkt dar. Closed für geschlossen bedeutet: Da waren wir schon einmal und wir können nicht mehr durch. Das Erste, das getan wird, ist den Startpunkt in die Closed List zu setzen, und alle angrenzenden Felder werden in die Open List gesetzt. Doch wie wird jetzt errechnet, über welches Feld die Figur als Nächstes geht?

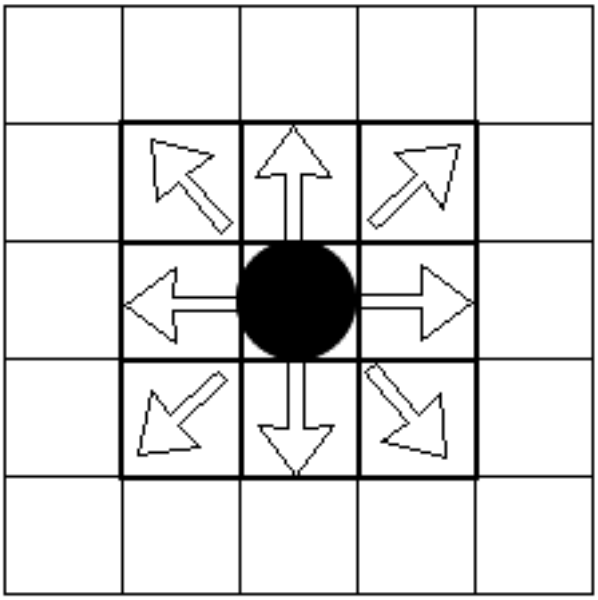


Abb. 3.2.1: Alle angrenzenden Felder, dick umrahmt, werden in die Open List gesetzt; Grafik erstellt mit Fireworks 8

Das ist eigentlich ganz simpel: Zuerst wird überprüft, ob das Feld blockiert ist. Wenn nicht, werden 3 Werte berechnet: die bisherigen Kosten, die Kosten des Feldes und die Kosten bis zum Endpunkt. Die bisherigen Kosten sind in unserem Fall die Kosten des Startpunktes. Die Kosten des zu errechnenden Feldes sind diejenigen Kosten, welche bei der Gestaltung der Map dem Viereck zugewiesen wurden. Bis

jetzt ist eigentlich alles klar. Doch wie berechnet man die Kosten bis zum Endpunkt, wenn man den Pfad noch gar nicht hat? Auch dies wird wieder auf eine andere Art gemacht als man zuerst denken würde: Und zwar zählt man einfach die Anzahl Felder in Richtung der x-Koordinate, bis man auf der Höhe des Endpunktes ist; und dasselbe macht man auch mit der y-Koordinate, zählt diese zusammen - und schon hat man die Kosten bis zum Endpunkt. Am Schluss zählt man alle diese drei Kosten zusammen und erhält die Gesamtkosten für das gewünschte Feld. Man berechnet diesen Faktor für jedes Feld auf der Open List, wählt dasjenige mit den kleinsten Kosten aus und setzt es in die Closed List. Dies geht so weiter, bis man den Endpunkt erreicht hat.

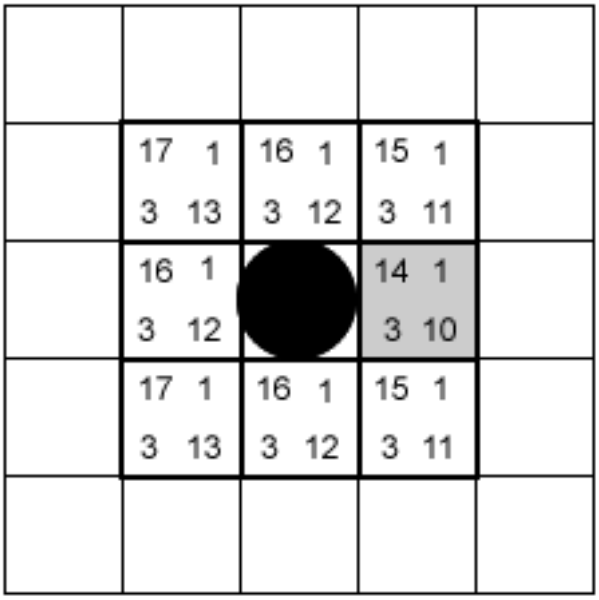


Abb. 3.2.2; Hier sieht man ein Beispiel für die Kosten der Angrenzende Felder. Rechts oben, die Kosten des Feldes, rechts unten die Zielkosten, links unten die Startkosten und links oben, die Gesamtkosten; Grafik erstellt mit Fireworks 8

Jetzt könnte es ja sein, dass am Anfang mehrere Felder den gleichen tiefsten Wert haben, und so könnte es doch zu Fehlern kommen. Das ist durchaus möglich, doch A* korrigiert sich selber, der gefundene Pfad wird mit Hilfe der geschlossenen Liste noch einmal zurückberechnet, und somit können Fehler - ganz besonders bei den ersten paar Feldern - auf einfache Weise korrigiert werden.
Hat jetzt eine Karte gar keine Lösung, das heisst, dass der Weg wird durch Hindernisse versperrt und zwar so fest, dass keine einzige Lücke da ist, endet das Programm in einer Endlosschleife. Doch um dieses Detail zu beheben, ist kein grosser Kunstgriff mehr nötig. Man setzt ein Maximum an begehbaren Feldern. Weist die geschlossene Liste plötzlich mehr Felder auf als das Maximum zulässt, so wird der Vorgang abgebrochen, weil kein Pfad gefunden wurde.

3.3 Die Programmierung

3.3.1 Map laden und Voreinstellungen

Damit die Pfadfindung später auch funktioniert, müssen zuerst einige Vorkehrungen getroffen werden. Ausser dem Zeichnen des Fensters, der Definition verschiedenster Variabeln, muss auch eine Defaultmap geladen werden. Dies geschieht mit der Funktion vLoadMap(). Mit dem Dateinamen als Argument. Diese Funktion öffnet aus einer binären Textdatei, eine Datei nur aus Nullen und Einsen, und liest danach die Werte aus. Die ausgelesenen Daten sind eigentlich schon die Map. Nur sie ergeben alleine noch kein Bild. Die entsprechenden Tiles müssen geladen werden und die Karte kann gezeichnet werden. Dies geschieht auch mit dem Start- und dem Zielfeld. Sie sehen logischerweise anders aus als der Rest der Tiles, damit wir sie unterscheiden können. Die Koordinaten der beiden Felder werden noch speziellen Variablen zugewiesen, damit wir sie später schneller finden.
Damit man überhaupt irgendwelche Befehle geben kann wird eine Command Bar erstellt. Zwei Buttons reichen dabei völlig aus, der eine um eine andere Map zu laden,

und der andere um die Pfadfindung auszulösen. Nach dem diese programmiert wurden, sollte das Ganze in etwa so aussehen, wie in Abbildung 3.3.1.

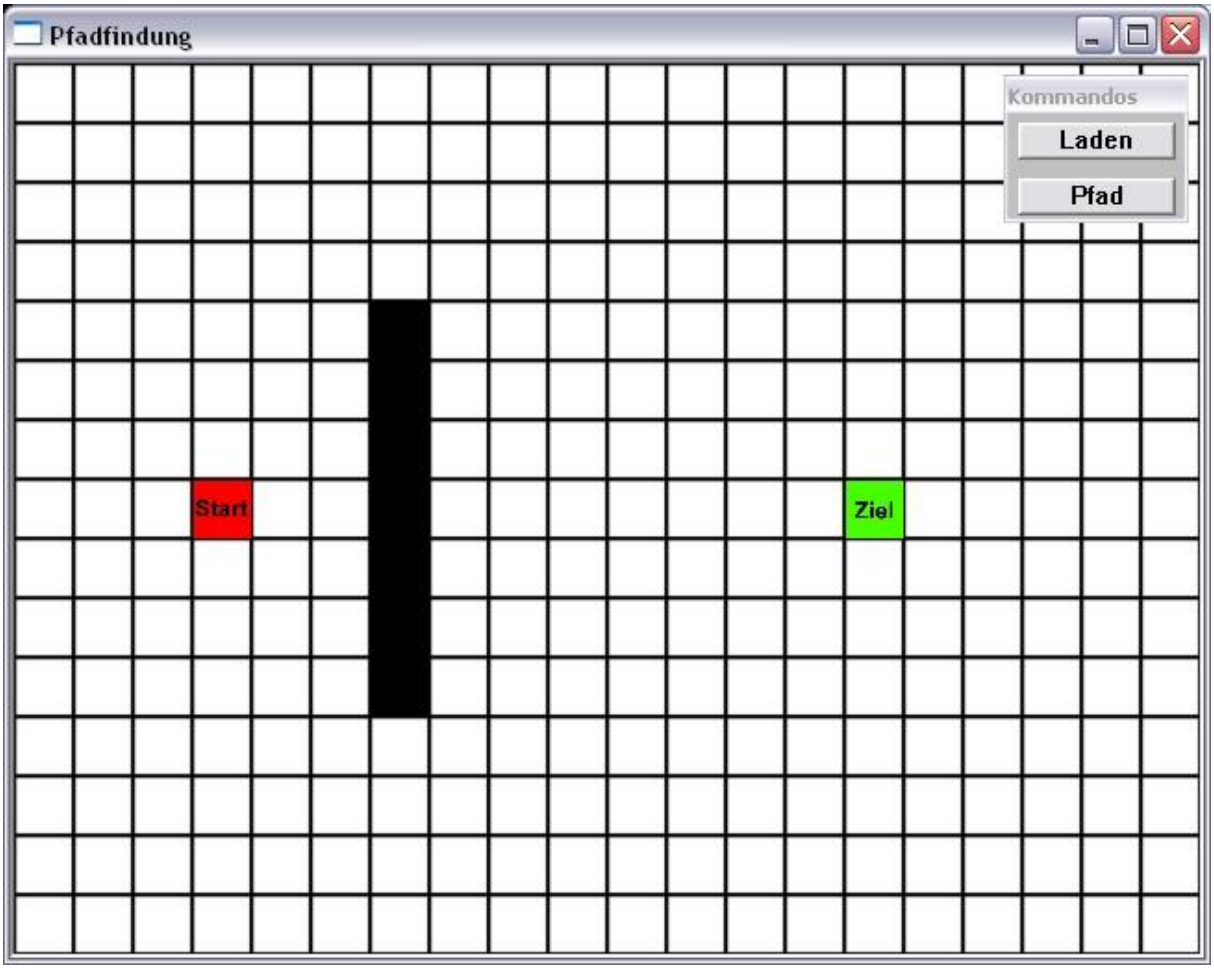


Abb. 3.3.1: Bild beim Starten des Programms; Screenshot von Stephan Vock

3.3.2 Die Pfadfindung

Kommen wir jetzt zum interessanten Teil des Programms, dem Suchen und Finden eines Pfades, natürlich nicht irgendeinem, sondern dem Schnellsten. Von den vier Dateien, main.h, main.ccp, pfadfindung.h und pfadfindung.ccp sind für das berechnen des Pfades, die zwei zuletzt genannten wichtig. Sie beinhalten die CPathFinder Klasse,

welche den Pfad im Ende für uns berechnen wird. Doch schauen wir uns zuerst die Funktion vInitPathing() in der Datei main.ccp an.

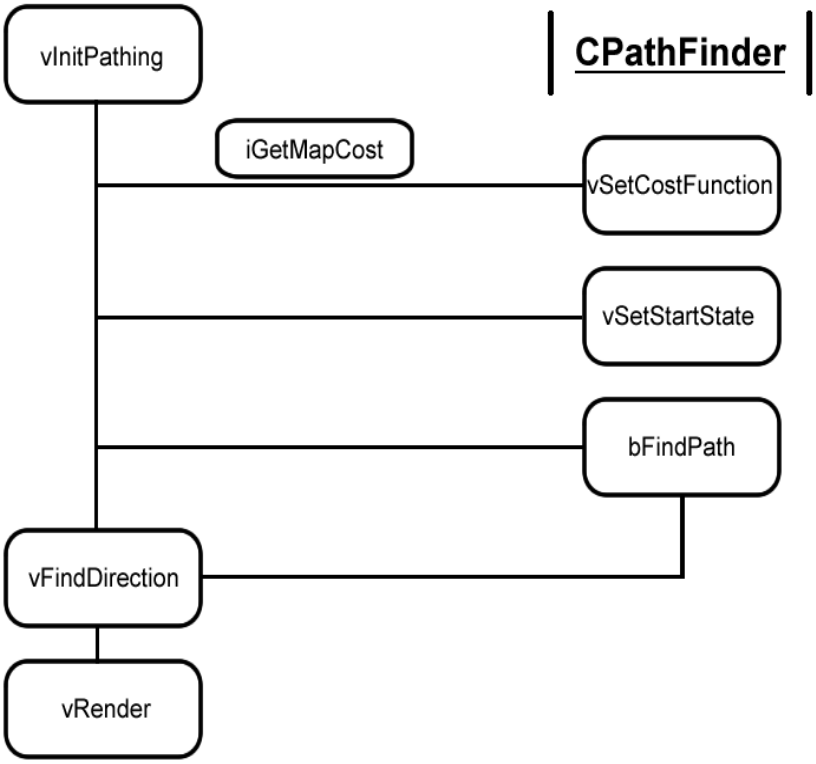


Abb. 3.3.2: Ein Schema für die Funktion vInitPathing(); Grafik erstellt mit Fireworks 8

```
//
// Pfad und folgen
//
void vInitPathing( void )
{
 bool    bRet;
 int     iTempX;
 int     iTempY;
 int     iDir;
 // Start und Zielposition
 int     iNodeStartX;
 int     iNodeStartY;
 int     iNodeEndX;
 int     iNodeEndY;
 // PathFinder Klasse Objekt
 CPathFinder  pathMyPath;

 // Pfeil löschen
 // Um danach den Pfad damit anzuzeigen
 for( int i = 0; i < g_iMapWidth * g_iMapHeight; i++ ) {
  g_iArrowMap[ i ] = -1;
 }
```

```
//
// Startposition
//

// Map durchsuchen, für Startfeld
for( int y = 0; y < g_iMapHeight; y++ ) {
 for( int x = 0; x < g_iMapWidth; x++ ) {
  if( g_iTileMap[ x+(y*g_iMapWidth) ][ 0 ] == 19 ) {
   iNodeStartX = x;
   iNodeStartY = y;
   break;
  }
 }
}

//
// Endposition
//

// Map durchsuchen, für Endfeld
for( y = 0; y < g_iMapHeight; y++ ) {
 for( int x = 0; x < g_iMapWidth; x++ ) {
  if( g_iTileMap[ x+(y*g_iMapWidth) ][ 0 ] == 20 ) {
   iNodeEndX = x;
   iNodeEndY = y;
   break;
  }
 }
}

vRender();

// Kostenfunktion
pathMyPath.vSetCostFunction( iGetMapCost );
// Start und Ziel
pathMyPath.vSetStartState( iNodeStartX, iNodeStartY, iNodeEndX, iNodeEndY );
// Pfad mit einem Maximum von 300 Feldern suchen
bRet = pathMyPath.bFindPath( 300 );

// Beenden bei Fehler
if( !bRet ) {
 return;
}
else {
 //Könnte man irgend etwas machen, z.B. angeben, dass der Pfad gefunden wurde
}

// Dem Pfad follgen
CPathNode *GoalNode = pathMyPath.m_CompletePath->m_Path[ 0 ];
int    iTotalNodes = 0;

// Die aktuelle Position speichern um Pfeilrichtung zu bestimmen
iTempX = GoalNode->m_iX;
iTempY = GoalNode->m_iY;

// Mit Position 1 und nicht 0 starten
iTotalNodes++;
GoalNode = pathMyPath.m_CompletePath->m_Path[ iTotalNodes ];

// Durch den ganzen Pfad loopen/gehen
// Einen Pfeil bei jedem Schritt zeichnen
while( iTotalNodes < pathMyPath.m_CompletePath->m_iNumNodes ) {
 // Richtung bestimmen
 iDir = vFindDirection( iTempX, iTempY, GoalNode->m_iX, GoalNode->m_iY );

 // Den Pfeil in die Pfeilebene zeichnen
 g_iArrowMap[ GoalNode->m_iX+(GoalNode->m_iY*g_iMapWidth) ] = iDir;
```

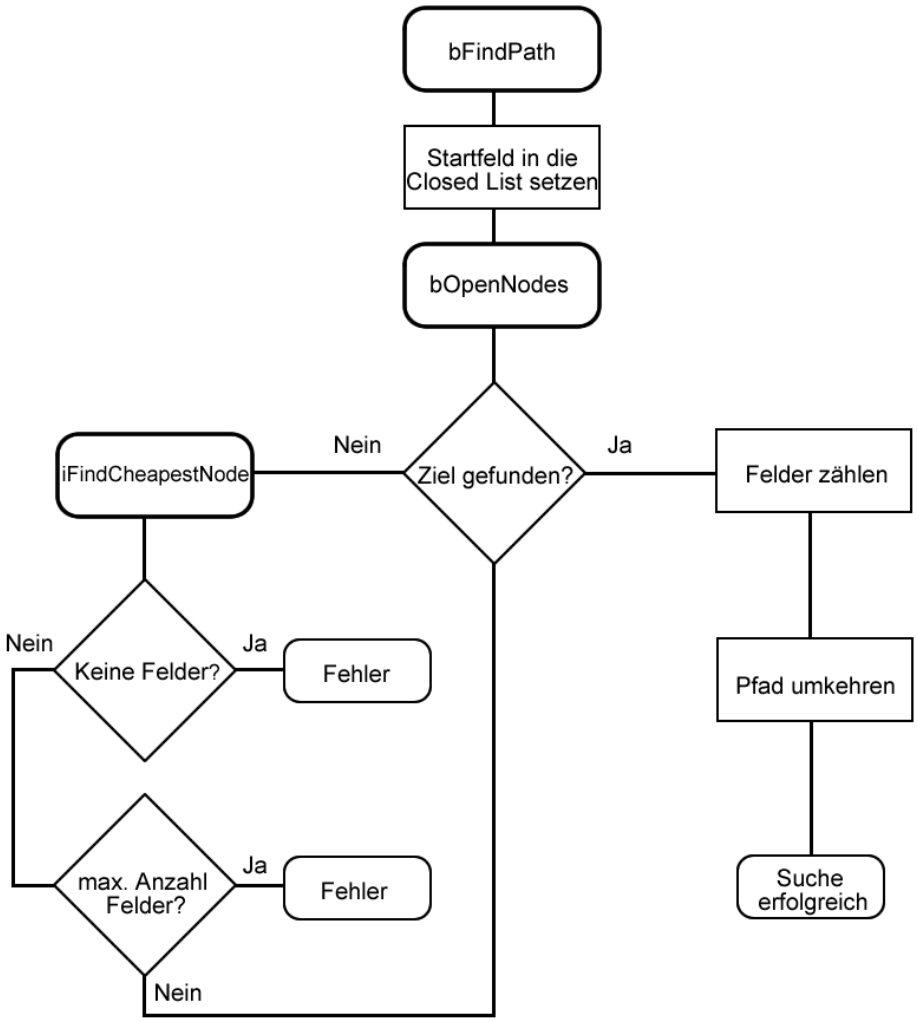
```
   // Bild rendern/zeichnen
   vRender();

   // Die aktuelle Position speichern um Pfeilrichtung zu bestimmen;(beim nächsten
Feld)
   iTempX = GoalNode->m_iX;
   iTempY = GoalNode->m_iY;

   // Anzahl Felder inkrementieren/eins dazu addieren
   iTotalNodes++;

   // Nächstes Feld/Node/Tile
   GoalNode = pathMyPath.m_CompletePath->m_Path[ iTotalNodes ];
  };
}
```

Diese Funktion ist der Grundstein. Sie wird bei einem Klick auf den Button „Pfad“ ausgeführt. Jedoch haben wir noch nicht gesehen wie der Pfad berechnet wird, sondern nur, was das alles während eines Vorganges passiert. So sehen wir aber doch schon, wie der Pfad später dargestellt wird, mit Pfeilen, welche die Richtung andeuten. Wie wir aus dem Schema oben und dem Code unschwer erkennen können, ist die Funkion bFindPath(), das Herzstück des Programms. Nach der Ausführung dieser Funktion haben wir einen kompletten Pfad. Schauen wir uns dieses Meisterwerk ein wenig genauer an.


bFindPath
Startfeld in die Closed List setzen
bOpenNodes
Ziel gefunden?
Nein
Ja
iFindCheapestNode
Felder zählen
Nein
Keine Felder?
Ja
Fehler
Pfad umkehren
max. Anzahl Felder?
Ja
Fehler
Suche erfolgreich
Nein

```
//
// Den besten Pfad von A nach B finden
//
bool CPathFinder::bFindPath( int iMaxNodes )
{
 CPathNode *GoalNode;
 int    iCurNode = 0;

 // Das Startfeld in die Closed List setzen
 m_ClosedList[ 0 ].m_iEnumerated = 1;
 // Koordinaten zuweisen
 m_ClosedList[ 0 ].m_iX = m_iStartX;
 m_ClosedList[ 0 ].m_iY = m_iStartY;
 // Zielkosten
 m_ClosedList[ 0 ].m_iDistFromGoalCost = iGetGoalDistCost( m_iStartX, m_iStartY,
0 );
```

Abb. 3.3.3: Schema für die Funktion FindPath; Grafik erstellt mit Fireworks 8

```
  // Gesammtkosten
m_ClosedList[ 0 ].m_iTotalCost = m_ClosedList[ 0 ].m_iDistFromGoalCost;
// Den Pfad finden
  for( ;; ) {

   // begehbare Felder um das Feld
   bOpenNodes( &m_ClosedList[ iCurNode ] );

   // Überprüfen ob das Zielfeld gefunden wurde
   if( m_iGoalFound ) {
    //
    // Feldanzahl auf 0 setzen
    //
    int iTotalNodes = 0;

    //
    // Totale Anzahl Felder zählen
    //
    GoalNode = &m_OpenList[m_iGoalNode];

    while( GoalNode->m_Parent != NULL ) {
     GoalNode = GoalNode->m_Parent;
     iTotalNodes++;
    };
    // letztes Feld dazuzählen
    iTotalNodes++;

    // Anzahl felder speichern
    m_CompletePath->m_iNumNodes = iTotalNodes;

    //
    // Den Pfad umkehren und speichern
    //
    GoalNode = &m_OpenList[m_iGoalNode];

    while( iTotalNodes > 0 ) {
     iTotalNodes--;
     m_CompletePath->m_Path[iTotalNodes] = GoalNode;
     GoalNode = GoalNode->m_Parent;
    };

    return( 1 );
   }
   //Das billigste Feld auf der Open List finden und in die Closed List setzen
   iCurNode = iFindCheapestNode();

   //Überprüfen ob keine Lösung existiert
   if( iCurNode == -1 ) {
    return( 0 );
   }

   // Abbrechen, wenn die max. Anzahl Felder erreicht ist
   if( m_iActiveClosedNodes >= iMaxNodes ) {
    return( 0 );
   }
  }

  return( 1 );
}
```

Durch die doch recht ausführlichen Kommentare, sollten sich diese Codesausschnitte eigentlich von selbst erklären. Auf die einzelnen kleineren Funktionen werde ich nicht

eingehen, da ihr Name schon erklärt, was sie machen, und es keinen Sinn hat eine Funktion zu beschreiben, die gerade mal zwei Werte vergleicht.

3.3.4 Anmerkungen

Ich möchte hier noch ein paar Dinge erwähnen, die sich auf die A-Star Pfadlösung beziehen: Ich habe im Internet einige Ansätze, wie auch wenige Beispielprogramme gefunden. Jedoch gab es überall kleinere und grössere Abweichungen; So zum Beispiel, ob man die Kosten des Feldes zu den Startkosten rechnet oder nicht, oder auch wie man den Weg nochmals zurück findet. So kann man den Weg zweimal berechnen, einmal vom Start zum Ziel und einmal vom Ziel bis zum Start und diese beiden Lösungen miteinander Vergleichen. Die andere Möglichkeit ist wesentlich eleganter und sicher auch schneller. Wird ein Feld in die Open List gesetzt, so wird im gleichen Zug gleich seine Richtung ermittelt. Steht das entsprechende Feld rechts-unten vom aktuellen Feld, so bekommt es diese Richtung. Der Weg wird danach so zurückgegangen. Es gibt nicht nur eine Lösung, von der man sagen kann, das ist die A-Star Pfadlösung, sondern es gibt mehrere Möglichkeiten, welche alle den gleichen Ablauf haben.

„Der übernächste Krieg wird nur noch mit Pfeil und Bogen entschieden.“ – Albert Einstein

4. Kampfsimulation

4.1 Einleitung in das Thema der Kampfsimulation

4.1.1 Persönliches Vorwort

Seit mein Vater einen Computer angeschafft hat, und ich von meinem Cousin das erste grössere Spiel geschenkt bekam, war ich gefesselt. Zu dieser Zeit war die Zeit, die ich und meine Brüder am PC verbringen durften stark limitiert, so dass wir immer wenn die Eltern aus dem Haus gingen, um irgend etwas zu erledigen sofort an den Computer, einer spielte, einer hielt Wache und der letzte durfte zusehen.
Dieses Spiel war das berühmte „Age of Empires I“, ein frühes Strategiespiel. Schon bald wird der dritte Teil dieser Serie erscheinen, und diese grossartige Tradition fortsetzen. Nach diesem Spiel war ich ein grosser Bewunderer dieses Genres, der Strategiespiele und soweit ich mich erinnere wollte ich immer einmal selber ein Spiel dieser Art programmieren lernen. Da ich damals nicht die Kenntnisse hatte, dieses Ziel zu erreichen, beschloss ich vorerst möglichst viele Strategiespiele zu spielen, um so die Komponenten und den Aufbau eines solchen Spiels in Herz und Nieren einzuschliessen.
Ich musste lange überlegen, über was ich für eine Maturaarbeit schreiben wollte, da meine Interessen sehr weit gefächert sind. Ich habe einen ähnlichen Schulweg, wie Stephan, und als wir eines Tages von der Schule heimfahren, und wir über unsere Themenwahl reden, kommt die Idee auf, zusammen die Teile einer RTS-Engine zu programmieren.
Von dieser Entscheidung an sind wir immer Schritt für Schritt vorwärtsgegangen, bis wir jetzt hier stehen und auf diese Entwicklung zurückblicken können.

4.1.2 Zielsetzung

Nach dem ersten Teil der Arbeit, der Pfadfindung, welche von Stephan programmiert wurde, übernahm ich mit dem programmieren des Kampfscriptes den zweiten Teil. Mein Kampfscript sollte, nach seiner Fertigstellung, fähig sein, zwei Einheiten zu generieren, diese Darzustellen und gegeneinander Antreten zu lassen.

4.1.3 Problemerarbeitung

Alle Spiele haben eine gemeinsame Grundlage, nämlich, dass sie eines oder mehrere Spielziele vorgeben. Diese müssen die Spieler innerhalb eines genau bestimmten Rahmens möglichst schnell erreichen.
In einem Memory gewinnt man, wenn man am meisten Kartenpaare aufdeckt, in den Kampfbasierenden Strategiespielen gewinnt man, indem man mit seiner Strategie entweder die besseren oder mehr Soldaten rekrutiert, um mit diesen dann durch ihren Einsatz, ähnlich wie beim Schachspielen, über den oder die Gegner triumphieren kann. Im Grossteil der Strategiespiele hat man zu Beginn nur eine kleine Basis in einem unbekannten Gebiet. Von hier aus muss man das Land erforschen, Rohstoffquellen sichern und abbauen und die Basis ausbauen, damit man ein Heer aufstellen kann. Man muss seine „Bauern“ auf einem komplizierteren Schachbrett verschieben, damit man eine gute Ausgangsposition für die Schlacht und den Krieg erlangt. Um dies zu machen, muss der Spieler zunächst einmal visualisiert bekommen, was das Spiel gerade am machen ist, was das Spiel mit seinen Abertausenden von Variabeln eigentlich darstellen will, und wie sich die Interaktionen des Spielers auf diese auswirken.
Nach dieser Phase, auf die je nach Spiel mehr oder weniger Schwerpunkte gelegt werden, kommt der direkte Konflikt zum tragen. Die Spieler müssen nicht nur gute Wirtschafter sein, um sich eine gute Ausgangslange zu schaffen, sie müssen diese schlussendlich auch nutzen können.
Die verschiedenen Spiele dieser Gattung weisen eine grosse Vielfalt an Taktiken und Möglichkeiten, nach welchen gearbeitet werden kann, auf. So kann ein Spieler

versuchen, den sogenannten „Schäferzug“ [1] anzuwenden, was ein Gegner seinerseits aber leicht verhindern kann.
Wer durch alle diese Schlachten hindurch den Kontrahenten ausrotten oder ein anderes Spielziel erreichen kann, hat die Partie gewonnen.

4.1.4 Einheiten

Da der Kriegerische Teil hochgerechnet etwa die Hälfte des Spiels ausmacht, drängt sich die Frage auf, wie in einem Spiel solche Schlachten simuliert werden können und wie ein solches virtuelles Heer organisiert ist.
Die Antwort praktisch aller Strategiespiele ist: „Einheiten“. Auf dem Schachbrett sind die Einheiten die verschiedenen Figuren, die Bauern, die Türme, die Springer, die Läufer und schliesslich die Königin und der König. In Strategiespielen hingegen können Einheiten Panzer, Ritter, Fussoldaten, Katapulte und Helden sein. Jedes Strategiespiel hat je nach Thema seine eigenen Einheiten. Ohne Einheiten kein Heer und ohne Heer keine Schlacht und ohne Schlacht kein Sieg.
Die für ein Strategiespiel wichtigen Attribute einer Einheit an einem möglichen Beispiel[2] anhand eines Philiisterkriegers sind:

[1] Der Schäferzug ist beim Schachspiel der Versuch, in 3 Zügen das Schachmatt zu erreichen

[2] Natürlich fehlen bei diesem Beispiel alle Zahlenwerte, aber diese werden immer willkürlich (am besten so, dass es möglichst wenig Rechenaufwand gibt und einfach zu programmieren ist) neu definiert.

Abb. 4.1.1: Philisterkrieger; Quelle: Atlas der alten Welt, S. 29

Kurzschwertkämpfer	**Name**
Bodentruppen	**Bewegung**
Kurzschwert	**Offensive**
Schild und Panzerung	**Defensive**
„Menschenleben“	**Widerstand**

Der **Name** wird benötigt, damit die Spieler die Einheiten zuordnen und mit ihnen arbeiten können, sie funktionieren wie „Eselsbrücken“ für die Einheiten. Deshalb sollten Namen möglichst eingängig und beschreibend gewählt werden, wie „Kurzschwertkämpfer“ bei Age of Empires oder beim Schachspiel „Springer“ und nicht etwas unklares („Hans Rübekopf“).

Die **Bewegungsart** sagt dem Spiel, wie sich eine Einheit fortbewegt, also ob sie zum Beispiel fliegt, am Boden geht oder herumfahren kann (beim Schachspiel kann der Springer bekanntlich als einzige Einheit über andere Schachfiguren hinwegbewegt werden), und die **Geschwindigkeit** muss definiert werden, damit das Spiel berechnen kann, wie weit eine Einheit in einer Spielrunde kommen kann. Um wieder das Beispiel des Schachs aufzugreifen: Ein Bauer kann pro Zug um 1 Feld verschoben werden, ein

Turm über das Ganze Brett hinweg. Auf dem Computer kann man zusätzlich noch mit der **Rotationsgeschwindigkeit** und der **Beschleunigung** arbeiten. Stärkere Einheiten (zum Beispiel Panzer) sollten im Normalfall langsamer sein, als Schwächere, damit das Spiel ausbalanciert bleibt.

Wenn eine Einheit eine andere töten können muss, muss sie einer anderen Einheit schaden zufügen können. Der **Angriff** definiert, wie der Schaden zugefügt wird (mit einem Schwert, einer Kugel, Gas oder einer Explosion), wie der Schaden sich weiter auswirkt (auf ein bestimmtes Gebiet oder auf eine einzelne Einheit), wie gross der Betrag des Schadens ist und aus welcher Reichweite dieser Schaden zugefügt werden kann.

Die **Verteidigung** bestimmt, ob die Einheit durch diesen Angriff verletzt werden kann, und wie stark diese Verletzung oder Beschädigung ausfällt, da kein Haus und kein Panzer einfach mit Gewehrkugeln zerstört werden kann.

Schlussendlich braucht man noch die **Widerstandskraft**, welche eine Aussage darüber macht, wieviel Schaden die betroffene Einheit erhalten kann, ohne zu sterben. Sobald dieser Wert überschritten wird, kann diese Einheit keinen Einfluss mehr auf das Spiel nehmen und verschwindet, wie beim Schachspiel geschlagene Einheiten von der Spielfläche entfernt werden.

Dies ist nur ein einfacher Umriss des Systems, es kann beliebig erweitert werden mit verschiedenen Angriffsarten, oder Spezialeffekten. Wenn man ein SciFi- Strategiespiel hat, dann kann man beispielsweise Schilde einführen, die energetische Angriffe abblocken können, aber dann sollte man auch Waffen wie zum Beispiel eine EMP-Granate haben, die speziell effektiv gegen Schilde eingesetzt werden können. Hier können die Entwickler ihre Phantasie einsetzen, um „ihr“ Spiel anders als alle anderen zu gestalten. Deshalb kann man keine hundertprozentig genaue Beschreibung aller möglichen Attribute liefern, aber die hier beschriebenen sind die für eine einfache Kampfsimulation benötigten Attribute.

Wenn man nun diese Einheiten alle definiert hat, kommt der nächste Schritt, die taktischen Möglichkeiten, die das Spiel anbietet.

Wenn man aus dem Hinterhalt angegriffen wird, ist der Schaden wahrscheinlich grösser, als wenn zwei Heere sich direkt gegenüberstehen. Genauso machen, wie schon bei der Einheitenverteidigung erwähnt, nicht alle Einheiten allen Einheiten den gleichen Schaden. Ein normaler Fussoldat kann ein Flugzeug kaum verwunden. Wenn eine Einheit von einer höher gelegenen Position aus kämpft, dann ist sie normalerweise im Vorteil.

Dies sind zwar alles nur kleine Details, aber wenn die Spieler diese abhängig von ihrer Taktik nutzen können, dann ist schon ein gutes Spielgefühl erreicht. Andererseits muss das alles aufwendig programmiert werden, aber genau diese Details machen unter dem Strich die gelungene Kampfsimulation aus.

Um die Einführung in das Konzept der Einheiten abzuschliessen, werde ich noch auf drei Beispiele aus bekannten Strategiespielen eingehen und einige Details zu deren Kampfengines Erläutern.

4.1.5 Age of Empires II

Age of Empires II ist eines der bekanntesten und, meiner Meinung nach, auch eines der besten Strategiespiele, die es je gegeben hat. Auf diesem Screenshot[3] habe ich eine kleine gestellte Schlachtszene gestellt, in welcher der Spieler als Teutone gegen sarazenische Kamelreiter antritt. Obwohl der Spieler hier zahlenmässig untergegen ist, wird er diese Schlacht gewinnen, da die Deutschritter zu den stärksten Einheiten gehören.

Abb. 4.1.2 Ingame Screen aus „Age of Ampires“

Man kann die eigenen Einheiten von den gegnerischen gut unterscheiden, da man selber blau und der Gegner rot „gefärbt“ ist. Die Einheit, über deren Kopf ein grüner Balken erscheint, das ist die angewählte Einheit, dieser Einheit kann man per Mausklick Kommandos erteilen.

[3] Ein Screenshot ist eine digitale Abbildung des Bildschirms zur Laufzeit

Abb.4.1.3: Bildausschnitt aus „Age of Ampires“

Auf dem oberen Bildausschnitt kann man den Infobalken, der in fast jedem Strategiespiel eingebaut ist, besser erkennen.

Man sieht zum Beispiel auf dem rechten Bildteil, dass die angewählte Einheit „Elite-Deutschritter“ genannt wird, eine Kampfkraft von 17+4 und eine Panzerung von 10+3/2+4 besitzt. Der aktuelle Gesundheitszustand ist 92/100, der Besitzer ist der Spieler, der die Einheit herumkommandieren kann, heisst „RamonKahn“ und sein Volk sind die „Teutonen“.

Daraus kann der erfahrene Spieler alle relevanten Informationen über die Einheit gewinnen. Er weiss, dass die Einheit insgesamt 100 „Lebenspunkte“ besitzt, und dass sie leicht verwundet ist (sie besitzt nur noch 92 Lebenspunkte). Er kann auch erkennen, dass die Einheit keine Fernkampfeinheit ist, da sie keine Reichweite aufweist. Mit einem Schlag mit der Waffe wird standardmässig 17 Schadenspunkte zugefügt, aber da der Spieler spezielle Technologien erforscht hat, zusätzlich noch einmal 4 Schadenspunkte. Die Panzerung der Einheit gegen Nahkampfangriffe ist 13 (10+3) und gegen Fernkampfeinheiten ist er nur 6 (Der Schaden, den die Einheit von einem Nahkampfangriff der Stärke 20 nehmen würde ist 7; Mindestschaden pro Angriff ist aber immer 1 Schadenspunkt).

Auf der linken Seite erkennt der Spieler weiterhin, dass die Einheit ein aggressives Kampfverhalten aufweist, sie wird jeden Gegner, den sie „sieht“ sofort angreifen und verfolgen. Man kann auch Kommandos erteilen wie Stop, Selbstmord, Einquartieren und Bewachen.

4.1.6 Empire Earth

Empire Earth ist ein erwähnenswertes Strategiespiel, da es äusserst viele Einheiten besitzt (ich kenne kein anderes Strategiespiel mit einer derartigen Einheitenvariation) und da es eines der ersten 3-D[4] Strategicals war. Im Vergleich zu Age of Empires II, welches im Mittelalter spielt, kann man hier das Spiel von einer prähistorischen Epoche bis in die nahe Zukunft dauern lassen. Deshalb hat es natürlich neue Einheiten wie Flugzeuge, die es im Mittelalter natürlich nicht gab.

Abb. 4.1.4: Ingame Screen aus „Empire Earth“

Auf diesem Screenshot kann man einige Flugzeuge sehen, die gegen mobile Flakstellungen ankämpfen.

[4] 3-D: Das Spiel wird vom Computer in 3 Dimensionen berechnet und in Echtzeit gerendert

Es ist natürlich verständlich, dass Flugzeuge gegen Flakstellungen keine Chance haben können. Im richtigen Spiel ist es so, dass man gegen 3 Flaks, die zusammen so billiger als ein einziges Flugzeug sind, von 15 Fliegern etwa 4 verliert.

Abb. 4.1.5: Bildausschnitt

Auf diesem Bildausschnitt sieht man wieder in einem kleinen Infoschirm die nötigsten Informationen über diese „Skywatcher FA“, ein Name so banal wie einleuchtend: Dass sie chinesischen Ursprungs ist, dass sie nur noch ¼ ihrer Lebenspunkte besitzt, dass sie 100 Schaden pro Schuss austeilt, 7 Reichweite hat, eine Geschwindigkeit von 12 besitzt und mit einer Kugelpanzerung von 10 ausgerüstet ist.

Auch hier könnte der Spieler Upgrades[5] kaufen, die bessere Werte versprechen würden. Auf dem Schlachtenbildschirm sieht man zusätzlich, dass die Einheit am rauchen ist, ein Hinweis, dass sie bald zerstört sein wird.

Trotz allen Neuerungen hat sich der Ablauf der Kampfsimulation nicht gewandelt. Bei diesem Spiel kommt das sogenannte „Schere-Stein-Papier“-Prinzip sehr stark zur Geltung. Es gibt immer ganze Kreisläufe von Einheiten, die einander „Schlagen“, also ohne relevante Verluste die Schlacht gewinnen. So muss der Spieler sein Heer vorausplanend zusammenstellen. Wenn man zum Beispiel mit Bodentruppen angreift, darf man die Flakstellungen nie vergessen, oder die Flugzeuge des Gegners werden einen schnell zugebombt haben.

[5] Upgrades vom englischen, bedeutet Aufbesserung eines Kampfwertes der Einheit

4.1.7 Command and Conquer: Generals

Im deutschen Sprachraum heisst das Spiel „Command and Conquer: Generäle“, und musste wegen seiner „Realitätsnähe“ zensiert werden. Es fällt auch hier auf, dass die Änderungen vor allem graphischer Natur sind, leuchtendere Effekte, detailliertere Modelle. Auch wenn dem Spieler hier weniger Zahlen gezeigt werden, ist es trotzdem immer noch genau dasselbe System

Abb. 4.1.5: Ingame Screen aus „Command and Conquer: Generals“

Hier ist ein Chinesischer „Overlord“ dargestellt (wird so genannt, weil es der stärkste Panzer im Spiel ist), der einen Bunker beschiesst. Seine Granate wirbelt soviel Staub auf, dass man kaum noch etwas sieht. Man sieht, wie der Bunker „heisst“ und wieviel

Widerstandskraft dieser hat, da die Maus über ihm ist und die Gesundheit des Panzers, da er angewählt ist. Neu ist auch, dass der Panzer sich ausrichten muss, um zu schiessen und dass er aus mehr als einem Punkt sein feuern kann (aus den beiden Hauptgeschützrohren und dem aufgebauten Geschütz auf seinem Turm, welches sich unabhängig von diesem ausrichten und feuern kann.

4.2 Umsetzung

4.2.1 Definition der Einheiten

Nachdem ich im früheren Kapitel das Prinzip der Einheiten vorgestellt habe, will ich jetzt noch zeigen, wie man dieses in einem Programm umsetzen kann. Um das System möglichst einfach und flexibel zu halten, wird am besten eine eigene Klasse für die Einheitenorganisation definiert. In dieser Klasse definiert man alle Attribute einer Einheit, und zusätzlich noch einige Funktionen, um diese zu initialisieren, wie zum Beispiel das Laden der Einheiten aus einer Datei, in welcher diese definiert sind. Am Beispiel der "Offense"- Klasse, des Angriffs werde ich vorstellen, wie dies gemacht wird, schlussendlich laufen alle Klassen danach auf dasselbe hinaus. Man könnte jeder Einheit einzeln alle Werte zuweisen, die sie für den Kampf braucht, also ihre Angriffskraft, die Angriffsweise, ihre Reichweite und was man sonst noch braucht, aber es ist sinnvoller, dafür eine Klasse zu definieren. Wenn man dies so macht, kann man allen Einheiten einen "Pointer" zu dieser Klasse zuweisen und so auf die Werte zugreifen, aber nicht jede Einheit braucht diese Variabeln einzeln zu haben. Dann definiert man alle verschiedenen Angriffsklassen, die es gibt, aber für Einheiten mit derselben Waffe, oder Einheiten, die mehrfach vorkommen, spart man Arbeitsspeicher.

```
class CUnitOffense
{
 public:
  int          m_iType;
  unsigned int m_iMissileDamageRating;
  unsigned int m_iBulletDamageRating;
  unsigned int m_iLaserDamageRating;
  unsigned int m_iMeleeDamageRating;
  unsigned int m_iSplashRadius;
  unsigned int m_iRateOfFire;
  float       m_fProjectileSpeed;
  unsigned int m_iRange;
  char         m_szName[ 64 ];

 public:
  CUnitOffense();
  ~CUnitOffense();
  virtual void vReset( void );
};
```

So sieht die Definition dieser Klasse aus. Zuerst werden alle Variabeln definiert, deren Namen selbsterklärend sind. Danach hat es drei Funktionen, eine zum Initiieren, eine zum Beenden, und eine, um alle Werte zurückzusetzen.

```
CUnitOffense::CUnitOffense()
{
 // Alle Variabeln werden zuerst auf ihre Standardwerte gesetzt.
 vReset();
}

CUnitOffense::~CUnitOffense()
{
}

//---------------------------------------------------------
// Die Variabeln auf ihre Standardwerte gesetzt.  Dadurch
// kann man die Klasse wieder löschen und neu brauchen,
// wenn dies gewünscht wird.
//---------------------------------------------------------
void CUnitOffense::vReset( void )
{
 m_iType                = 0;
 m_iMissileDamageRating = 0;
 m_iBulletDamageRating  = 0;
 m_iLaserDamageRating   = 0;
 m_iMeleeDamageRating   = 0;
 m_iSplashRadius        = 0;
 m_iRateOfFire          = 0;
 m_fProjectileSpeed     = 0.0f;
 m_iRange               = 0;
 strcpy( m_szName, "N/A" );
}
```

Wenn die Klasse initialisiert wird, werden zuerst alle ihre Werte zurückgesetzt, bei ihrer Terminierung wird gar nichts gemacht, und die letzte Funktion ist die Funktion, die man beim Initialisieren oder anderswo aufruft, um die Werte zu standardisieren, damit man diese Funktion auch unabhängig aufrufen kann, wenn man dies irgendwann einmal will. Fast alle anderen Klassen, um die Einheiten festzulegen, sind analog aufgebaut. Nur die Einheitenklasse, in welcher jede Einheit während dem Spielverlauf gespeichert wird, ist noch einmal anders. Da es jedoch zu aufwendig wäre, in jedes Detail ihrer Programmierung zu sehen, liefere ich nur einen kürzeren Beschrieb ihrer Funktionen und Variabeln.

```
class CUnit
{
 public:
  // Die Pointer zu den vordefinierten Klassen für
  // die Verteidigung, die verschiedenen Angriffe,
  // die Bewegung und die Animation.
  CUnitDefense *m_Defense;
  […]

  // Alle Variabeln, die die Einheit für eine
  // Kampfsimulation in einem Strategiespiel
  // benötigt.
  int     m_iUnitID;
  […]

 public:
  CUnit();
  ~CUnit();
  virtual void vReset( void );
  virtual void vSetBaseValues( CUnitDefense* ptrDef,
            CUnitOffense* ptrOff1,
            CUnitOffense* ptrOff2,
            CUnitOffense* ptrOff3,
            CUnitMovement* ptrMove,
            CUnitAnimation* ptrAnim );

  virtual void vSetPosition( float fX, float fY );
};
```

Ich habe hier die Variabelndefinition aus Platzgründen gelöscht, diese ist im Quellcode ersichtlich. Es hat wieder die definierende und die terminierende Funktion, die "Reset"-Funktion. Zusätzlich hat es die Funktion "vSetBaseValues", welche die Pointer der Einheit zu ihren Klassen definiert. Die Funktion "vSetPosition" kann der Einheit eine beliebige Position auf der Karte zuweisen.

Um diesen Teil des Programms abzuschliessen wird nur noch die "CUnitManager"-Klasse benötigt, welche die Einheiten auf der globalen Ebene organisiert (die "CUnit"-Klasse organisiert die Einheiten auf der individuellen Ebene). Diese beinhaltet auch das Einlesen der Einheiten aus den jeweiligen Quelldateien. Auch hier liest man am besten den Quellcode durch, um sich einen guten Überblick zu verschaffen. Nun sind die strukturellen Grundlagen einer Kampfsimulation, schon fast eines ganzen Strategiespiels geschaffen, aber andererseits hat dieser Programmteil nichts an sich, was man später im Spiel bemerken kann. Deswegen ist es wichtig, schon vor dem Schreiben der ersten Codezeile sich ein genaues Konzept zu erstellen, ansonsten kann man nicht effizient weiterarbeiten. Aber man darf auch nicht erwarten, dass das Konzept schon perfekt ist, man wird immer nachträglich verändern und verbessern müssen, aber ohne das Konzept bringt man keinen geordneten Verlauf zustande, ohne einen allzu grossen Aufwand zu betreiben.

4.2.2 Darstellung

Auch hier will ich nicht zu stark ins Detail gehen, da dies zu weitschweifig würde. Die Darstellung wird berechnet, indem man immer wieder (je nach dem, wie schnell es der Computer erlaubt) alles Sichtbare in dem Fenster der Simulation zeichnet . Zusätzlich wird in bestimmten Intervallen jede Einheit durchgerechnet und alles nötige aktualisiert (Was die Einheit gerade macht, wo sie dargestellt werden muss, u.s.w.). Wenn man dies genau wie das Rendern der Graphik andauernd machen würde, liefe das Spiel zum einen langsamer, da es mehr zu tun hätte, und wenn mit der Zeit die Computer schneller werden (in ein paar Jahren), wird das Spiel so schnell durchgerechnet werden, dass man gar nicht mehr sehen kann, was passiert, und was man machen kann.

```
void CD3DFramework::vDrawUnit( float fXPos, float fYPos, float fXSize, float fYSize,
float fRot, CUnitAnimation *animObj, int iTexture, int iOwner )
{
 //D3DX-Matrizen Definieren
 D3DXMATRIX matWorld;
 D3DXMATRIX matRotation;
 D3DXMATRIX matTranslation;
 D3DXMATRIX matScale;

 // Standartwerte für Position, Darstellungsgrösse und Rotation setzen
 D3DXMatrixIdentity( &matTranslation );
 // Das Einheitentile skalieren...
 D3DXMatrixScaling( &matScale, fXSize, fYSize, 1.0f );
 D3DXMatrixMultiply( &matTranslation, &matTranslation, &matScale );
 // ... und rotieren
 D3DXMatrixRotationZ( &matRotation, (float)DegToRad( -fRot ) );
 D3DXMatrixMultiply( &matWorld, &matTranslation, &matRotation );

 // Einheitentile bewegen
 matWorld._41 = fXPos-0.5f; // X-Pos
 matWorld._42 = fYPos+0.5f; // Y-Pos

 // Matrix definieren
 m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
 // Den "Tile Vertex Buffer" benützen
 m_pd3dDevice->SetStreamSource( 0, m_pVBUnit, 0, sizeof(TILEVERTEX) );
 // Das "Tile Vertex Format" benützen
 m_pd3dDevice->SetFVF( D3DFVF_TILEVERTEX );

 //
 // Das Bild der Einheit rendern
 //

 // Textur bestimmen
 m_pd3dDevice->SetTexture( 0, animObj->m_Textures[ iTexture ].m_pTexture );
 // Darstellen
 m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );

 //
 // Die Besitzerfarbe rendern
 //

 // Textur bestimmen
 m_pd3dDevice->SetTexture( 0, animObj->m_Textures[ iTexture+iOwner+1 ].m_pTexture );
 // Darstellen
 m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );

 // Textur entleeren
 m_pd3dDevice->SetTexture( 0, NULL);
}
```

Auf diesem Codebeispiel ist das Darstellen einer Einheit abgebildet. Diese Funktion muss für jede Einheit und für jedes dargestellte Bild aufgerufen werden. Dann übergibt man der Einheit die benötigten Werte, die X- und die Y- Koordinate, die Skalierung, die Rotation, die Animation, die Textur und den Besitzer . Sobald man diese Werte mit der jeweiligen Matrix auf die absoluten Positionen umgerechnet hat, kann man die Einheit sehr einfach darstellen.

Genauso wird die Karte dargestellt, für Details kann man auch hier den Quellcode konsultieren. Danach hat man die für den Spieler sichtbare Komponente schon. Was noch bleibt ist das andauernde Aktualisieren der Einheiten und schlussendlich noch die KI , um die Werte so zu verändern, dass es eine funktionierende Simulation ergibt.

4.2.3 Die Aktualisierung der Einheiten

```
void CD3DFramework::vUpdateUnits( void )
{
 // VariabelnDefinition
 CUnit *ptrUnit;
 static DWORD   dwLastHealTime = 0;

 m_iAction=0;

 // Alle verfügbaren Einheiten durchrechnen
 for( int i = 0; i < m_UnitManager.m_iTotalUnitObjs; i++ ) {

  // Einen "Pointer" zu der aktuellen Einheit initiieren
  ptrUnit = &m_UnitManager.m_UnitObjs[ i ];

  // Nur wenn die Einheit "aktiv" ist, weiterrechnen
  if( ptrUnit->m_bActive ) {

   //_____________________Regeneration der HP
   if( […] ) { […] }


   // Soll die Einheit an einen anderen Punkt in der Karte?
   if( [...] ) // [...]
   {
    // Winkel berechnen mit Vektorgeometrie
    […]
    m_fWinkel = 180/PI * ( m_fd + 0.5*PI - m_fn3 );

    // Muss sich die Einheit drehen?
    if( icompare( ptrUnit->m_fRot, m_fWinkel, 5 ) == 0 ) {
     ptrUnit->m_fRot += ptrUnit->m_Movement->m_fTurnSpeed; [...];}
    } else { // Wenn nein Einheit bewegen
       ptrUnit->m_iCurMoveFrame++;
       if( ptrUnit->m_iCurMoveFrame >= ptrUnit->m_Animation->m_iNumMoveFrames ) {
        ptrUnit->m_iCurMoveFrame = 0;
       }
       ptrUnit->m_iCurAnimFrame =
        ptrUnit->m_Animation->m_iStartMoveFrames +
        (ptrUnit->m_iCurMoveFrame*(UNITMANAGER_MAXOWNERS+1));

       ptrUnit->m_fmovXY =  (abs(ptrUnit->m_fTargetPosX-ptrUnit->m_fXPos)) /
((abs(ptrUnit->m_fTargetPosX-ptrUnit->m_fXPos)) + (abs(ptrUnit->m_fTargetPosY-
ptrUnit->m_fYPos)));
       if( ptrUnit->m_fmovXY == 1 ) { ptrUnit->m_fmovXY = 0; }


       if ( ptrUnit->m_fXPos < ptrUnit->m_fTargetPosX )
        {ptrUnit->m_fXPos += ptrUnit->m_Movement->m_fMovementSpeed*(1-ptrUnit-
>m_fmovXY) ;}
       else if ( ptrUnit->m_fXPos > ptrUnit->m_fTargetPosX )
        {ptrUnit->m_fXPos -= ptrUnit->m_Movement->m_fMovementSpeed*(1-ptrUnit-
>m_fmovXY) ;}

       if ( ptrUnit->m_fYPos < ptrUnit->m_fTargetPosY )
        {ptrUnit->m_fYPos += ptrUnit->m_Movement->m_fMovementSpeed*(ptrUnit-
>m_fmovXY) ;}
       else if ( ptrUnit->m_fYPos > ptrUnit->m_fTargetPosY )
        {ptrUnit->m_fYPos -= ptrUnit->m_Movement->m_fMovementSpeed*(ptrUnit-
>m_fmovXY );}

    }
   } else { ptrUnit->m_bKI = FALSE; }

    // Animation der Einheit
    […]
```

An diesem grösseren Codebeispiel wird dargestellt, wie die Einheit alle 33 Millisekunden aktualisiert wird. Zuerst werden die beiden benötigten Variabeln definiert, eine, um zu messen, ob die Einheit sich schon wieder heilen kann, je nach dem, wie es in der "Defense"- Klasse definiert ist, die andere Variable ist der Zeiger, der benötigt wird, um die Einheit als Referenz nehmen zu können. Die Überprüfung, ob geheilt wird oder nicht, habe ich aus Platzgründen herausgenommen, dies muss man im Code einsehen. Nun wird die berechnet, ob die Einheit überhaupt an einen anderen Punkt auf der Karte gehen will. Wenn die Einheit nirgends hin will, muss man das Folgende gar nicht berücksichtigen. Falls diese Abfrage erfolgreich war, wird die Einheit zuerst rotiert, bis sie in die richtige Richtung schaut, und danach wird die Einheit abhängig von ihrer Bewegungsgeschwindigkeit zu ihrem festgelegten Zielpunkt bewegt. Zusätzlich wird die Bewegung direkt animiert.

Der Codeausschnitt, der jetzt kommt, beschreibt die Animation der Einheiten. Da er auf dem ersten Beispiel herausgeschnitten ist, habe ich ihn jetzt nachfolgend abgebildet. Zuerst frage ich ab, ob die Einheit schon "sterben" sollte, wenn ja, wird direkt die "Animation des Todes" aufgerufen, welche aber erst am Ende im Code ist. Andernfalls wird die Einheit je nach ihrer momentanen Eingestellung animiert. Nun könnte man meinen, dass, da die Einheit schon bewegt wurde, und dort schon animiert wurde, es jetzt einen Fehler geben könnte, weil plötzlich zwei Animationen berechnet werden. Aber da dieser Codeausschnitt nie aufgerufen wird, solange die Einheit fortbewegt wird, tritt dieser Fall nicht ein.

```
// Animation der Einheit
     if( ptrUnit->m_iAnim != 4 )
     {
      // Stillstandsanimation
      if( ptrUnit->m_iAnimx == 3 || ptrUnit->m_iAnimx == 0 ) {
       ptrUnit->m_iCurStillFrame++;
       if( ptrUnit->m_iCurStillFrame >= ptrUnit->m_Animation->m_iNumStillFrames ) {
        ptrUnit->m_iCurStillFrame = 0;
       }
       ptrUnit->m_iCurAnimFrame =
        ptrUnit->m_Animation->m_iStartStillFrames +
        (ptrUnit->m_iCurStillFrame*(UNITMANAGER_MAXOWNERS+1));}
      // Einheiten abfeuern
      else if( ptrUnit->m_iAnimx == 2 ) {
       ptrUnit->m_iCurAttackFrame++;
       //_____Wenn EinheitAttAnimation fertig, wieder auf "IdleAnimation"
setzen...
       if( ptrUnit->m_iCurAttackFrame >= ptrUnit->m_Animation->m_iNumAttackFrames )
{
        ptrUnit->m_iCurAttackFrame = 0; ptrUnit->m_iAnimx = 0;
       }
       ptrUnit->m_iCurAnimFrame =
        ptrUnit->m_Animation->m_iStartAttackFrames +
        (ptrUnit->m_iCurAttackFrame*(UNITMANAGER_MAXOWNERS+1));
      }
     } else {
      // Einheit sterben lassen
      ptrUnit->m_iCurDieFrame++;
       //_____Wenn EinheittotAnimation fertig ist, einheit löschen
       if( ptrUnit->m_iCurDieFrame >= ptrUnit->m_Animation->m_iNumDieFrames ) {
        ptrUnit->m_bActive = FALSE;
       }
       ptrUnit->m_iCurAnimFrame =
        ptrUnit->m_Animation->m_iStartDieFrames +
        (ptrUnit->m_iCurDieFrame*(UNITMANAGER_MAXOWNERS+1));
[…]
```

4.2.4 Die KI und die Kampfsimulation

Auch hier möchte ich den benötigten Code kurz erläutern. Der erste Teil der Funktion ist praktisch gleich, wie bei der "vUpdateUnits" - Funktion. Es wird wieder eine Schleife, die jede Einheit abtastet gestartet, und danach wird abgefragt, ob die Einheit aktiv ist, und der Pointer gesetzt. Danach kommt der eigentliche "KI"- Teil, in welchem ausgerechnet wird, welche Einheit eines anderen Spielers am wenigsten weit entfernt von der aktuellen Einheit ist. In einem richtigen Strategiespiel müsste man dies nur in einem bestimmten Umfeld machen, da eine Einheit normalerweise nicht selber eine Einheit sucht, und diese dann angreift, sondern sich nur wehrt, wenn sie oder ein Freund angegriffen wird, und nur in einem bestimmten Radius selber nachschaut, ob sie einen Gegner angreifen kann. Wenn die nächste Einheit gefunden ist, wird diese

als Ziel festgelegt, und der zweite Teil der Funktion, die eigentliche Kampfsimulation wird aufgerufen.

```
void CD3DFramework::vUpdateUnitKI( void )
{
 CUnit *ptrUnit;
 CUnit *ptrUnitn;

 // Alle verfügbaren Einheiten durchrechnen
 for( int i = 0; i < m_UnitManager.m_iTotalUnitObjs; i++ ) {

  // Einen "Pointer" zu der aktuellen Einheit initiieren
  ptrUnit = &m_UnitManager.m_UnitObjs[ i ];

  // Nur wenn die Einheit "aktiv" ist, weiterrechnen
  if( ptrUnit->m_bActive ) {
   // Kein Ziel? -> Ziel bestimmen
   if( ptrUnit->m_iTargetID == -1 )
   {
    // nächste Einheit  bestimmen (eines anderen Spielers)
    for( int n = 0; n < m_UnitManager.m_iTotalUnitObjs; n++ ) {
     ptrUnitn = &m_UnitManager.m_UnitObjs[ n ];
     if( ptrUnitn->m_bActive == TRUE ) {
     if( ptrUnit->m_iOwner != ptrUnitn->m_iOwner ) {

      m_fsD = sqrt( pow( ( ptrUnitn->m_fXPos - ptrUnit->m_fXPos ), 2) +
pow( ( ptrUnitn->m_fYPos - ptrUnit->m_fYPos ), 2) );
      if( m_fsMDist == 0 ) {m_fsMDist = m_fsD; }
      if( m_fsMDist <= m_fsD ) { ptrUnit->m_iTargetID = ptrUnitn->m_iUnitID; }
     }}}

   }
```

Der Codeausschnitt oben ist die Zielbestimmung und nachfolgend zu diesem Text ist der zugehörige Codeausschnitt. In diesem Programmteil wird der Kampf simuliert, indem man von den "HitPoints" der feindlichen Einheit den nicht von den Schilden abgewehrten Schaden abzieht. Da es 4 vordefinierte Schadenstypen gibt, muss das ganze vier Mal berechnet werden.

Dem aufmerksamen Leser ist vielleicht aufgefallen, dass die Einheiten drei verschiedene "Offense"- Klassen besitzen, in der Kampfsimulation aber nur auf die erste zurückgegriffen wird. Das ist so, weil es in einem Strategiespiel eher dazu verwendet würde, je nach Wunsch des Spielers eine dieser Waffen auszuwählen, und dann könnte man vielleicht noch eine begrenzte Munition einführen, um das ganze System noch spannender zu gestalten. Aber in diesem Beispiel der Kampfsimulation habe ich der Einfachheit halber darauf verzichtet, dies einzuprogrammieren. Man müsste einfach eine weitere Variable haben, welche speichert, mit was man gerade

schiesst, und dann könnte man dies abfragen und je nach dem den Schaden genau gleich ausrechnen.

```
    else{
     ptrUnitn = &m_UnitManager.m_UnitObjs[ ptrUnit->m_iTargetID ];
     m_fsDist = sqrt( pow( ( ptrUnitn->m_fXPos - ptrUnit->m_fXPos ), 2) +
pow( ( ptrUnitn->m_fYPos - ptrUnit->m_fYPos ), 2) );
     if( 13*ptrUnit->m_Offense1->m_iRange < m_fsDist ) {ptrUnit->m_fTargetPosX =
m_UnitManager.m_UnitObjs[ ptrUnit->m_iTargetID ].m_fXPos; ptrUnit->m_fTargetPosY =
m_UnitManager.m_UnitObjs[ ptrUnit->m_iTargetID ].m_fYPos; }
     else { // Die eigentliche Simulation, mit allem drum und dran...
      //Genug herumgeflogen
      ptrUnit->m_fTargetPosX = ptrUnit->m_fXPos; ptrUnit->m_fTargetPosY = ptrUnit-
>m_fYPos;
      //Zuerst mit RoF schauen, ob man Feuern kann, wenn nicht -> warten
      //Ansonsten Feuern (->Animation setzen)
      if( timeGetTime() > ptrUnit->dwLastFireTime ) {

       //Schaden abrechnen (Panzerung - Angriff; nicht kleiner 0)
       ptrUnit->m_iAnimx = 2;

       m_Dam[ptrUnit->m_iUnitID][ 0 ] = m_UnitManager.m_UnitBaseObjs[ptrUnitn-
>m_iUnitID].m_Defense->m_iBulletArmorRating - ptrUnit->m_Offense1-
>m_iBulletDamageRating;
       m_Dam[ptrUnit->m_iUnitID][ 1 ] = m_UnitManager.m_UnitBaseObjs[ptrUnitn-
>m_iUnitID].m_Defense->m_iMissileArmorRating - ptrUnit->m_Offense1-
>m_iMissileDamageRating;
       m_Dam[ptrUnit->m_iUnitID][ 2 ] = m_UnitManager.m_UnitBaseObjs[ptrUnitn-
>m_iUnitID].m_Defense->m_iLaserArmorRating - ptrUnit->m_Offense1-
>m_iLaserDamageRating;
       m_Dam[ptrUnit->m_iUnitID][ 3 ] = m_UnitManager.m_UnitBaseObjs[ptrUnitn-
>m_iUnitID].m_Defense->m_iMeleeArmorRating - ptrUnit->m_Offense1-
>m_iMeleeDamageRating;
```

Nun weiss das Programm schon, wie viel Schaden zugefügt wird, aber es hat der angegriffenen Einheit diesen noch gar nicht zugefügt. In diesem Code wird, falls der Schaden grösser als 0 ist, also die Panzerung den zugefügten Schaden nicht ganz deflektiert, nun der Schaden verrechnet. Dieses auftrennen ist nützlich, wenn man die verschiedenen "Offense"- Klassen benutzen möchte, man müsste nur die Schadensberechnung an die Bedingung knüpfen, und das Verrechnen hier könnte man genau gleich vonstatten gehen lassen.
Zuletzt wird noch berechnet, ob die andere Einheit schon gestorben ist, also sie weniger als 0 HitPoints besitzt, und wenn das zutrifft, wird die Animation, dass sie stirbt, initiiert, und beide Einheiten hören auf, einander anzugreifen. Zusätzlich wird noch von dem Einheitenlimit des Besitzers eine Einheit abgezogen, so, dass dieser wieder eine Einheit mehr bauen könnte.

```
        m_TD = 0;

        m_TD = -m_Dam[ ptrUnit->m_iUnitID ][ 0 ] ;
        if( m_TD > 0 ) { ptrUnit->m_iCurHitPoints -= m_TD; }
        m_TD = -m_Dam[ ptrUnit->m_iUnitID ][ 1 ] ;
        if( m_TD > 0 ) { ptrUnit->m_iCurHitPoints -= m_TD; }
        m_TD = -m_Dam[ ptrUnit->m_iUnitID ][ 2 ] ;
        if( m_TD > 0 ) { ptrUnit->m_iCurHitPoints -= m_TD; }
        m_TD = -m_Dam[ ptrUnit->m_iUnitID ][ 3 ] ;
        if( m_TD > 0 ) { ptrUnit->m_iCurHitPoints -= m_TD; }

        m_Dam[ ptrUnit->m_iUnitID ][ 0 ] = 0;
        m_Dam[ ptrUnit->m_iUnitID ][ 1 ] = 0;
        m_Dam[ ptrUnit->m_iUnitID ][ 2 ] = 0;
        m_Dam[ ptrUnit->m_iUnitID ][ 3 ] = 0;

        if( ptrUnit->m_iCurHitPoints < 0 ) {
         ptrUnit->m_bKI = FALSE;
         ptrUnit->m_iTargetID = -1;
         ptrUnitn->m_iAnim = 4;
         m_UnitManager.m_iOwnerTotal[ ptrUnit->m_iOwner ] -= 1; }
```

Das ist schon alles, was es braucht, um den Kampf zu simulieren, im Vergleich zu den Funktionen, um die Szene zu rendern und um die Einheiten zu Beginn zu definieren ist diese Funktion fast ein Zwerg. Und trotzdem ist das alles, was der Spieler dann sieht, wenn die Einheiten kämpfen, von diesem Ausschnitt ausgehend.

5. Schlusswort

Endlich ist diese Arbeit abgeschlossen! Wir sind himmelhoch jauchzend, zu Tode betrübt.
Nach diesem halben Jahr des Programmierens, des Schreibens und des Korrigierens können wir nun auf unsere Arbeit zurückblicken, auf den Weg, den wir gegangen sind und auf die Entwicklung, die wir durchgemacht haben.
Nicht alles lief gleich ohne Probleme, viele Codezeilen wurden geschrieben, nur um gleich wieder gelöscht und verworfen zu werden, der Computer hat nicht immer das gemacht, was er sollte und unsere Arbeitsweisen sind nicht deckungsgleich. Der eine Versucht alles möglichst früh zu machen, der andere kann seine Zeit schlechter einteilen.
Aber diese Arbeit hat auch ausser dem, dass sie fertig ist, ihre positiven Seiten, unsere Programmierkenntnisse haben sich sicherlich weiterentwickelt, und das führte dazu, dass das Programmieren je länger, desto mehr Spass machte. Deshalb hat die Arbeit ihr Ziel auch erreicht, und es ist schön, auf seine Leistung zu blicken, und stolz zu sein.
Wir möchten aber auch noch einen Rat an alle zukünftigen Schreiber einer solchen Arbeit warnen. Es ist nicht immer einfach, seine Motivation für ein solches Projekt aufrechtzuerhalten, und man steckt sich sehr schnell höhere Ziele, als man erreichen kann, und wenn man die nicht erreicht, ist das eine Motivationsbremse.
Aber wenn man am Ende auf seine fertiggestellte Arbeit zurückblicken kann, ist man befreit, und kann beruhigt zum nächsten Lebensabschnitt voranschreiten.

6. Glossar

Age of Empires: Eine Reihe von Echtzeit-Strategiespiele, welche von den *Ensemble Studios* entwickelt wurden. Mit ca. 16 Millionen verkauften Spielen gehören sie zu den erfolgreichsten Reihen überhaupt. Das erste Spiel wurde 1997 veröffentlicht, das Dritte kommt im November 2005 auf den Markt.

Command & Conquer: Eine Reihe von Echtzeit-Strategiespiele, entwickelt von den *Westwood Studios*. Das Studio wurde im Jahre 2000 von *EA* übernommen. Seit dem Jahre 2003 werden die Spiele von *EA Pacific* programmiert, diese verzeichnet aber grosse Verluste an ausgezeichneten Programmierern.

Compiler: Ein Kompilierer (*engl. Compiler = Übersetzer*) wandelt einen Quellcode einer Sprache in die Assembler- oder Maschinensprache um.

DirectX SDK: Ist eine Sammlung, Bibliothek von APIs (*application programming interface; zu deutsch: Schnittstelle zur Anwendungsprogrammierung*). Sie erleichtern das Programmieren, da viele Aktionen (z.B. Klicken mit der linken Maustaste) schon in einer Funktion beschrieben wurden. Es führt auch zu einem übersichtlicheren Quellcode.

Dune II: Ein Computerspiel aus dem Hause des französischen Spiele Hersteller und Vertreiber *Cryo Interactive Entertainment*. Es ist eine Mischung aus Strategie und Adventure.

Ego-Shooter: Computerspielegenre, bei welchem der Spieler aus der Ich-Perspektive (*lat. ego = Ich*) das Spiel betrachtet. Der Spielverlauf ist geprägt durch einen regelmässigen Konsum von Schusswaffen (*engl. Shooter = Schütze*).

Header: Das Wort Header (*engl. Head = Kopf*) bedeutet in der Welt der Programmiersprachen Kopf, im Sinne von Anfang. Eine Header-Datei besteht zu grossen Teilen aus Schnittstellendefinitionen (Anzumerken: Nur die Definitionen).

Herzog Zwei: Ein im Jahre 1989 veröffentlichtes Computerspiel aus dem Hause *Technossoft* für den *Sega-Mega-Drive*.

Havok-Physik-Engine: Die Havok-Physik-Engine ist eine Physik-Engine aus dem Unternehmen, welches denselben Namen wie das Produkt trägt. Eine Physik-Engine simuliert in einem Spiel verschiedenste Dinge. So beispielsweise, das Fahren der Fahrzeugen, die Animation von Charakteren und die Simulation physikalischer Effekte,

das Fliegen eines Balles. Die *Havok* ist die im Moment wohl beste Physik-Engine auf dem Markt.

KI: KI ist die Abkürzung für Künstliche Intelligenz. Man bezeichnet damit, das Verhalten von Computergesteuerten Objekten, meistens die Gegenspieler. Sei jetzt das in einem Strategiespiel, einem *Ego-Shooter* oder in einem Schachspiel.

Map: Das englische Wort Map bedeutet auf Deutsch, soviel wie Karte. Man bezeichnet damit, insbesondere bei Strategiespielen und *Ego-Shootern*, die vom Spieler aus gesehen sichtbare Umgebung.

Rendern: Der Begriff rendern (*engl. rendern = zeichnen*) steht für das Anzeigen der Grafiken an einem festgelegten Ort.

Quellcode: Der Quellcode ist der lesbare, noch nicht in die Maschinensprache umgewandelten, Text mit den Anweisungen für den Compiler..

Sega Mega Drive: Eine Spielkonsole des japanischen Herstellers Sega. Erschienen 1988.

Strategie: Das Wort Strategie kommt aus dem Altgriechischen und bedeutet etwa soviel wie Heer (*griech. stratos = Heer*). Erst später hat sich der militärische Begriff auch in die Umgangssprache gewagt und wird heute nur noch selten in Bezug auf ein Heer gebraucht.

Tile: Tiles (*engl. Tiles = Fliesen*) sind meist kleinere Grafiken oder Bilder, die in Computerspielen verwendet werden. Dadurch dass sie Klein sind verbrauchen sie relativ wenig Speicher, was der Spielgeschwindigkeit zugute kommt.

Visual Stduio.NET oder **Visual Studio 6.0:** Sind von Microsoft programmierte Entwicklungsumgebungen.

WarCraft: Warcraft (*engl. Warcraft = Kriegshandwerk*) ist eine seit 1994 von der Firma *Blizzard Entertainment* entwickelte und vertriebene Computerspielereihe des Genres *RTS*.

7. Literaturverzeichnis

Bücher

Adams, Jim (2002): *Programming Role Playing Games with DirectX*, 2nd ed. Thomson Course Technology PTR (Premier Press)

Prinz, Peter und Kirch-Prinz, Ulla (2002): *C++ Lernen und professionell anwenden* 2nd ed (mitp)

Saumweber, Walter (2003): *magnum C++* (Markt + Technik)

Internet

http://msdn.microsoft.com/directx/ *DirectX Download (SDK)*

http://www.codesampler.com/ *Beispielprogramme für DirectX*

http://www.codeworx.org/ *Tutorials und Infos*

http://www.codezone.ch/ *Download MS Visual Studio*

http://www.cplusplus.com/ *Seite über C++*

http://www.gamedev.net/ *Tutorials und viele Informationen*

http://www.games-net.de/ *Tutorials und Infos*

http://www.freeprogrammingresources.com/ *Viele Tutorials*

http://www.scorpioncity.com/djdirectxtut.html/ *Tutorials zu DirectX*

http://de.wikipedia.org/ *Informationen und Hilfen für das Glossar*

8. Danksagung

Edith Roth	Korrektur
Johanna Vock	Korrektur
Erika Ochsenbein	Korrektur

Unseren Eltern	Für ihre nie enden wollende Geduld mit uns, auch wenn es ab und zu mal ein wenig später wird.
Unseren Geschwistern	Für die ruhige Zeit, die sie uns gelassen haben, damit wir uns in Ruhe unseren Problemen widmen konnten.
Andreas Locher	Betreuung
Bjarne Stoustrup	Für die Entwicklung, der Programmiersprache C++
Bill Gates	Für die Gründung einer Firma die wirklich nur nervenschonende Programme erstellt. Wir danken dir, aus tiefstem Herzen und wünschen dir, auch nicht mehr Probleme mit deinen Errungenschaften, als wir sie hatten.
Allen Programmierern,	die ihre Freizeit opfern, um für uns ein *benutzerfreundliches* Endprodukt zu erstellen, das man auch gebrauchen kann, danke.
Richard Bumann	Für deine unglaubliche Motivation und dass du uns von Zeit zu Zeit auch wieder auf den Boden der Realität zurückgeholt

hast. Ohne dich hätten wir unsere doch eher hochgesteckten Ziele nicht erreicht.

CD- Inhalt

Alle verwendeten Bilder
Kampfsimulationscode
Pfadfindungscode
Quellen
Quellencode
DirectX 9 SDK
Maturarbeit