ZFX PRÄSENTIERT

Spieleentwicklung mit Direct3D

von
Stefan Zerbst

EIN eBOOK ZUM FREIEN DOWNLOAD AUS DEM INTERNET

SPIELEENTWICKLUNG MIT DIRECT3D IM

---

von Stefan Zerbst

Aus dem Inhalt

Diese Tutorial Reihe führt den aufmerksamen Leser durch die wunderbare (*komplizierte*) Welt des Immediate Mode von Direct3D Version 8 und wie wir diesen in unsere Windows Programme integrieren und nutzen können um beispielsweise unseren eigenen Quake Clone zu programmieren.

Einleitung: Inhalt und Voraussetzungen

Kapitel 1: Grundlagen der Spieleentwicklung

Kapitel 2: Eine einfache Windows Anwendung

Kapitel 3: Direct3D 8 initialisieren

Kapitel 4: Ein Dreieck rendern

Kapitel 5: Das Dreieck dreht sich

Kapitel 6: 3D Modelle laden und rendern

Kapitel 7: Bewegung des Spielers

Kapitel 8: Pointer, Arrays und 3D Goodies

Kapitel 9: BSP Bäume, 3D Indoor Level Rendering

Kapitel 10: Minikapitel: Direct3D Lichtobjekte für die Cat **by TheWanderer**

Kapitel 11: Solide BSP Bäume mit Kollisionsabfrage

(c) 2001/2002 [des gesamten Tutorials] by Stefan Zerbst
Die Texte der Tutorials unterliegen dem Copyright und dürfen ohne schriftliche Genehmigung von Stefan Zerbst weder komplett noch auszugsweise vervielfältigt, auf einer anderen Homepage verwendet oder in sonst einer Form veröffentlicht werden. Die zur Verfügung gestellten Quelltexte hingegen stehen zur freien Weiterverwendung in Software Projekten zur Verfügung.
Die Quelltexte dieses Tutorials werden auf einer "as is" Basis bereit gestellt. Der Autor übernimmt keinerlei Garantie für die Lauffähigkeit der Quelltexte. Für eventuelle Schäden die sich aus der Anwendung der Quelltexte ergeben wird keinerlei Haftung übernommen.

SPIELEENTWICKLUNG MIT DIRECT3D - EINLEITUNG

von Stefan Zerbst

Intention

Dieses neue Tutorial zur Spieleentwicklung dient vor allem einem Zweck: Anfängern auf dem Gebiet der Spieleprogrammierung einen guten Einstieg in die Materie zu bieten. Dabei konzentriert sich dieses Tutorial natürlich auf 3D Grafik unter Verwendung des Direct3D Immediate Modes der Version 8 von DirectX.

Die einzelnen Kapitel bauen logisch aufeinander auf und erklären alle Schritte die notwendig sind, um ein Programm für das Betriebssystem Windows zu entwickeln welches 3D Modelle am Bildschirm (*Vollbild*) anzeigen kann. Zu jedem Kapitel gibt es ein fertiges Projekt mit dem entwickelten Quelltext zum Download. Die hier verwendeten Programme basieren zu grossen Teilen auf den Quelltexten der Programme aus meinem Buch "3D Spieleprogrammierung mit DirectX in C/C++ - Band I", jedoch werden die hier gegebenen Erklärungen eher pragmatischer Natur sein und sich auf die 3D Spieleentwicklung konzentrieren. Ich werde mich also auf das Wesentliche konzentrieren und weniger ins Detail gehen. Das 9. und 11. Kapitel hingegen basiert auf einer Prototypstudie des Demoprogramms **Pandora's Box** welches ich für Band III meiner Bücherreihe entwickle. Es ist sozusagen ein Vorgeschmack auf das was Euch dort erwarten wird.

Voraussetzungen

Um diesem Tutorial gut folgen zu können und alles zu verstehen setze ich voraus, dass der Leser oder die Leserin bereits über ein einfaches Grundwissen in der Programmiersprache C verfügt und mit einem Compiler wie beispielsweise VC++ umgehen kann. An Wissens-Voraussetzungen ist das dann tatsächlich schon alles.

Natürlich muss man auch diverse Software-Voraussetzungen mitbringen. Zunächst benötigt man natürlich einen DirectX kompatiblen C++ Compiler. Ich verwende Microsoft Visual C++ 6.0, welcher für die Arbeit mit DirectX optimal geeignet ist. Daneben muss man auch über das DirectX SDK Version 8 verfügen (*nicht die Home-User Variante*) die man unter http://msdn.microsoft.com/directx downloaden kann.

Lernziel

Das Lernziel dieses Tutorials ist es, zunächst den Umgang mit Direct3D zu lernen. Diese API stellt eine gewaltige Menge an Methoden zur Verfügung, mit denen wir die virtuellen Phantasiewelten in unseren Köpfen auf den Bildschirm bringen können. Wir werden sehen, dass wir innerhalb von zwei Kapiteln bereits die ersten 3D Objekte am Bildschirm anzeigen können und dabei so viel gelernt haben, dass wir Direct3D gut im Griff haben. Danach befassen wir uns dann noch mit ein oder zwei Kleinigkeiten die immer noch zum Teil API spezifisch, also auf Direct3D bezogen, sind. Dazu gehören Projektion und Kamerabewegung. Wobei man hier auch viel Informationen über die Realisierung dieser Aufgaben in anderen API, wie beispielsweise OpenGL, gewinnen kann.

Das Hauptziel dieses Tutorials ist es aber, nicht den Umgang mit einer bestimmten API zu lernen, sondern die reine 3D Programmierung. Ab dem 8. Kapitel beginnen wir damit, ein wenig über den API Tellerrand hinaus zu schauen und widmen uns dann ganz und gar, abgesehen von einem kleinen Intermezzo in Kapitel 10, der Programmierung von 3D Indoor Level Renderern. Am Ende des 11. Kapitels steht dann zu guter Letzt ein 3D Indoor Level mit Kollisionsabfrage, den der motivierte Leser dann zu seinem eigenen Quake Clone ausbauen kann. Dort werden auch genügend Hinweise geliefert, wie das auszusehen hat.

Am Ende der Durcharbeitung dieses Tuorials sollte der Leser dazu in der Lage sein, mittels des Werkzeugs Direct3D, seine eigenen 3D Welten zu programmieren und eine eigene 3D Engine auf mittlerem Niveau zu programmieren.

Was ist DirectX

Bleibt nur noch die Frage zu klären: Was ist DirectX eigentlich? Hier möchte ich nicht zu sehr ins Detail gehen, da uns das eigentlich ziemlich egal sein kann. DirectX ist prinzipiell nichts anderes als eine grosse Bibliothek die mehrere kleine Bibliotheken enthält welche wiederum viele Interface-Objekte enthalten. Wenn wir also DirectX Bibliotheken in unsere Programme einbinden, so können wir diese Interface Objekte erzeugen und dann über diese Objekte verschiedene Methoden aufrufen.
DirectX beinhaltet unter anderem die folgenden Bibliotheken:

- **DirectGraphics**

Hinter DirectGraphics verbirgt sich die Funktionalität zur Grafikausgabe mit DirectX. Früher gab es die beiden getrennten Komponenten DirectDraw (*Bildschirmeinstellung, 2D Grafik*) und Direct3D (*Retained Mode, Immediate Mode*) doch seit der Version 8 von DirectX wurden diese beiden verschmolzen und es gibt nur noch Direct3D welches offiziell auch unter dem Namen DirectGraphics läuft. Über die Interface Objekte von Direct3D kann man die Bildschirmeinstellungen vornehmen, 2D und 3D Grafik anzeigen und noch vieles mehr. Direct3D wird in diesem Tutorial ausführlich behandelt.

- **DirectInput**

In der DirectInput Bibliothek befinden sich alle Interface Objekte über die wir Zugriff auf die Tastatur, die Maus und natürlich auch den Joystick haben. In dem Wing Captain 1.0 Tutorial auf der Homepage www.stefanzerbst.de findet sich eine beispielhafte Implementierung einer älteren DirectInput Version, denn in diesem Tutorial wird DirectInput nicht behandelt. Da die normalen Eingaberoutinen aber insbesondere in Kapitel 9 und 11 den Programmfluss ein wenig bremsen, lege ich jedem die zusätzliche Implementierung von DirectInput Eingabefunktionen ans Herz. Mit Hilfe der DirectX SDK Dokumentation wird das auch kein Problem sein, denn dort finden sich einige einfache Beispiele wie man Tastatur, Maus und Joysticks integriert.

- **DirectAudio**

Die DirectAudio Bibliothek beinhaltet die beiden Komponenten DirectSound und DirectMusic mit deren Hilfe wir Soundeffekte oder Musik in unser Programm integrieren können. Daneben kann man aber auch professionelle Anwendungen zum Bearbeiten von Sound oder Musik erstellen. DirectSound wird ebenfalls in einer älteren Version in dem Wing Captain Tutorial auf dieser Homepage implementiert und wir werden es hier nicht weiter behandeln.

Jetzt sollten wir also einen guten Überblick darüber haben, was DirectX ungefähr ist. Wir wir es anwenden können werden wir gleich in Life-Fire Beispielen für Direct3D sehen. Zunächst aber beginnen wir damit, ein einfachen Windows Programm als Basis für unser Spiel zu entwerfen.

Auf geht's...

von Stefan Zerbst

"She always favours the blunt approach," Western Europe said. "We all know that. That's why I like her so much, makes one feel constantly superior."
Night's Dawn Trilogy, Peter F. Hamilton

Grundlagen der Spieleentwicklung

Da sind wir also, mitten im ersten Kapitel dieses Tutorials in dessen Verlauf wir lernen wollen, 3D Spiele mit Hilfe von Direct3D Immediate Mode zu entwickeln. Dieser Abschnitt dient aber erst mal dazu, ein gewisses *Feeling* für die Sache zu erzeugen. Insbesondere ist die Erschaffung eines Spiels wesentlich mehr, als nur die reine Programmierarbeit oder das Erstellen hübscher Grafiken. Daher bevorzuge ich es, das Wort Spieleentwicklung dem Begriff Spieleprogrammierung vorzuziehen. Die folgenden Abschnitte zeigen zunächst nur die elementaren Grundlagen der Spieleentwicklung, die einem angehenden Spieleprogrammierer direkt ins Blut übergehen sollten. Also Augen und Ohren auf!

Einleitende Worte

Am Anfang schuf der Programmierer die Idee... Für Anfänger schwer zu glauben, aber bereits hier können die ersten Hürden auftreten. Grundlage für jedes Spiel ist eine gute Idee, die man auf alle Fälle systematisch und nicht aus dem Bauch heraus entwickeln sollte. Ebenso unverzichtbar ist das Aufschreiben aller Ideen und Konzepte. Idealerweise entwirft man also ein sogenanntes **Entwicklungsskript**, in dem alle Ideen schriftlich festgehalten werden. Die Idee hinter dem Entwicklungsskript ist die strukturierte Entwicklung von Software zur Vermeidung von Fehlern.

In den 70'er Jahren bestanden Computerprogramme noch aus Hunderten von Lochkarten und die Berechnung eines Programmes dauerte natürlich wesentlich länger als heutzutage. Daraus resultierte in erster Linie, daß Rechenzeit an einem Computer teuer war. Niemand konnte sich die heute leider gängige Methode der **Trial and Error** Programmierung leisten. Mal eben schnell den Wert einer Variablen zu ändern oder eine Schleife einen Index länger laufen zu lassen konnte sich niemand leisten – weder finanziell noch zeitlich. Zu dieser Zeit war es nur natürlich, sich vor der eigentlichen Programmierung Gedanken über das Programm zu machen. Heutzutage wird eher **Quick and Dirty** programmiert. Da die Computer immer billiger wurden und die Rechenleistung sich ständig potenziert, kann man heutzutage eben doch den Prozess des Ausprobierens und Neukompilierens innerhalb weniger Sekunden vollziehen. In der Industrie wird daneben auch betriebswirtschaftlich argumentiert, daß das Trial and Error Verfahren einfach billiger ist, als einen Entwickler ein paar Tage gescheit nachdenken zu lassen.

Da fällt es einem schon schwer, in die Bereiche des **Entwickelns** von Software zurückzukehren und sich nicht nur auf die reine Programmierung von Software zu beschränken. Aber es macht tatsächlich Sinn, ein Programm vorher gut zu planen. Ein oder gar mehrere Tage Ausprobieren kostet eben *doch* mehr Zeit, als wenn man vorher die eine oder andere Stunde in die gedankliche Planung investiert hätte. Auf diese Weise löst man durch statt einer kleinen Sackgasse im Ausprobierverfahren nämlich meistens ein grösseres Design-Problem der gesamten Software, so dass der kleine eben noch hilfreiche Kniff später nicht als gigantischer Boomerang wieder zurückkommt. Bei kommerzieller Softwareentwicklung schlägt das dann auch wieder finanziell zu Buche.

Lange Rede kurzer Sinn: Im folgenden werde ich nun das sogenannte **Entwicklungsskript** vorstellen. Dabei handelt es sich um ein Dokument, das man **vor** Beginn der Programmierarbeiten erstellen sollte. Ich verwende hier eine stark verkürzte Variante des gängigen fünf Phasen Modells der Softwareentwicklung und wir werden uns mit drei Phasen begnügen. Die ersten beiden Phasen, der Grobentwurf und der Feinentwurf, werden durch das Entwicklungsskript realisiert. Die dritte Phase ist dann die eigentliche Implementierung die zwar auch Teil des gesamten Skriptes ist, aber das Ausdrucken des fertigen Quelltextes können wir uns hier sicherlich sparen.

Das Entwicklungsskript

Ja ich wiederhole mich, aber man kann das Entwicklungsskript nicht oft genug erwähnen. Es enthält also die beiden Gliederungspunkte:

- Grobentwurf
- Feinentwurf

Welche Unterpunkte diese jeweils enthalten und wozu sie gut sind, das werde ich jetzt kurz erläutern und hoffen, daß sich möglichst viele Neulinge wenigstens mal mit der Idee hinter dem Konzept des Skriptes befassen. In der Industrie beginnt heute überall die Arbeit an einem Softwareprojekt immer mit dem Satz: „*Schreib mir erst mal ein Konzept, dann sehen wir weiter*!"

1. Grobentwurf

Den Grobentwurf kann man sich als eine Dokumentation der gewünschten Merkmale eines Spiels vorstellen und wie sie nach außen auf den Spieler wirken werden. Dazu zählen so triviale Sachen wie der Name des Spiels aber auch komplexere Abläufe wie das geplante Gameplay. Der Grobentwurf enthält also eine Art Beschreibung des Spiels, wie es der Spieler später einmal sehen wird. Diese Beschreibung dient einerseits dazu, sich selbst klar zu machen, was **genau** man eigentlich will. Andererseits bildet der Grobentwurf auch die Grundlage der Entscheidung YES or NO, entwickeln oder nicht.

1.1 GE: Der Arbeitsname des Spiels

Eine der ersten Sachen, die man im Kopf hat, ist der Name den das Spiel haben soll. Aber Vorsicht: Meistens wird man nach ein paar Tagen oder Wochen wieder eine bessere Idee haben, und dann noch eine und noch eine. Man sollte aber trotzdem einen festen Arbeitsnamen für das Spiel wählen und alle weiteren Ideen nebenbei sammeln. Wenn das Spiel erst einmal fertig ist, kann man dann den richtigen Namen auswählen. Das ist auf alle Fälle besser, als wenn man seinem Baby alle drei Tage einen neuen Namen verpasst und alle entsprechenden Dateien in der Benennung ändern muß.

Man sollte aber auch folgendes bedenken: Ein kurzer, prägnanter Name ist zwar eine feine Sache, es ist aber wissenschaftlich erwiesen, daß sich

Menschen komplexere Zusammenhänge besser merken können als einfache Bruchstücke von Informationen. Das heißt einfach nur, daß kryptische Abkürzungen wie *KKAD* schneller vergessen werden als sprechende Namen wie *Y – Beyond the Frontier*. Namensähnlichkeiten der beiden Beispiele zu eventuell sogar existierenden Programmen sind hier natürlich rein zufällig.

1.2 GE: Das Genre und die Systemanforderungen des Spiels

Das Genre, in dem man sein Spiel ansiedeln möchte, ist meist recht leicht festzulegen, denn wer würde sich hier nicht für das entscheiden, was ihm selbst am besten gefällt. Mögliche Genre sind *3D-First-Person-Shooter*, *Raumschiff-*, *Flugzeug-* oder *Fahrzeug-Simulationen*, *Sport-Spiele*, *2D-Strategie-Spiele* und viele andere. Natürlich ist auch ein guter Genre-Mix erlaubt und höchst abwechslungsreich, erfordert jedoch meistens wesentlich mehr Entwicklungsaufwand. Und ein gutes 2D Strategiespiel ist auch weiterhin ein gutes 2D Strategiespiel, auch wenn es an keiner Stelle 3D Grafik bietet. Aber dieser Punkt des Entwicklungsskriptes dürfte ja eigentlich keinerlei Probleme bereiten. Das Genre seines Projektes sollte man schließlich schon ganz gut vor Augen haben.

Aus dem gewählten Genre ergibt sich zudem noch die Mindestsystemanforderung, die man auch festhalten sollte. Entwickelt man das Spiel nur für die Elite unter den Prozessoren und Beschleunigerboards oder auch für *normalsterbliche* Computer? Sicherlich zählte jeder Teil der *Wing Commander* Saga oder auch jedes neue *Quake* immer wieder zu den optischen Highlights des technisch Machbaren, aber jedesmal zu Lasten der Systemanforderungen. Man sollte sich also vor Beginn seiner Arbeit darauf festlegen, welche Zielgruppe man anvisiert. Dazu gehört ebenfalls die Festlegung weiterer technologischer Anforderungen, wie beispielsweise die Verwendung einer bestimmten Mindestversion von DirectX, oder gegebenenfalls weiterer Programmierbibliotheken wie beispielsweise OpenGL.

1.3 GE: Der Ablauf des Spiels

Hier sollte man grob festlegen, was der Spieler alles machen oder sehen können soll beziehungsweise welche Handlungen der Spieler vornehmen kann. Ebenfalls von elementarer Wichtigkeit an dieser Stelle ist die *Definition der Regeln* und vor allem des *Ziels des Spiels*. Für eine „Space-Combat-And-Trade“ Simulation nach dem **Elite** Vorbild könnte das ungefähr so lauten:

Der Spieler beginnt das Spiel in einem Raumhangar aus einem 1-Person Blickwinkel auf sein Cockpit. Er kann sein Schiff durch Tastatur-, Maus- oder Joystickeingaben durch den Weltraum steuern und auf diversen Planeten oder Raumstationen landen.

Dort kann er Handel betreiben, Flugaufträge beziehungsweise Missionen annehmen und sein Raumschiff ausbauen oder reparieren lassen.

Während des Fluges wird der Spieler von Piratenschiffen angegriffen, denen er sich erwehren muß.

***Ziel des Spiels**: Profit erwirtschaften, ein bestimmtes Ansehen bzw. einen Titel erreichen!*

Wie man sieht, kann man selbst den Ablauf eines komplexen Spiels mit wenigen Kernsätzen zusammenfassen. An dieser Stelle interessiert es (*noch*) nicht, welche Handelswaren es alles gibt, wie viele verschiedene Schiffe, usw. Dieser Punkt ist quasi eine allgemeine Beschreibung des Spiels so wie man sie später als Kurzzusammenfassung in einem Spieletest Magazin lesen möchte.

1.4 GE: Das Storyboard des Spiels

Hm...jetzt wird es haarig. Eine gute Story kann den Spieler eng an das Spiel fesseln, auch wenn sie bei Computerspielen bei weitem nicht so wichtig ist, wie das Drehbuch eines Kinofilms. Es gibt zwar Spieler die sich weder Intros ansehen, noch Storylines verfolgen und einfach *drauflosspielen*, dennoch gibt es auch viele Spieler (*und Spieletester!!!*) die viel Wert auf eine gute Story legen.

Unter dem allgemeinen Begriff Storyboard würde ich zwei Elemente zusammenfassen. Zum einen die schriftliche Storyline, also die Geschichte des Spiels, zum anderen grafische Skizzen von geplanten Intros und Zwischensequenzen, ebenso wie von Hauptbestandteilen des Spiels. Letzteres wären beispielsweise Raumstationen oder Raumschiffe. Die **Abbildung 1** zeigt beispielsweise eine Skizze und das daraus resultierte 3D Modell eines Raumjägers von *WC Prophecy*.

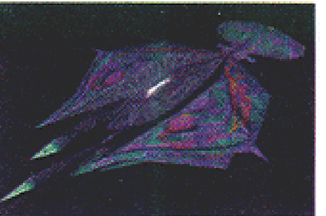


Abbildung 1: Origin's Wing Commander Skizze eines *Manta* Jägers
[aus PC Games 8/97]

Man sollte also wenigstens in groben Zügen bereits zu Beginn eine Storyline festlegen und diese auch durch ein Storyboard fixieren. Die Storyline unterteilt sich dann auch wieder in zwei Elemente. Jedes Spiels sollte eine Hintergrundstory haben, also eine Geschichte erzählen die in der Vergangenheit passierte und bis zum zeitlichen Beginn dieses Spiels reicht. Daneben kann man als zweites und wesentlich aufwendigeres Element auch noch eine begleitende Storyline einführen. Diese führt die Geschichte von Level zu Level des Spiels immer weiter. Die **Abbildung 2** zeigt als illustrierendes Beispiel die Storyline und den variablen Handlungsablauf von *WC Prophecy*.

Abbildung 2: Origin's Wing Commander Storyline
[aus PC Games 8/97]

Natürlich ist kaum ein Programmierer auch ein ausgebildeter und talentierter Grafiker. Trotzdem sollte man auf ein Storyboard nicht verzichten, auch Strichmännchen oder Strichmodelle sind okay...*so lange man diese Bilder nach Vollendung des Spiels verbrennt.* **;-)**

1.5 GE: Gameplay Design

So langsam nimmt das Spiel konkrete Formen an. Nun gilt es etwas detaillierter ins Gameplay einzusteigen. Bisher haben wir unter Punkt 1.3 eine recht allgemeine Beschreibung des zukünftigen Spiels, was uns aber fehlt, sind noch ein paar technische Details. Da wäre zum Beispiel die konkrete Steuerung des Spiels, also Eingaben durch Joystick und Tastatur, Reaktionen des Spiels auf Eingaben des Spielers und ähnliches. Daneben sollte man auch einen kleinen Überblick darüber erstellen, welche Optionsmenüs und Auswahlbildschirme das Spiel bieten soll. Hier ein paar Punkte, die auf jeden Fall unter diesem Punkt aufgeführt werden sollten.

- Welche Bildschirme gibt es neben dem Spielfeld? (*Startbildschirm, Auswahlmenüs, etc.*)

- Welche Menüs bzw. Auswahlbildschirme mit welchen Optionen gibt es und wie hängen diese ggf. zusammen? (*Auswahlmenü mit Schwierigkeitsstufe, Laden/Speichern usw.*). Hier hilft ein übersichtliches Diagramm meist besser weiter als tausend Worte!!!

- Welche Tastaturkommandos stehen dem Spieler zur Verfügung und ist der Input auch vom Spieler selbst konfigurierbar? (*Abbruch mit ESC, Schiessen mit Space oder Feuerknopf, Steuerung der Spielfigur durch den Joystick usw.*)

- Welche Art von künstlicher Intelligenz soll implementiert werden (*allwissende, superdumme, intelligent lernende oder durchschnittliche Gegner usw.*)?

1.6 GE: Level-/Umgebungs-Design

Neben dem Gameplay Design muß auch das Level Design definiert werden, bevor man mit der Arbeit beginnt. Dabei muß man eine Aufstellung aller geplanten Level, beziehungsweise Missionen entwerfen, die es eigentlich für jedes Spiel gibt. Diese sollte dann möglichst detailliert das beschreiben, was man unter 1.3 noch grosszügig übergangen hat. Für das dort aufgeführte Beispiel könnte das so aussehen:

- *Keine einzelnen Level vorhanden*

- *Raumstationen im Universum:*
RS 1: Agrarorientiert: Getreide, Landwirtsch. Maschinen, Medikamente
Weitere Bereiche: Transportauftragsvermittlung
RS 2: Industrieorientiert: Maschinen, Roboter, Werkzeuge, Waffen
Weitere Bereiche: Raumwerft, Ersatzteilshop
RS 3: ...usw.

Wenn man dagegen ein level- oder missionsbasiertes Spiel plant, dann muß man hier auf alle Fälle jeden einzelnen Level mit seinen wichtigsten Kernelementen beschreiben:

- Level 1: - Monster-Typen 1, 4 und 5
- Versteckte Waffen: Pistole, Magnetgewehr
- Versteckte Goodies: Medizinpacks, Zaubersprüche
- Levelziel: 5 Schalter an verschiedenen Stelle betätigen
- Folgelevel: 2
- Level 2: ...usw.

Auf jeden Fall sollte ein modernes Spiel heutzutage auch einen nicht-linearen Ablauf der Level oder Missionen unterstützen. Das bedeutet, daß der Folgelevel oder die Folgemission von dem Ergebnis der vorherigen Mission abhängt. Beispielsweise könnte eine Mission das Angreifen eines gegnerischen Raumjägergeschwaders beinhalten. Je nachdem, ob diese Mission erfolgreich war fliegt man danach entweder die Mission zum Angriff auf das gegnerische Trägerschiff, oder halt die Mission zur Verteidigung seines eigenen Trägerschiffes.

Im Falle eines solchen dynamischen Spielverlaufs benötigt man logischerweise auch hier eine Art Storyline, sozusagen eine *Level*- oder *Missionline*, in der alle möglichen Abläufe dargestellt werden. Dabei ist man natürlich nicht auf zwei potentielle Folgemissionen beschränkt. Ebenso wenig muß jede Mission dieselbe Anzahl an möglichen Nachfolgern bieten.

Unfairness sollte man aber vermeiden, denn nichts ärgert den Spieler mehr, als wenn er nach einem Verlauf von zwei, drei schlechten Mission feststellt, daß er das Spiel überhaupt nicht mehr gewinnen kann, selbst wenn ab jetzt alles glattläuft. Ebenso ist es superärgerlich, wenn der Spieler irgendwo in einen Fluß oder ein Loch fällt, aus dem es keinen Ausweg mehr gibt und man das Spiel neu starten muß. Denn dann hat der Leveldesigner kläglich versagt...

1.7 GE: Last Chance Check

Wenn man nun sein Entwicklungskript so weit fertig hat und es wenigstens ein paar Seitchen umfaßt, so sollte man es einem letzten Test unterziehen. Die Testfragen, die man möglichst objektiv beantworten sollte, lauten dabei:

- *Wird das Spiel Spaß machen?*

- *Wird das Spiel auch anderen Leuten Spaß machen?*

Diese Fragen erscheinen trivial, sind es aber nicht. Der Computerspielemarkt ist heute überschwemmt von Nachahmungen berühmter Titel, tonnenweise Shareware Games und anderen üblen Dingen wie zum Beispiel dem *Jagd Simulator*. Erst wenn man die beiden obigen Fragen tatsächlich mit „JA!“, und nicht nur mit „ja“, beantworten kann, sollte man mit der eigentlichen Arbeit beginnen. Für die ersten Spiele, die man entwickelt, kann man in der Interpretation der Antworten auf die obigen beiden Fragen natürlich etwas großzügiger sein.

2. Feinentwurf

Der Grobentwurf stellte ja quasi eine Repräsentation des Spiels nach außen hin dar. Logischerweise gehen wir jetzt nach innen. Der Feinentwurf ist dazu gedacht, dem Programmierer eine Art Programmieranleitung zu liefern. Er enthält beispielsweise eine Auflistung wichtiger Variablen, die man auf alle Fälle benötigen wird.

Hier beginnt im Prinzip schon fast die eigentliche Programmierung, nur halt ohne Compiler, sondern auf dem Papier. Was aber hier auf alle Fälle beginnt, ist die Überlegung über die Strukturierung der Software, also zum Beispiel die Gestaltung der verwendeten Datenstrukturen. Die gedankliche Arbeit, die man hier investiert, spart man bei der tatsächlichen Implementierung dann wieder ein.

2.1 FE: Unterteilung des Programms in Dateien

Nun hat die Idee also den *Last Chance Check* bestanden und es kann mit der Umsetzung begonnen werden. Bevor man aber weiter ins Detail geht, sollte man sich überlegen, wie viele Quellcode Dateien man verwenden will und wie man seinen Code am logischsten über mehrere Dateien verteilen kann. Es gibt kaum etwas unübersichtlicheres als eine Software, deren gesamter Inhalt in einer einzelnen Datei steckt. Okay, die meisten Programme verwenden zwar einen Satz an Bibliotheken, beispielsweise um DirectX zu initialisieren und zu verwenden, aber der gesamte spielbezogene Quelltext von der Bewegung des Spielers über die künstliche Intelligenz der Gegner bis hin zur Kollisionsabfrage steht in einer Datei. Selbst bei den simpelsten Spielen kommt man so locker auf mehrere tausend Zeilen Quelltext. Und wenn man seinen Code dann auch noch gut kommentiert, paßt das sowieso nicht mehr auf eine Kuhhaut. Sinnvollerweise unterteilt man sein Projekt dann in mehrere Häppchen, in denen jeweils logisch zusammengehörende Funktionen in einer Datei stecken. Zur Verdeutlichung ein kleines Beispiel:

- `Global.h`
Diese Header Datei beinhaltet alles Globales, was im Prinzip jeder weiteren Datei zur Verfügung stehen muß. Namentlich sind dies Definitionen, Makros und Datenstrukturdefinitionen.

- `Main.cpp & Main.h`
Wie der Name es schon nahelegt beinhalten diese beiden Dateien alles, was rund um den Kern des Spiels, insbesondere die WinMain() Funktion mit der Hauptschleife, sowie die wichtigsten globalen Variablen. Daneben werden hier natürlich in der Hauptschleife die Eingaben des Spielers abgefragt und dann weitere Funktionen aus den anderen Dateien aufgerufen, um auf diese Eingaben zu reagieren.

- `Sprite.cpp & Sprite.h`
In diesen Dateien stecken dann alle Makros, Variablen und Funktionen, die zur Verwaltung eines Sprite Datentyps notwendig sind. Also insbesondere das Laden und Anzeigen von Grafiken.

- `Objekt.cpp & Objekt.h`
Und in diesen beiden Dateien steckt schließlich alles, was mit den Objekten des Spiels zu tun hat. Das ist beispielsweise die künstliche Intelligenz zum Bewegen der Objekte, Kollisionsabfragen, usw.

Das soll hier nur ein kleiner Überblick sein, damit man sieht, wovon ich gerade spreche. Später werden wir dann genauer sehen, wie man so ein Spiel unterteilen kann und sollte. Diese Unterteilung ist durchaus sinnvoll, denn es gibt nichts unübersichtlichereres, als eine Quelltextdatei die Tausende von Zeilen lang ist. Auch das Argument, daß ein Spiel dann aus weniger Dateien besteht, zählt hier nicht. So eine Datei mit Monsterlänge verdrängt nicht nur die Übersichtlichkeit von der Bildfläche, sie terminiert auch die Motivation, sie zu lesen. Das menschliche Gehirn wird schlicht und einfach abgeschreckt, sich durch diesen Berg an Informationen zu wühlen, wenn man kein Ende absehen kann.

Eine sorgsame Planung der einzelnen Bestandteile des Spiels in eine jeweils passende Datei sorgt dann auch dafür, dass man sich selbst die Arbeit etwas einfacher macht. Wenn man sich in VC++ ein Projekt angelegt und die entsprechenden Datei hinzugefügt hat so kann man die einzelnen logischen Happen des Projektes gut voneinander separieren und getrennt fertig stellen. Das ist ein zusätzlicher psychologischer Motivationsfaktor.

2.2 FE: Datenstrukturen des Spiels

Als nächstes sollte man dem Entwicklungsskript jetzt eine Liste der geplanten Datenstrukturen hinzufügen. Damit kommen wir zum ersten echten Knackpunkt des Entwicklungsskriptes denn ja(!), man muss wirklich an dieser Stelle bereits im Kopf überlegen, welche Datenstrukturen man für das gesamte Spiel brauchen wird. Es macht wenig Sinn, erst einmal *los zu programmieren* und immer erst dann, wenn man im Programm merkt dass man eine Datenstruktur braucht, diese implementiert. Das resultiert meistens in nicht ausgereiften Strukturen die man mindestens dreihundert mal umändern muss, damit sie den ständigen Änderungen im Programm gerecht werden...was wiederum selbst Änderungen auslösen würde...usw.
Also erstellt man besser bereits in der Planungsphase (*dazu ist sie ja da*) eine Liste für alle eigenen Datenstrukturen in bereits korrekter C/C++ Syntax, so wie sie später im Quelltext erscheinen. Dazu erhält jede Struktur eine kurze Erläuterung aller Felder, die sie enthält und wozu die Struktur benutzt werden soll. Und nun das gute daran: Um so eine Liste zu erstellen muss gescheit darüber nachdenken, was für Strukturen man alles benötigt. Und das muss man bis zum bitteren Ende durchdenken.

Mögliche Kandidaten für Datenstrukturen sind in erster Linie Datenstrukturen für die Spielfigur (*Position, Geschwindigkeit, Ausrichtung, Lebensenergie, Zustand usw.*) und für die weiteren statischen oder dynamischen Objekte des Spiels, aber auch Strukturen zum Speichern von Bitmap Grafiken oder 3D Modellen. Hier ein kleines Beispiel:

```
typedef struct SPIELER_TYP {
   int x,y,z;       // Position
   int vx, vy, vz; // Geschwindigkeit
   int Zustand;   // Angriff, Flucht, Suche, Tod
  int Energie;   // Lebensenergie
```

```
} SPIELER;
```

Die Datenstrukturen eines Computerprogramms, insbesondere eines Spiels, zählen zu den wichtigsten Dingen überhaupt. Sie stellen quasi die Infrastruktur dar auf der das gesamte Programm basiert. Bevor man eine Funktion schreiben kann, die auf die Datenstrukturen zugreift muss man schliesslich die Datenstrukturen endgültig definiert haben. Stellt man im Laufe der weiteren Entwicklung dann aber fest, dass eine Struktur geändert werden muss, so zieht dass ja einen ganzen Rattenschwanz an Änderungen in anderen Programmteilen nach sich. Auch wenn das kräftige Nachdenken mit Stift und Papier (*oder MS Word*) vor dem eigentlichen Programmieren sicherlich etwas gewöhnungsbedürftig ist, so ist es doch unerlässlich und wird später mehr Zeit und Frust einsparen als es anfangs erzeugt.

2.3 FE: Datenstrukturen des Spiels

Ebenso wie bei den Datenstrukturen geht man bei den globalen Variablen vor, bei denen man sowohl ihren Datentyp bzw. verwendete Datenstruktur und Bezeichnung als auch eine kurze Erläuterung über ihren Verwendungszweck angibt. Und wieder ein kleines Beispiel:

```
SPIELER Spielfigur;  // globale Variable für die Spielfigur
```

2.4 FE: Wichtige Funktionsprototypen des Spiels

Der letzte und wohl schwierigste Schritt ist dann die Auflistung aller wichtigen Funktionsprototypen. Das „*wichtig*“ heißt dabei nicht, daß man Funktionen wegläßt, die man für unwichtig hält. Es deutet vielmehr an, daß man meistens gar nicht weiß, welche Funktionen eventuell noch im Verlauf der eigentlichen Programmierung dazu kommen und die Bezeichnung w*ichtige Funktionen* klingt halt besser als die Bezeichnung *Alle Funktionen die mir bis hier eingefallen sind*...

Neben den reinen Funktionsprototypen, also dem Rückgabetyp, dem Namen und der Parameterliste der Funktion sollte man hier auch schon die Funktionsrümpfe für besondere Kernfunktionen in Pseudocode angeben. Solche Funktionen zeichnen sich dadurch aus, daß sie nicht ganz alltägliche Aufgaben übernehmen. Beispielsweise reicht es aus, für eine Funktion die nur DirectX initialisiert, den Prototypen anzugeben:

```
BOOL Initialisiere_DirectX(void);
```

In dieser Funktion wird nichts passieren, was irgendwie etwas Besonderes wäre. Jedes Programm, das DirectX verwendet wird eine Funktion haben, die fast identisch ist. Wenn man aber auf etwas interessantere Funktionen stößt, wie zum Beispiel eine Kollisionsabfrage zwischen zwei Objekten, so sollte man im Pseudocode angeben, wie man diese Kollisionsabfrage später implementieren will.

```
BOOL Kollision(OBJEKT obj1, OBJEKT obj2) {
   Berechne den Radius des Umfangs von obj1;
   Berechne den Radius des Umfangs von obj2;
   Berechne die Entfernung der beiden Objekte zueinander;

   WENN Entfernung kleiner als Radius obj1 + Radius obj2 DANN
      return TRUE;
   SONST
   return FALSE;
   }
```

Natürlich gehört zu jedem Funktionsprototypen auch eine entsprechende Erklärung, was diese Funktion leisten soll, wozu die einzelnen Parameter benötigt werden, und welche Rückgabewerte sie bei welchen Ergebnissen liefert. Bei einigen Funktionen kann man solche Erklärungen mit ein, zwei Sätzen sehr knapp halten, bei anderen Funktionen muß man schon etwas ausführlicher erklären.

2.5 FE: Zeitplan und Meilensteine des Spiels

Hatte ich eben gesagt, daß insbesondere die Auflistung aller Funktionsprototypen Schwierigkeiten bereitet? Dann wird es bei diesem Punkt erst recht kritisch. Einen Zeitplan, jetzt ganz modern auch *Projektplan* genannt, zu erstellen ist für Laien beinahe unmöglich. Vor allem wenn man an einem Spiel nur nebenbei als Hobbyprogrammierer arbeitet. Zum einen läßt sich der benötigte Zeitaufwand von Anfängern kaum abschätzen, zum anderen hat man seine zur Verfügung stehende Zeit als reiner Hobbyprogrammierer natürlich nicht so statisch eingeteilt wie ein Berufsprogrammierer und man weiß nie so genau, wann man Zeit findet an seinem Projekt zu arbeiten.

Als Faustregel gilt, daß man sinnvolle Zeitpläne nur aufgrund eigener Erfahrung erstellen kann. Daher sollte man auch bereits bei seinem ersten Projekt damit anfangen. Es stört ja niemanden, wenn man sein Zeitbudget um 500 % überzieht. Aber wie soll man jemals den Aufwand vernünftig abschätzen können, wenn man nie anfängt Erfahrungen zu sammeln?

Ein weiterer wichtiger Begriff, den ich noch schnell erwähnen möchte, sind die Meilensteine oder modern auch Milestones genannt. Auch dieses Wort stammt aus dem zur Zeit sehr hippen Projektmanagement und bedeutet einfach eine Art Checkpunkt. In seinem Entwicklungsskript sollte man auf alle Fälle diese Meilensteine definieren, das löst gewissermaßen auch das Zeitplanproblem ein wenig. Diese Meilensteine stellen gewisse Punkte im Entwicklungsablauf dar, die sehr markant und von elementarer Bedeutung für die Fortführung der Arbeit sind. Für ein Spiel könnte das so aussehen wobei die Termine Arbeitstage gerechnet ab der eigentlichen Implementierungsphase darstellen:

	Meilenstein	**Termin**
1. Meilenstein	Initialisierung einer Fullscreen Anzeige mit DirectInput, Direct3D und DirectSound	Tag 3
2. Meilenstein	Einfaches Prototyp (*unoptimiertes*) Laden von 3D Modellen	Tag 4
3. Meilenstein	Kollisionsabfragen und Bewegung des Spielers in der Spielwelt möglich	Tag 12
4. Meilenstein	Dynamischer Objekte wie Waffenfeuer, Explosionen usw.	Tag 25
5. Meilenstein	Implementierung der künstlichen Intelligenz.	Tag 45
6. Meilenstein	Einbinden der Zwischensequenzen, Start- und Abschlußbilder	Tag 52
usw.		

Diese Einteilung ist einem dabei behilflich, den Fortschritt eines Projektes zu beurteilen und auch in jeder Phase der Entwicklung ein konkretes Zwischenziel definiert zu haben. Denn selbst wenn man erst mal die Funktionsprototypen alle zusammen hat, dann wird man vor einem Haufen von Arbeit sitzen und sich fragen, womit man eigentlich beginnen soll. Aber dazu kommen wir gleich noch einmal ausführlich. Damit man hier nicht die Übersicht verliert, verwendet man diese Meilensteine um sich selbst zielgerichtet und geplant auf den erfolgreichen Projektabschluß zuzusteuern.

Fazit

Sinn das ganzes ist folgendes: Wenn man das Entwicklungsskript einem Programmierer in die Hand drückt, der von dem konkreten Projekt keine Ahnung hat, so soll er auf Anhieb ohne weitere Erklärungen das Projekt selbst implementieren können.

Wie bereits angedeutet, macht der gesamte Feinentwurf besondere Schwierigkeiten. Selbst bei genauster Planung eines Spiels wird es immer wieder vorkommen, daß man im Laufe der Programmierung Mängel feststellt. Dann muß man entweder Variablen, Strukturen oder Funktionen ändern oder meistens auch neue hinzufügen. Trotzdem macht der Feinentwurf Sinn, denn es ist im Prinzip eine Programmieranleitung für sich selbst. Wenn man erst im Laufe der Programmierung darüber nachdenkt, was in welche Datei kommen soll, was für Funktionen man als nächstes schreibt usw. dann macht man sich unnötige Arbeit. Ebenso resultiert daraus meistens ein recht unorganisiertes Programm.

Das Entwicklungsskript bereitet einem eine Menge Arbeit, gerade im zweiten Teil. Dies ist aber Arbeit, die man dann zu 200 % bei der eigentlichen Programmierung spart und deshalb lohnt es sich auf alle Fälle. Auch Anfänger sollten sich dazu zwingen und erstaunlicherweise ist die Bereitschaft der Anfänger, ein Entwicklungsskript zu erstellen, meistens höher als die der erfahrenen Programmierer, die zu sehr in einen *tipp-drauflos-Trott* gekommen sind.

Angemerkt hatte ich ja schon, daß meine Variante des Entwicklungsskriptes schon eine stark verkürzte Version ist. Es gibt ganze Bücher die hunderte von Seiten lang beschreiben, welche Dokumente zur Entwicklung und Planung einer Software VOR der eigentlichen Programmierung anzufertigen sind. Ebenso gibt es auch genaue Vorschriften darüber, was diese Dokumente enthalten sollen. Die Wichtigkeit der gewissenhaften Durchführung der Erstellung eines Entwicklungsskriptes ist nicht nur damit begründet, daß man strukturierter und geplanter arbeiten kann. Bei kommerziellen Softwareprodukten kommt noch hinzu, daß die jeweiligen Berichtshefte der einzelnen Schritte als Vertragswerk zwischen dem Auftraggeber und den Programmierern dienen.

Der Auftraggeber kann so bereits frühzeitig die geplante Umsetzung mit seinen Wünschen abgleichen und eventuelle Änderungen fordern. Die Programmierer dagegen haben eine sehr genaue Anleitung, welche Funktionalitäten zu implementieren sind.

Geschafft - und was nun?

Wenn man also das Entwicklungsskript erfolgreich fertig gestellt hat, dann kann man mit der Programmierung des Spiels beginnen. Das ist jetzt eigentlich nur noch reine Routinearbeit, denn man kann nun das Entwicklungsskript beziehungsweise den Feinentwurf daraus als Programmieranleitung verwenden.

Aufgrund einiger Zuschriften habe ich aber festgestellt, dass viele Leute zwar gute Ideen haben was sie programmieren wollen und auch schon eine kleine 2D oder 3D Engine zum Anzeigen von Grafiken bzw. Modellen entworfen haben, dann aber vor dem Problem stehen dass sie nicht wissen, was sie als nächstes tun sollen. Darum hier ein kleiner Leitfaden wie man beim Programmieren sinnvoll vorgehen kann (*aber nicht unbedingt muss*):

Erzeugen einer Rahmenanwendung

Zuerst programmiert man eine Anwendung, die unter Windows einen Fullscreen erzeugt und einen Front- und Backbuffer benutzt. Dabei lässt man einfach einen leeren Bildschirm anzeigen, das genügt völlig. Falls man DirectX benutzt (*ohne würde ich das nicht versuchen wollen*) beinhaltet dieses Rahmenprogramm dann auch die notwendigen Initialisierungen für DirectInput, so dass das Programm dann auch auf DirectInput Tastatur-, Maus- und Joystickeingaben reagieren kann.

Erstellen der Rendering Pipeline

Ob nun 2D oder 3D, als nächstes benötigt man auf alle Fälle eine Pipeline zum Anzeigen der Grafik. Das beinhaltet sowohl die Funktionalität zum Laden von Bitmap Grafiken als auch das Anzeigen der Grafik am Bildschirm. Bei 2D bedeutet dies zum Beispiel Funktionen für den DirectGraphics Blit eigener Datenstrukturen. Bei 3D Programmen bedeutet dies zusätzlich ein System zum Laden und Anzeigen von 3D Modellen.

Aufbauen eines Testuniversums

Bisher beschränkte man sich ja nur auf das testweise Anzeigen von Grafik auf dem Bildschirm. Bevor man jetzt etwas Sinnvolles zufügen kann, bedarf es einer besseren Testumgebung. Für 2D bedeutet dies, dass man ein Spielfeld mit diversen statischen Objekten erzeugt. In 3D wird man dann ein einfaches kleines Universum entwerfen das einige statische Objekte enthält an denen man sich orientieren kann.

Erkunden des Testuniversums

Jetzt kommt der Input ins Spiel denn nun ist es an der Zeit, dem Programm etwas Leben einzuhauchen. Nun kommt also die Funktionalität zur Bewegung der Spielfigur (*noch nicht der anderen Objekte!!!*) hinzu. In 2D sollte man also die Figur des Spielers auf dem Spielfeld per Tastatur, Maus oder Joystick bewegen können. Bei 3D sollte der Spieler sich selbst durch entsprechende Eingaben durch das Testuniversum bewegen können. Letzteres schliesst auch Rotationen um die eigenen Achsen mit ein.

Entwicklen eines Kollisionssystems

Während seiner virtuellen Erkundung des Testuniversums wird man schnell feststellen, dass man sich geisterhaft durch alle Testobjekte hindurch bewegen kann. Nun sollte man das Programm also um Kollisionsabfragen bereichern, die das verhindern. Wann immer der Spieler mit einem 2D oder 3D Testobjekt kollidiert, wird seine Geschwindigkeit auf 0 gesetzt. Jetzt hat man einen etwas realeren Eindruck von der Testumgebung.

Bewegung des Testuniversums

Jetzt kann der eigentliche Spass beginnen. Nun wird man Funktionen zur künstlichen Intelligenz entwickeln, damit sich die bisher statischen Testobjekte des Testuniversums auch bewegen können. Je nach Art des Objektes wird das ziemlich aufwendig. Wenn man aber klein anfängt und die KI dann schrittweise erweitert wird man schnell Erfolgserlebnisse haben.
Aus eigener Erfahrung kann ich sagen, dass es schon ein erhebendes Gefühl ist, wenn man beim Erkunden seiner Testumgebung feststellt, dass die Objekte sich zum ersten Mal ohne Eingaben selbst bewegen.

Ergänzen der Action

Nun fehlt es dem Programm nicht mehr an viel, um als vollwertiges Spiel durchzugehen. Jetzt ergänzt man die Action die man haben möchte. Das wären beispielsweise eigener und gegnerischer Waffeneinsatz. Physikalische Gesetzmäßigkeiten wie beispielsweise die Schwerkraft und, und, und...

Wie gesagt, das ist eine kleine Hilfestellung wie man vorgehen kann. Dem einen oder anderem wird das so vielleicht nicht gefallen und er wird einen anderen Weg wählen. Wer aber eine kleine Hilfe nicht von der Bettkante stösst (*es sei denn in Richtung Bett*), der kann diese Auflistung als kleinen Leitfaden für seine Vorgehensweise verwenden.

Uah...genug gelabert, jetzt wird es an der Zeit auch mal etwas Code anzufassen, oder?!

Weiter geht's zum Kapitel 2..

SPIELEENTWICKLUNG MIT DIRECT3D IM - KAPITEL 2

von Stefan Zerbst

"The bad guys always have the better technology."
André LaMothe, Tricks of the Windows Game Programming Gurus

Eine einfache Windows Anwendung

Okay, here we go. Nach der Einleitung wissen wir ja bereits, worum es sich bei der Spieleentwicklung handelt und was wir so alles zu beachten haben. Die Frage stellt sich aber nun: wie beginnen wir mit der Spiele*programmierung*? Die einfache Antwort lautet natürlich, dass wir zunächst einmal ein einfaches Windows Programm zum Laufen bekommen müssen. Darum werden wir uns in diesem Kapitel kümmern. Wir benötigen lediglich ein Rahmenprogramm welches uns ein einfaches Fenster auf dem Desktop öffnet, welches weder eine Menüleiste noch einen Inhalt hat, sondern einfach blank weiss ist.

Ein einfaches Windows Programm ist natürlich nicht weiter schwer herzustellen, daher werden wir gleich noch das eine oder andere Gimmick in den Code einbauen welches für die Spieleprogrammierung von Bedeutung ist. Alles in allem benötigen wir lediglich drei, vier Funktiönchen um unser kleines Fensterchen das Licht der Welt erblicken zu lassen. Natürlicherweise beginnen wir mit der `WinMain()` Funktion. Die Funktion in unserem Programm die diesen Namen trägt wird bei der Programmausführung automatisch aufgerufen. Daher packen wir die Erzeugung unseres Fensters ebenso wie die Hauptschleife genau in diese Funktion, eben damit unser Programm beim Start ein Fenster erzeugt und dann die Hauptschleife startet.

Der Kopf der Funktion muss wie folgt aussehen:

```
int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hprevinst,
                   LPSTR lpcmdline, int ncmdshow)
```

Wir quälen uns jetzt nicht weiter damit herum, was die einzelnen Parameter dieser Funktion genau bedeuten. Für unsere Zwecke ist das auch vollkommen uninteressant denn wir benötigen diese Parameter einfach nicht! Beim Start unseres Programms ruft das Betriebssystem die Funktion `WinMain()` auf und füllt die Parameter mit den notwendigen Werten. Aber auch das kann uns egal sein. Fragen wir uns also lieber, was uns nun interessiert. Und damit sind wir auch schon mitten drin in den Fenstern, denn nun wollen wir ein einfaches, leeres Fenster erzeugen.

Erzeugung eines Fensters

Dazu sind drei Schritte notwendig: Wir müssen (a) ein Fensterobjekt erzeugen und diesem die gewünschten Eigenschaften unseres Fensters angeben. Dann müssen wir (b) unser Fensterobjekt bei Windows registrieren lassen, denn es führt schliesslich kein Weg am Betriebssystem vorbei. Zu guter letzt müssen wir (c) ein echtes Fenster aus unserem registrierten Fensterobjekt erzeugen. Das ist alles.

Beginnen wir bei Schritt (a):
Unser Fensterobjekt ist unter Windows ein Objekt der Datenstruktur mit der Bezeichnung `WNDCLASSEX`. Diese Struktur enthält viele Unterelemente durch die wir das gewünschte Aussehen des Fensters beschreiben können.

```
WNDCLASSEX winclass;                      // Objekt vom Typ Fenster
const char chKlassenname[] = "Unwichtig"; // Name für das Objekt

// Initialisiere die Eigenschaften des Fensters
winclass.cbSize         = sizeof(WNDCLASSEX);
// Bei horizontal und vertikal Verschiebung Neuzeichnen
winclass.style          = CS_HREDRAW | CS_VREDRAW;
// Name der Callback-Funktion (s.u.)
winclass.lpfnWndProc    = WindowProc;

winclass.cbClsExtra     = 0;      // Dies und das...
```

```
winclass.cbWndExtra    = 0;     // Dies und das...
winclass.hInstance     = hinst; // Dies und das...
winclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);     // Standardicon
winclass.hCursor       = LoadCursor(NULL, IDC_ARROW);         // Standardcursor
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); // Weisser Hintergrund
winclass.lpszMenuName  = NULL;          // Kein Menü
winclass.lpszClassName = chKlassenname; // Name unserer Klasse
winclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION); // Icon bei verkleinertem Fenster
```

Nun gut, das ist glaube ich einfach genug. Vielleicht ist hier nicht alles im Detail erklärt, aber die Kommentare sollten doch ausreichen, um uns einen Eindruck davon zu vermitteln was dort passiert, beziehungsweise was dort passieren soll. Das einzig wirklich spannende ist der Name der Callback Funktion `WindowProc`, denn dazu kommen wir weiter unten noch einmal. Merken wir uns das also im Hinterkopf und ziehen gleich weiter. Nicht aber ohne zu übersehen, dass wir den Namen unseres Objektes mit der Variablen `chKlassenname` angegeben haben. Jetzt haben wir also eine Art Schablone für ein Fensterobjekt erstellte mit deren Hilfe wir das Aussehen unseres Objektes beschrieben haben.

Damit sind wir bei Schritt (b):
Jetzt müssen wir unsere neue Schablone bei dem Betriebssystem Windows registrieren lassen.

```
if (!RegisterClassEx(&winclass))
   return 0;
```

Wir rufen ganz einfach die Funktion `RegisterClassEx()` auf, mit der wir unsere Schablone (*die bei Windows als Windows-Class bezeichnet wird*) beim Betriebssystem registrieren können. Welche Sinn das hat werden wir im Schritt (c) noch sehen. Der Funktion geben wir als einzigen Parameter das eben erstellte Objekt unserer Schablone an und erwarten den Rückgabewert `TRUE`. Anderenfalls brechen wir die `WinMain()` Funktion sofort durch die Rückgabe von 0 beleidigt ab.

Yeah, nun kommt schon Schritt (c):
Im letzten Schritt zur Erzeugung und Anzeige eines Fensters müssen wir ein tatsächliches Fenster unter Windows erzeugen. Und das geht wie folgt:

```
HWND hwnd;

if (!(hwnd = CreateWindowEx(NULL,
                 chKlassenname,
                 "3D Spieleprogrammierung",
                 WS_POPUPWINDOW | WS_VISIBLE,
                 10,10,
                 400,300,
                 NULL,
                 NULL,
                 hinst,
                 NULL)))
   return 0;
```

Mittels der Funktion `CreateWindowEx()` erzeugen wir unter Windows ein Fenster. Der erste Parameter würde verschiedene Features des Fensters aufnehmen, für unsere Zwecke geben wir aber keinen Wert an. Und nun zum zweiten Parameter. Für diesen geben wir die Variable `chKlassenname` an, die wir bereits für die Benennung unserer Schablone verwendet habe ...*aha!*... nun weiss Windows also dass es die entsprechende Schablone für das Fenster zu verwenden hat. Windows sucht also in allen seinen registrierten ...*aha!*... Schablonen nach der passenden und erzeugt daraus ein tatsächliches Fenster. Der dritte Parameter gibt den Namen an, der in der Titelleiste des Fensters stehen soll (*die wir nicht haben*) ebenso wie in der Taskleiste.
Der folgende Bezeichner `WS_POPUPWINDOW` spezifiziert das Aussehen des Fensters (*ohne Titelleite!*) während der "dazu-geoderte" Bezeichner `WS_VISIBLE` dafür sorgt, dass das Fenster sofort am Bildschirm sichtbar ist. Nachfolgende stehen dann die Koordinaten der oberen linken Ecke des Fensters sowie dessen Grösse. Die nun noch verbleibenden Parameter der Funktion (*Vaterfenster, Menüobjekt*) sind für uns auch nicht weiter von Bedeutung, wir geben hier lediglich `NULL` an beziehungsweise die von Windows beim Aufruf der `WinMain()` Funktion erzeugte Variable `hinst`.

Interessant und überaus wichtig ist nun noch der Rückgabewert der Funktion vom Typ `HWND`. Diese kryptische Abkürzung bedeutet nix anderes als "***H**andle for a **Wind**ow*" und ist eine Art Nummer anhand derer wir unser tatsächliches Fenster (*und nicht die Schablone*) eindeutig von allen anderen aktiven Fenstern unterscheiden können. Daher speichern wir diese Nummer auf alle Fälle in einer globalen Variablen, wir wissen ja nie wann wir sie später noch einmal brauchen werden. Auch hier brechen wir unsere Funktion ab, wenn wir nicht die gewünschte Erfolgsmeldung `TRUE` als Rückgabewert

erhalten.

Nachrichten und die Hauptschleife

Nun ist es bereits an der Zeit dafür, unserem Programm eine Hauptschleife zu spendieren. Diese steht logischerweise direkt nach der Erzeugung des Fensters in der `WinMain()` Funktion. Wir verwenden dazu eine einfache `while` Schleife deren Abbruchbedingung eine globale Variable ist. So lange wie diese Variable ihren Startwert behält wird auch die Schleife laufen. Wollen wir das Programm beenden, so müssen wir die Hauptschleife verlassen und dazu setzen wir dann wenn es so weit ist die Abbruchvariable der Schleife auf den entsprechenden Wert.

```
#define SPIEL_AUSWAHL  0
#define SPIEL_START    1
#define SPIEL_LAEUFT   2
#define SPIEL_NEUSTART 3

BOOL blnBeenden = FALSE;
MSG  message;
int  Spiel_Zustand = SPIEL_AUSWAHL;

// H A U P T S C H L E I F E //////////////////////
while (!blnBeenden) {
   // Ist eine Nachricht zu verarbeiten?
   while(GetMessage(&message,NULL,0,0)) {
      TranslateMessage(&message);
      DispatchMessage(&message);
      } // GetMessage

   // Startzeit der Hauptschleife
   dwStartzeit = GetTickCount();

   switch(Spiel_Zustand) {
      case SPIEL_AUSWAHL:
         {
         // [...]
         Spiel_Zustand = SPIEL_START;
         } break;
      case SPIEL_START:
         {
         // [...]
         Spiel_Zustand = SPIEL_LAEUFT;
         } break;
      case SPIEL_LAEUFT:
         {
         // [...]
         } break;
      case SPIEL_NEUSTART:
         {
         // [...]
         Spiel_Zustand = SPIEL_START;
         } break;
      default: break;
      } // switch

   // Framerate bremsen
   while ((GetTickCount()  - dwStartzeit)  <  40) /*tu nix*/;
   } // Hauptschleife
```

Jaja, keine Panik...ich erkläre ja schon **;-)**. Das Betriebssystem Windows ist auf der Versendung von Nachrichten aufgebaut. Das bedeutet wann immer sich etwas bewegt so wird von Windows eine Nachricht vom Typ `MSG` (***m**e**s**sa**g**e*) erzeugt und an das Programm geschickt zu dem sie gehört...damit dieses sich dann gefälligst darum kümmert. Solche Nachrichten werden also immer dann erzeugt, wenn eine Taste gedrückt, die Maus bewegt, auf etwas geklickt wird usw. Dann erzeugt Windows die entsprechende Nachricht und stellt sie in eine Warteschlange des Programms, welches gerade

aktiv ist.
Damit diese Warteschlange nicht unendlich lang wird, und das Programm natürlich auch auf Eingaben reagieren muss, müssen wir auch brav dafür sorgen, dass sich unser Programm diese Nachrichten aus seiner Warteschlange holt. Wir tun dies durch die Funktion `GetMessage()`, die in einer eigenen Schleife läuft bis wir alle neuen Nachrichten eines Frames abgeholt haben. Diese Nachrichten speichern wir in der Variablen `message` und verarbeiten sie weiter. Die Funktion `TranslateMessage()` sorgt dafür, dass die von Windows codierte Nachricht in ein für uns lesbares Format umgewandelt wird. Durch den Aufruf der Funktion `DispatchMessage()` schicken wir diese Nachricht dann an die *Callback* Funktion unseres Fensters.

Okay...was zur Hölle sind Callback Funktionen? Nun wir hatten oben in unserer Schablone bereits angegeben, dass wir eine Callback Funktion namens `WindowProc()` verwenden wollen, an die alle Nachrichten gesendet werden sollen. Und genau das heisst Callback Funktion: Windows *ruft* diese Funktion automatisch *zurück* und gibt ihr unsere `message` Variable zur Verarbeitung. Es ist also unser Job eine geeignete Callback Funktion namens `WindowProc()` zu schreiben in der wir auf die Eingaben des Benutzers oder Nachrichten des Systems reagieren können. Doch das verschieben wir noch einmal auf später und sehen uns den gesamten Nachrichtenverkehr in Windows sicherheitshalber noch einmal grafisch an.

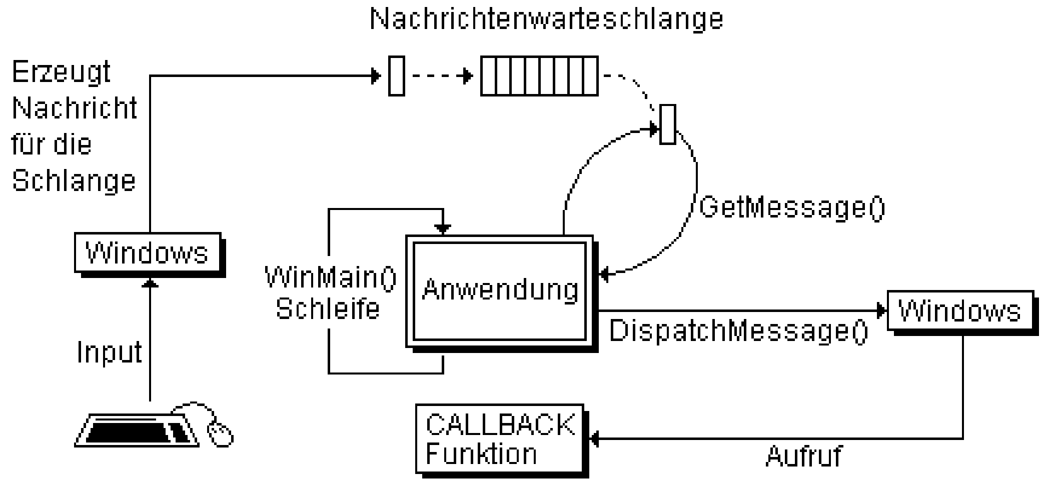


Abbildung 1: Nachrichtenverkehr unter Windows

Der Rest der Hauptschleife ist eine gigantische `switch` Anweisung. Hierbei unterteilen wir unser Spiel in verschiedene Zustände, wie beispielsweise Start des Spiels wo wir eventuell ein Startbild anzeigen werden oder einen Zustand in der das Spiel gerade läuft. Dazu führen wir die jeweils notwendigen Aktionen in den verschiedenen Zuständen aus, dazu zählt gegebenenfalls auch der Wechsel in einen andere Zustand. Beispielsweise könnte das Spiel im Zustand `SPIEL_INTRO` starten und ein Intro oder ein Titelbild anzeigen. Sobald der Spieler auf eine Taste drückt (*also eine entsprechende Nachricht auslöst*) wechselt das Spiel (*in der Callback Funktion* `WindowProc`) dann in den Zustand `SPIEL_AUSWAHL` wo der Spieler beispielsweise ein Level auswählen möchte das er spielen will, oder seine Spielfigur oder was auch immer.
Der Sinn dieser einzelnen Zustände ist ganz einfach zu erkennen. Wir unterteilen unser Spiel damit in logisch zusammengehörende kleine Häppchen. Anstatt das gesamte Spiel, welches wir entwickeln wollen, auf einmal zu betrachten und nicht zu wissen wo wir anfangen sollen, picken wir uns einfach einen Zustand heraus und implementieren dessen überschaubarere Funktionalität.

Zu guter Letzt messen wir natürlich noch die Laufzeit eines Frames mit Hilfe der Funktion `GetTickCount()`. Diese Funktion liefert uns die Zeit, die seit dem Start des Computers vergangen ist in Millisekunden. Diese Zeit nehmen wir einmal beim Start eines Durchlaufes der Hauptschleife und einmal am Ende, nachdem alle Berechnungen eines Frames innerhalb der Hauptschleife abgelaufen sind. Dann prüfen wir einfach die Differenz zwischen den beiden Zeiten und so lange diese Differenz kleiner ist als 40 lassen wir eine Schleife laufen die nur ein Semikolon als leere Anweisung enthält (*also nix tut ausser zu laufen und Zeit zu kosten*).
Wie kommen wir nun auf 40 und was hat das für einen Sinn? Ganz einfach:

```
1 Millisekunde / 1000 = 0,001 Sekunden
0,001 Sekunden * 40 = 0,04 Sekunden
```

Damit wird festgelegt, dass ein Durchlauf der Hauptschleife ***mindestens*** 0,04 Sekunden dauern muss. Damit haben wir:

```
1 Sekunde / 0,04 Sekunden = 25
```

In einer Sekunde schafft unsere Hauptschleife damit höchstens 25 Durchläufe denn mehr lassen wir nicht zu! Diese Zahl nennt man dann fps oder auch **F**rames **P**er **S**econd denn diese Zahl drückt aus, wie viele Frames unser Spiel pro Sekunde

anzeigt. In einem Durchlauf der Hauptschleife unseres Spiels passiert doch folgendes: Wir fragen die Eingaben des Spielers ab, bewegen die Spielerfigur/-en, lassen die künstliche Intelligenz die computergesteuerten Figuren bewegen und dann malen wir anhand der neuen Bewegungsdaten ein Bild (=*Frame*) auf den Monitor. Jeder Durchlauf der Hauptschleife entspricht also eine Bild auf dem Monitor und wir haben gerade festgelegt, dass wir nie mehr als 25 Bilder je Sekunde anzeigen wollen.

So eine Framerate von 25-30 fps wird vom menschlichen Auge als flüssige Animation erkannt, das Auge kann hier nicht mehr unterscheiden, dass es eigentlich nur mit einzelnen Bildern konfrontiert wird. Wenn die Framerate höher ist und mehr Bilder pro Sekunde angezeigt werden, dann kann das menschliche Auge diese nicht mehr wahrnehmen und der Spieler verpasst einiges von dem was am Monitor angezeigt wird. Das ist auch der Grund dafür warum wir diese Framebremse einbauen. Wenn wir unser Spiel auf unterschiedlichen schnellen Rechnern laufen lassen so soll das Spiel trotzdem überall die gleiche Framerate habe und mit maximal 25 Bilder pro Sekunde laufen. Ein schönes Negativbeispiel hier ist *Origins Wing Commander 1* bei dem eine solche Bremse fehlte. Startete man das Spiel auf einem schnellen 486'er oder gar Pentium Rechner so war man meistens bereits abgeschossen bevor man die feindlichen Raumjäger überhaupt gesehen hat. Diese wurden zwar am Bildschirm dargestellt, aber einfach viel zuuuuuu schnell als dass ein menschliches Auge das hätte wahrnehmen können.
Nun zur anderen Seite der Medaille. Normalerweise kämpft man ja eher damit, sein Spiel so schnell wie möglich zu machen. Wenn die Berechnungen innerhalb eines Durchlaufes der Hauptschleife bereits länger dauern als 0,04 Sekunden so schaffen wir in einer Sekunde auf alle Fälle weniger als 25 fps. Damit kann das Auge aber die einzelnen Frames erkennen und sieht das Spiel nicht mehr als flüssige Animation. Viel mehr sieht man ein Ruckeln des Bildes oder es dauert einfach viel zu lange bis der Computer auf Eingaben reagiert usw. Das einzige was man dagegen tun kann ist eine Optimierung seiner Berechnungen während der Hauptschleife, um einen schnelleren Ablauf zu gewährleisten. Daher gilt auch, dass man sein Spiel nicht nur auf High-End Hardware mit den neuesten fancy 3D Beschleunigern testet, sondern auch auf etwas langsameren 3D Karten. Schliesslich sollen nicht nur die Reichsten unter den Spielern zu unseren Kunden zählen...

Die Windows Prozedur

Unsere `WinMain()` Funktion ist damit bereits komplett. Doch wir wissen natürlich immer noch nicht, wie wir auf Nachrichten reagieren die Windows in unsere Warteschlange packt. Aus der **Abbildung 1** konnten wir bereits sehen, dass Windows diese Nachricht dann an die von uns definierte Callback Funktion (*die man normalerweise immer* `WindowProc()` *nennt*) zur Verarbeitung sendet wenn wir `DispatchMessage()` aufrufen. Ebenso wie der Kopf der `WinMain()` Funktion ist auch der Kopf dieser Callback Funktion, die man allgemein hin als *Windows Prozedur* bezeichnet, bereits fest vorgegeben, wir dürfen lediglich den Funktionsrumpf so schreiben wie wir es wollen:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam);
```

Der Rückgabewert dieser Funktion ist vom Typ `LRESULT` was uns nicht weiter stören soll. Zusätzlich müssen wir diese Funktion auch als vom Typ `CALLBACK` definieren, das ist für Windows Prozedur Funktionen einfach so vorgeschrieben. Nun zu den Parametern des Biests. Der erste Parameter ist das Handle des tatsächlichen Fensters in dem diese Nachricht stattgefunden hat. Der zweite Parameter ist die eigentliche Nachricht selbst und die beiden folgenden Parameter enthalten zwei zusätzliche Informationen zu der Nachricht. Darüber müssen wir uns aber keine Gedanken machen, denn schliesslich ruft Windows diese Callback Funktion für uns auf und füllt daher auch die richtigen Werte für die Parameter!

Unser Job ist nun wie folgt. Wir müssen den Parameter `message` daraufhin prüfen um was für eine Nachricht es sich handelt und dann entsprechend auf Nachrichten reagieren die wir verarbeiten wollen. Alle anderen Nachrichten die uns nicht interessieren dürfen wir aber nicht einfach ignorieren, sondern müssen sie ordentlich an die Windows Funktion `DefWindowProc()` weiterleiten. So sorgt Windows dann dafür, dass diese Nachrichten sauber abgebaut werden und keinen *Stau* verursachen können. Okay, hier ist unsere `WindowProc()`:

```
LRESULT  CALLBACK WindowProc(HWND hwnd,  UINT message,
                             WPARAM wParam,  LPARAM lParam) {
   // Können wir die Message selbst verarbeiten?
   switch(message) {
      // Eine Taste wurde gedrückt
      case WM_KEYDOWN:
         switch (wParam) { // Welche Taste?
            // Falls Escape Taste...
            case VK_ESCAPE:
               {
               //...dann Beenden Nachricht posten!
```

```
            PostMessage(hwnd, WM_CLOSE, 0, 0);
            return 0;
            } break;
         } break;
    // Das Fenster wurde beendet
    case WM_DESTROY:
       {
       blnBeenden = TRUE;
       PostQuitMessage(0);
       return 0;
       } break;
    default:break;
    } // switch
  // Sonst Windows-Standardverarbeitung aufrufen!
  return (DefWindowProc(hwnd, message, wParam, lParam));
  } // WindowProc
```

Zunächst hetzen wir eine `switch` Anweisung auf den `message` Parameter um herauszufinden um welche Nachricht es sich handelt. Wir behandeln hier erst mal nur zwei Fälle. Zum einen die Nachricht `WM_KEYDOWN` die anzeigt, dass irgendeine Taste gedrückt wurde. Dann müssen wir den Parameter `wParam` abfragen, denn dieser enthält die Information darüber welche Taste gedrückt wurde. Von allen möglichen Tasten reagieren wir hier nur auf die Escape Taste. Wurde diese gedrückt so wollen wir unser Programm beenden. Dazu müssen wir uns an den Windows Nachrichtenverkehr anpassen und eine Nachricht an das Betriebssystem schicken.
Mit der Funktion `PostMessage()` senden wir die Nachricht `WM_CLOSE` an Windows um anzuzeigen, dass sich das Fenster mit dem Identifier `hwnd` schliessen möchte. Darauf wird Windows so reagieren, dass es eine entsprechende Nachricht `WM_DESTROY` in die Nachrichtenschlange unseres Fensters `hwnd` stellt.

Daher ist es unser Job auch diese Nachricht `WM_DESTROY` zu verarbeiten. Finden wir diese in unserer Warteschlange so setzen wir die Abbruchvariable der Hauptschleife so, dass die Hauptschleife beendet wird. Abschliessend posten wir dann durch `PostQuitMessage()` die Nachricht an Windows, dass unser Programm nun beendet ist, so dass Windows entsprechende Ressourcen freigeben kann.
Und das waren auch schon die beiden Fälle von Nachrichten die wir hier behandeln wollen. Handelte es sich um eine andere Nachricht so rufen wir wie bereits besprochen die Standard Windows Prozedur auf und ruhen uns auf unseren Lorbeeren aus.

Die 3 Phasen eines Spiels

Bevor wir jetzt den Quelltext für unser kleines Programm endgültig zusammensetzen fehlt uns nur noch eines: ein bisschen Organisation. Neben den einzelnen Zuständen die unser Spiel in der Hauptschleife annehmen kann gibt es noch eine etwas übergeordnetere Unterteilung des Spiels in drei Phasen wie die **Abbildung 2** zeigt.

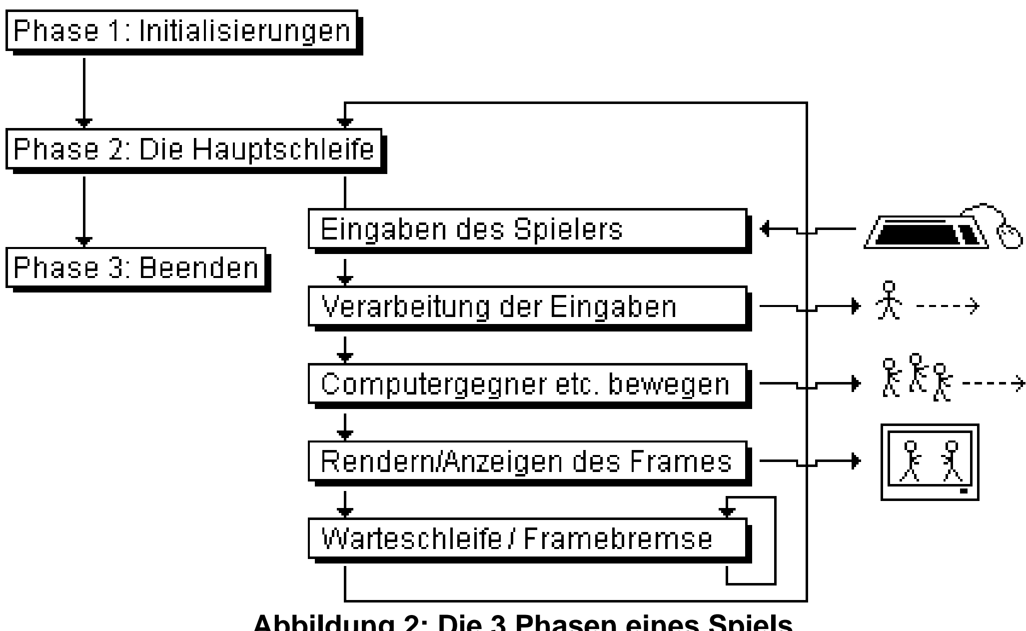


Abbildung 2: Die 3 Phasen eines Spiels

In der ersten Phase beginnen wir mit der Initialisierung des Spiels. Doch wo ist da der Unterschied zu dem Zustand `SPIEL_START` den das Spiel in der Hauptschleife annehmen kann? Nun, während der Phase 1 initialisieren wir die gesamte technische Seite des Spiels. Wir initialisieren dort später beispielsweise Direct3D. Ein komplettes Spiel wird dort auch DirectInput und DirectSound aktivieren und auf die Verwendung in dem Spiel vorbereiten. In dem Zustand `SPIEL_START` geht es dann eher darum, alle Sounds, Texturen, 3D Modelle usw. zu laden die für einen konkreten Level benötigt werden.
Analog dient die dritte Phase dazu, alle initialisierten Komponenten des gesamten Programms herunter zu fahren. Insbesondere DirectX braucht hier noch ein bisschen Nachbearbeitung, um den belegten Speicher wieder freizugeben und Windows nicht zum Absturz zu bringen.

Die zweite Phase kennen wir eigentlich schon, denn das ist genau die Hauptschleife der `WinMain()` Funktion. Die **Abbildung 2** zeigt bereits die wichtigsten Schritte die hier vorgenommen werden müssen, natürlich immer unter der Voraussetzung dass wir uns im Zustand `SPIEL_LAEUFT` befinden.

Wie ich sehe gibt es keine Fragen mehr...? Gut, dann setzen wir die `WinMain()` Funktion endgültig zusammen und bringen auch unsere anderen beiden Phasen ins Spiel:

```
int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hprevinst,
                   LPSTR lpcmdline, int ncmdshow) {
   WNDCLASSEX winclass;                      // Objekt vom Typ Fenster
   const char chKlassenname[] = "Unwichtig"; // Name für das Objekt
   MSG        message;                       // Windows Nachricht
   DWORD      dwStartzeit;                   // Schleifenzeit

   winclass.cbSize        = sizeof(WNDCLASSEX);
   winclass.style         = CS_HREDRAW | CS_VREDRAW;
   winclass.lpfnWndProc   = WindowProc;
   winclass.cbClsExtra    = 0;     // Dies und das...
   winclass.cbWndExtra    = 0;     // Dies und das...
   winclass.hInstance     = hinst; // Dies und das...
   winclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
   winclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
   winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   winclass.lpszMenuName  = NULL;
   winclass.lpszClassName = chKlassenname; // Name unserer Klasse
   winclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

   // Brav bei Windows registrieren lassen
   if (!RegisterClassEx(&winclass))
      return 0;

   // Dann erstellen des eigentlichen Fensters
   if (!(hwnd = CreateWindowEx(NULL,
                   chKlassenname,
                   "3D Spieleprogrammierung",
                   WS_POPUPWINDOW | WS_VISIBLE,
                   10,10,
                   400,300,
                   NULL,
                   NULL,
                   hinst,
                   NULL)))
      return 0;

   // Startwerte festlegen
   Spiel_Zustand = SPIEL_AUSWAHL;

   // PHASE 1:
   Spiel_Initialisieren();

   // PHASE 2:
   // H A U P T S C H L E I F E ///////////////////////
```

```
   while (!blnBeenden) {
      // Ist eine Nachricht zu verarbeiten?
      while(GetMessage(&message,NULL,0,0)) {
         TranslateMessage(&message);
         DispatchMessage(&message);
         } // GetMessage

      // Startzeit der Hauptschleife
      dwStartzeit = GetTickCount();

      switch(Spiel_Zustand) {
         case SPIEL_AUSWAHL:
            {
            // [...]
            Spiel_Zustand = SPIEL_START;
            } break;
         case SPIEL_START:
            {
            // [...]
            Spiel_Zustand = SPIEL_LAEUFT;
            } break;
         case SPIEL_LAEUFT:
            {
            // [...]
            } break;
         case SPIEL_NEUSTART:
            {
            // [...]
            Spiel_Zustand = SPIEL_START;
            } break;
         default: break;
         } // switch

      // Framerate bremsen
      while ((GetTickCount() - dwStartzeit) < 40) /*tu nix*/;
      } // Hauptschleife

   // PHASE 3:
   Spiel_Beenden();

   return message.wParam;
   } // WinMain
/*----------------------------------------------------------*/


// PHASE 1:
BOOL Spiel_Initialisieren(void) {
   // Zu erledigen:
   // Initialisierungen
   return TRUE;
   }
/*----------------------------------------------------------*/


// PHASE 3:
BOOL Spiel_Beenden(void) {
   // Zu erledigen:
   // Speicher freigeben etc.
   return TRUE;
   }
/*----------------------------------------------------------*/
```

Was tut unser Programm nun? Eigentlich noch nicht viel und doch schon eine ganze Menge. Wenn man das Projekt als

Win32 Anwendung erstellt, kompiliert und ausführt so wird man feststellen, dass unser Programm ein weisses Fenster ohne Menüleiste oder sonstigen Schnickschnack erzeugt welches wir durch Druck auf die Escape Taste beenden können. Nicht mehr...aber auf nicht weniger!

Und hier gibt es den Code des *gesamten* Tutorials zum Download: Projektdateien

---

Weiter geht's zum Kapitel 3...

SPIELEENTWICKLUNG MIT DIRECT3D IM - KAPITEL 3

von Stefan Zerbst

"Now listen up people. I want this thing to go smooth and by the numbers."
Lieutenant Goreman, Aliens

Direct3D 8 initialisieren

Jetzt haben wir ein schickes Windows Fenster, toll. Dem Fenster fehlt es aber noch an zwei Dingen. Zum einen ist es etwas zu klein und zum anderen kann es noch nichts in 3D darstellen. Obwohl das Programm, welches wir in diesem Kapitel entwickeln werden, auch noch keine 3D Ausgabe haben wird, so bietet es doch bereits das entsprechende Potential um direkt auf die verfügbare Hardware zuzugreifen und 3D Dinge zu rendern. Und da wir gerade bei direkt sind haben wir auch schon geklärt woher DirectX wahrscheinlich seinen Namen hat.
In der Tat werden wir unser Fenster nun darauf einrichten, durch Direct3D auf ein echtes Vollbild aufgeblasen zu werden. Dadurch übernehmen wir mehr oder weniger die gesamte Prozessorkapazität für unser Programm und müssen uns die Rechenzeit nicht mit anderen Windows Prozessen oder anderen parallel laufenden Programmen teilen. Wenn man nämlich in reinen Fenstermodus arbeitet, so verteilt Windows ganz gerecht die Rechenleistung des Computers in dem es alle paar Bruchteile einer Sekunde jedes Programm kurz laufen lässt, danach sind wieder die anderen der Reihe nach dran usw. Wenn wir aber ein echtes Vollbild Programm machen, dann verhindern wir damit dass andere Prozesse / Programme etwas von der Rechenleistung abbekommen.

ACHTUNG: Wir müssen unser Projekt nun zu der Datei `d3d8.lib` linken, denn diese enthält die Direct3D Funktionen und Datentypen die wir hier verwenden werden.

Enumeration und wozu sie gut ist

Es gibt zwei Wege um Direct3D zu initialisieren, also für die Verwendung in unserem Programm vorzubereiten. Zum einen den einfachen Weg und zum anderen den schweren Weg, welchen wir hier beschreiten werden. Der einfache Weg zeichnet wich dadurch aus, dass man Direct3D durch zwei einfache Funktionsaufrufe initialisieren kann und zwar mit Standardwerten. Das führt aber dazu, dass Direct3D automatisch die erste Grafikhardware verwenden wird die auf dem Zielcomputer auf dem das Programm läuft installiert ist. Verwendet jemand aber beispielsweise eine 2D Grafikkarte die auch 3D Operationen unterstützt und zusätzlich noch ein super (-*teures*) 3D Beschleunigerboard so wird Direct3D unter Umständen die ungeeignete 2D Karte auswählen und seine Grafikausgabe über diese langsame Karte laufen lassen. Der Spieler ärgert sich dann - zu recht - darüber, dass unser Spiel saulangsam läuft obwohl er die beste 3D Karte auf dem Markt installiert hat.

Der harte Weg ist also folgender: Wir müssen Funktionen schreiben mit deren Hilfe wir herausfinden können was für Hardware auf einem Computer installiert ist und dann eine geeignete davon auswählen. Wir *enumerieren* also sämtliche installierte Grafikhardware, was so viel heisst wie suchen der verfügbaren Hardware.

Sinn der 3D Hardware

Noch ein Wort zu Hardwarebeschleunigung: Die meisten 3D Bibliotheken, und auf alle Fälle auch Direct3D, können im Endeffekt wenig mehr als nur Dreiecke auf den Bildschirm zu rendern (*rendern* = *malen*). Man spricht dabei auch oft im Englischen von Rasterization aus folgendem Grund: Wenn man zwischen drei Punkten auf einem Blatt Papier ein Dreieck malen will, so verbindet man einfach die Punkte mit einem Stift und malt die

Fläche aus. Dabei erhält man eine Fläche aus vielen, unendlich kleinen Punkten. Auf einem Computerbildschirm geht das aber nicht. Jeder Punkt ist maximal so klein wie ein Pixel. Früher konnten Spiele gerade mal 320x200 Pixel grosse Bildschirme verwenden und man konnte die einzelnen Pixel noch relativ gut sehen. Heutzutage ist sicherlich schon 800x600 Pixel Standard aber selbst bei einer Auflösung von 1024x768 oder noch grösser führt kein Weg daran vorbei, dass die einzelnen Punkte auf dem Bildschirm eben nicht unendlich klein sind.
Ein Dreieck Rasterizer muss daher die drei Punkte des Dreieckes nehmen und dann berechnen, welche Pixel auf dem Bildschirm am besten zu den unendlichen kleinen Punkten des eigentlichen Dreiecks passen. Dieser Prozess ist für Dreiecke bereits kompliziert genug (*wenn es schnell genug gehen soll*), wird aber bei Vierecken, Fünfecken, usw. immer komplizierter. Im Endeffekt werden also alle 3D Modelle in Dreiecke zerlegt und als solche rasterisiert.

Den Algorithmus der Dreieck Rasterization kann man nun ganz normal in einer C Funktion implementieren, dann spricht man von einer Software Implementierung. Wesentlich besser ist es aber, wenn man dasselbe in Hardware implementiert. Das bedeutet dass man diesen Algorithmus in einem Chip auf der Grafikkarte speichert und die Grafikkarte stellt dem Algorithmus dann einen eigenen Prozessor für die notwendigen Berechnungen bereit. Damit ist die Rasterization komplett auf die Grafikhardware ausgelagert und kostet den Haupt-Prozessor des Computers keine Rechenzeit. Wir können also die gesamte Rechenzeit des Prozessors nutzen um unsere anderen Funktionen auszuführen und das gesamte (*sonst viel zu langsame Rendering*) wird dann vor der Grafikhardware erledigt. Daher bieten heutzutage auch schon fast alle normalen 2D Grafikkarten das Rendern von Dreiecken in Hardware an und auf den Packungen der 3D Beschleuniger wird man immer die Angaben darüber finden, wie viele Dreiecke die Karte in wie kurzer Zeit malen kann.

Kauderwelsch

Was wäre die Welt ohne Fachsprachen? Ob nun Mathematiker, Physiker, Informatiker, BWL'er, Lehrer, Schüler, Finnen...jede soziale Gruppe verwendet ihre eigenen *Slang*-Begriffe in die man erst mal eingeweiht werden muss. So auch wir Direct3D'ler. In diesem Absatz werden wir uns also erst mal die notwendigen Fachbegriffe aneignen um die nachfolgenden Code gut verstehen und mit Dingen in der realen Welt verbinden zu können.

Die Grafikkarten eines Computers werden nun bei Direct3D als **Adapter** bezeichnet. Jeder Adapter hat verschiedene mögliche Bildschirmauflösungen, also 640x480 oder 800x600 mit jeweils wiederum verschiedenen Farbtiefen wie 8 Bit, 16 Bit oder 32 Bit Farben. Diese Bildschirmauflösungen werden nun als **Modi** (*Einz.: Modus*) bezeichnet. Zusätzlich verfügt jeder Adapter (*jede Grafikkarte*) in der Regel über ein oder mehrere angeschlossene **3D Devices**. Bei modernen 2D Grafikkarten sind das OnBoard 3D Chipsets aber auch insbesondere separate 3D Beschleunigerchips (*Voodoo*) sind 3D Devices. Sollte ein Adapter mehrere Devices oder ein Computer mehrere Adapter installiert haben, beispielsweise eine gute 2D Karte mit einem 3D Chipset (*gut*) und zusätzlich noch eine 3D Beschleunigerkarte (*besser*!!!) dann wird die Standardinitialisierung von Direct3D unter Umständen das schlechtere der beiden Devices auswählen und der Spieler ist genervt dass er teure Hardware gekauft hat die wir nicht ausnutzen.

Unser Job ist es also eine Liste aller verfügbaren Adapter (*in der Regel nur einer*) auf dem Computer durch Enumeration zu ermitteln. Für jeden Adapter suchen wir dann die möglichen Modi, also Bildschirmeinstellungen, die der jeweilige Adapter unterstützt. Danach suchen wir für jeden Adapter alle angeschlossenen 3D Devices und prüfen nach, welche der möglichen Modi des Adapters von dem jeweiligen Device unterstützt werden. Bei der gesamten Sucherei erstellen wir eine Liste als Array aus den Datenstrukturen `XD3DADAPTER`, `XD3DMODE` und `XD3DDEVICE`. Das Adapter Array ist dabei global und enthält das Device Array, welches wiederum das Modi Array enthält. Aber das werden wir gleich im Code sehen.

Nachdem die Sucherei beendet ist wählen wir aus der globalen Liste einen Adapter und ein Device aus. Dazu lassen wir eine Schleife über das Adapter Array laufen in welches wir bei der Enumeration nur Devices mit Hardwarebeschleunigung (HAL) aufgenommen haben und wählen einfach das erste aus. Das wird für unsere Zwecke genügen da wir somit sichergestellt haben, wenigstens 3D Hardwarebeschleunigung zur Verfügung zu haben. Aber der Code enthält ebenfalls die Funktion `xD3DEnum_Device_Okay()` in der man ein Device auf spezielle Fähigkeiten (*zB. Single-Pass*) hin prüfen kann, bevor es in die Liste aufgenommen wird. Bisher gibt

diese Funktion für jedes Device den Wert `TRUE` zurück und steht einer Aufnahme in die Liste also nicht im Wege. In diese Funktion kann man dann aber bei Bedarf eine Abfrage für die Funktionalitäten einbauen die die Hardware unterstützen soll, beispielsweise also das Multi Texturing oder andere Spezial Effekte. Aber keine Panik, dazu kommen wir später noch.

Datenstrukturen

Wir beginnen mit einer Datenstruktur für die Adapter (=*Grafikkarten*) die wir während der Enumeration finden werden.

```
/**
 * Struktur für Adapterdaten, inkl. Liste  verfügbarer 3D Devices
 */
struct XD3DADAPTERINFO {
   // Adapter Daten
   D3DADAPTER_IDENTIFIER8 d3dAdapterIdentifier;
   // Verfügbare, enumerierte 3D Devices des Adapters
   DWORD          dwAnz_Devices;
   XD3DDEVICEINFO aDevices[5];
   // Aktuelles, enumeriertes, kompatibles Devices
   DWORD          dwAkt_Device;
   };
/*----------------------------------------------------*/
```

Wir speichern in der Adapterstruktur insbesondere wie viele Devices an den Adapter angeschlossen sind (*OnBoard 3D Chips, 3D Beschleunigerkarten, usw.*), ein Array mit den Daten für die Devices und ein Element welches Device aus dem Array dieses Adapters wir verwenden wollen falls wir diesen Adapter auswählen. Damit sind wir auch schon bei der Datenstruktur die wir für alle 3D Devices anlegen werden die wir finden:

```
/**
 * Struktur für 3D Device, inkl.Liste verfügbarer Modi
 */
struct XD3DDEVICEINFO {
   // Device Daten
   D3DDEVTYPE   devTyp;  // Reference, HAL, etc.
   D3DCAPS8     d3dCaps; // Capabilities des Device
   const CHAR*  achName; // Name des Device

   // Verfügbare Modi des Device
   DWORD        dwAnz_Modi;
   XD3DMODEINFO aModus[150];

   // Aktuelle, enumerierte, kompatible Einstellung
   DWORD        dwAkt_Mode;
   D3DMULTISAMPLE_TYPE MultiSampleType;
   };
/*----------------------------------------------------*/
```

Zu den einzelnen Elementen sage ich dann bei Zeiten etwas, wir sehen aber dass wir auch hier einen Zähler haben wie viele Modi ein Device unterstützt, ein Array der entsprechenden Modi Informationen und dann noch eine Variable die sagt welchen Modus aus diesem Array wir verwenden werden.
Okay, fehlt nur noch eine Struktur für die Informationen über einen Bildschirmmodus:

```
/**
 * Struktur für Informationen eines Device-Modus
 */
```

```
struct XD3DMODEINFO {
   DWORD     dwBreite;           // Bildschirmbreite
   DWORD     dwHoehe;            // Bildschirmhöhe
   D3DFORMAT Format;             // Pixel Format
   DWORD     dwEgnschftn;        // HW/SW/Mixed TnL Operationen
   D3DFORMAT DepthStencilFormat;// Z/Stencil Format
   };
/*-----------------------------------------------------*/
```

Bewaffnet mit diesen Datenstrukturen werfen wir uns nun in den Kampf um die Enumeration der verfügbaren Hardware.

Globale Variablen

Beginnen wir unsere Arbeit also damit uns zu überlegen, welche Werte wir global speichern müssen. Offensichtlicherweise kann es nie schaden, das Handle auf das Fenster unserer Anwendung zu kennen. Schließlich soll Direct3D später dieses Fenster übernehmen, und dazu müssen wir Direct3D eben dieses Handle mitteilen. Man sollte hier auch immer im Kopf haben, dass eine globale Variable schneller ist als wenn wir dieses Handle beispielsweise immer als Funktionsparameter übergeben würden.

```
extern HWND hwnd;    // Handle des Hauptfensters
```

Als nächstes kommen schon die ersten Direct3D Objekte ins Spiel. Zum einen haben wir hier ein Objekt für den BackBuffer unseres Games vom Typ `IDirect3DSurface8`. Eine Surface ist im Prinzip so etwas wie ein Bitmap Bild und unser Game benötigt wenigstens zwei dieser Surface Objekte. Einen **FrontBuffer** der am Bildschirm zu sehen ist und einen **BackBuffer** in den bereits der nächste Frame gerendert wird während der FrontBuffer noch auf den Bildschirm gemalt wird. Dieses Prinzip von zwei Render-Targets nennt man Flipping Chain und sie dient dazu, den gesamten Ablauf des Renderns und der Anzeige eines Frames am Bildschirm zu beschleunigen. Während Direct3D einen Buffer auf den Bildschirm rendert müssen wir nicht wertvolle Zeit mit Warten verschwenden sondern malen den nächsten Frame in den zweiten Buffer.

Seit der neuen Version 8 von DirectX müssen wir uns mit diesen Details nicht mehr so sehr abmühen, denn sowohl FrontBuffer als auch BackBuffer werden nun automatisch für uns erzeugt. Da man aber immer mal wieder selbst Daten *von Hand* zur Anzeige bringen möchte, beispielsweise Spielstände, Munitionszähler usw., ist es immer ratsam sich selbst einen globalen Pointer auf den BackBuffer zu organisieren was wir später auch tun werden. Daher legen wir eine globale Variable dafür an. Wann immer wir irgend etwas selbst auf den Schirm malen wollen haben wir über diese Variable Zugriff.

```
extern LPDIRECT3DSURFACE8 lpDDSBack;    // BackBuffer
```

Die nächste globale Variable ist eigentlich die wichtigste von allen, denn das Direct3D Device vom Typ `IDirect3DDevice8` ist das Objekt, über das wir eigentlich alle 3D Arbeit erledigen lassen. Also muss auch hier ein globaler Pointer her, den wir von überall verwenden können. Daneben benötigen wir noch ein Hauptinterface für Direct3D welches aber für unsere praktische Arbeit eigentlich gar nicht so wichtig ist. Darum ist es ein globaler Pointer der nur in dieser Datei bekannt sein muss.

```
extern LPDIRECT3DDEVICE8 lpD3DDevice;  // Direct3D Device
LPDIRECT3D8              lpD3D;        // Direct3D Hauptobjekt
```

Okay, lassen wir uns das noch einmal durch den Kopf gehen. Wir benötigen das Direct3D Hauptinterface Objekt, um ein Direct3D Device Interface Objekt zu erstellen. Über dieses Device Interface Objekt können wir dann alle wichtigen Funktionen von Direct3D (*beispielsweise das Rendern von textuierten Dreiecken*) aufrufen. Dieses Device Interface Objekt ist sozusagen unsere Verbindung zu dem physischen 3D Hardware Device welches im Computer installiert ist.

Jetzt wird es wieder etwas spannender denn nun kommt alles was irgendwie mit der Enumeration zusammenhängt. Wir haben ein Array für alle enumerierten Adapter (=*Grafikkarten*) des Zielcomputers auf dem

unser Spiel läuft. In diesem Beispiel gehen wir mal davon aus, dass das höchstens 20 Stück sind. Dazu benötigen wir noch einen globalen Zähler wie viele Adapter wir gefunden haben und eine Variable in der wir speichern werden, welchen Adapter wir nach der Enumeration ausgewählt haben:

```
XXXD3DADAPTERINFO g_Adapter[20];        // Enumerierte Grafikadapter
DWORD             g_dwAnz_Adapter =0;   // Anzahl enumerierter Adapter
DWORD             g_dwAdapter     =0L;  // Ausgewählter Adapter
```

Nun fehlt wirklich nicht mehr viel, bevor wir mit dem Coden anfangen können. Hier haben wir noch zwei globale Variablen in denen wir speichern werden wie viele Bits und Z-Buffer (=*DepthBuffer*) und wie viele Bits unser Stencil Buffer gross sein soll. Der Z-Buffer dient dazu, die Tiefeninformationen der gerenderten Objekte zu speichern, so dass wir die Objekte vor dem Rendern nicht sortieren müssen. Direct3D kann so entscheiden, welches Objekt im Vordergrund liegt und damit die hinteren Objekte verdeckt. Der StencilBuffer dient für diverse Special Effects. Einerseits wird der Buffer um so genauer je grösser die Bitanzahl weil dann einfach genauere Kommazahlen gespeichert werden können. Andererseits verbraucht ein Buffer mit grösserer Bit Tiefe auch mehr Speicherplatz.

Aber das ist ja noch nicht das Thema hier. Abschliessend benötigen wir noch zwei Zeichenketten in denen wir Informationen über das ausgewählte Device speichern werden. Schliesslich wollen wir ja später auch wissen, welche Hardware wir erwischt haben. Das dient aber nur zu Testzwecken und hat keinen tieferen Sinn.

```
DWORD g_dwMinDepthBits,         // Z-Buffer Bits
      g_dwMinStencilBits;       // Stencil Buffer Bits
char  g_D3DDevicename[256],     // Name des Device
      g_D3DDevicemode[100];     // Modus des Device
```

Erstellen der Hardwareliste

Alle noch da? Dann fangen wir jetzt die eigentliche Arbeit an und erstellen uns eine Liste mit der gesamten Grafikhardware die auf dem Computer installiert ist auf dem unser Programm läuft. Wir schreiben uns aber vorher ein handliche Funktion die uns später einen einfachen Start unserer Engine erlaubt.
Die folgende Funktion nimmt als Parameter die gewünschte Bildschirmbreite, -höhe, die Z Buffer Bit Tiefe und die Stencil Buffer Bit Tiefe auf und erledigt dann still und heimlich die gesamte Arbeit und startet bei Erfolg der Arbeit Direct3D mit den gewünschten Werten. Dazu führt die Funktion drei Schritte aus: (a) Zuerst wird das Direct3D Hauptinterface Objekt angelegt und in der globalen Variablen gespeichert. Dann wird (b) die Funktion aufgerufen die die Enumeration durchführt und die geeignete Hardwarekombination (*Adapter* + *Device* + *Modus*) auswählt. Im letzten Schritt (c) initialisieren wir dann die ausgewählte Hardware für die Verwendung von Direct3D.

```
BOOL xD3D_initialisieren(DWORD dwBreite,       // Bildschirm x
                         DWORD dwHoehe,        // Bildschirm y
                         DWORD dwZ_Bits,       // Z Bit Tiefe
                         DWORD dwStencil_Bits) // Stencil Bits
   {
   BOOL blnOkay;

   // Werte in globalen Variablen speichern
   g_dwMinDepthBits   = dwZ_Bits;
   g_dwMinStencilBits = dwStencil_Bits;

   // (a) Erstelle das Direct3D Hauptinterface-Objekt
   g_lpD3D = Direct3DCreate8(D3D_SDK_VERSION);
   if (g_lpD3D == NULL) {
      fprintf(Protokoll, "Fehler: Direct3DCreate8() failed\n");
      return FALSE;
      }
```

```
   // (b) Erstellt eine Liste von Adaptern (Grafikkarten), deren mögl.
   // Grafikmodi und deren angeschlossenen 3D Devices. Verwendet die
   // Funktionen xD3DEnum_Device_Okay() um die Devices zu verifizieren
   // die die Anforderungen der Applikation erfüllen
   blnOkay = xD3D_Erstelle_Deviceliste(dwBreite, dwHoehe);
   if (!blnOkay) {
      fprintf(Protokoll, "Fehler: Erstelle_List() failed\n");
      return FALSE;
      }

   // (c) Initialisierung des Devices und BackBuffer
   blnOkay = xD3D_Objekte_initialisieren();
   if (!blnOkay) {
      fprintf(Protokoll, "Fehler: Obj_Init() failed\n");
      return FALSE;
      }

   fprintf(Protokoll,"D3D_Init: successful \n");
   return TRUE;
   } // xD3D_initialisieren
/*-------------------------------------------------------*/
```

Wie man übrigens auch sehen kann werden wir unsere Fehler- und Erfolgsmeldungen von jetzt ab in die Textdatei `Protokoll` ausgeben. Diese wird in Phase 1 des Spiels durch `fopen()` geöffnet und in Phase 3 durch `fclose()` geschlossen. In der Zwischenzeit bleibt sie die ganze Zeit für uns geöffnet...wenn das mal auch für die Supermärkte gelten würde!

Okay, nun ist es aber echt an der Zeit dass wir unseren hohen Orbit um den heissen Brei herum verlassen und uns direkt auf die Erstellung der Hardwareliste werfen. Die entsprechende Funktion heisst `xD3D_Erstelle_Deviceliste` und nimmt als Parameter die Breite und Höhe des Bildschirms auf die wir gerne haben wollen. In diesem Beispiel werden wir unsere Funktion fest verdrahten so dass sie einen 16 Bit Bildschirmmodus auswählt. Bei Bedarf kann man das noch ändern aber 8 Bit bieten einfach zu wenig Farben und 32 Bit verbrauchen meiner Meinung nach zu viel Speicher um heute bereits als Standard verwendet werden zu können.

Die folgende Funktion ist leider etwas sehr lang und unübersichtlich...wenigstens auf den ersten Blick. Daher werden wir sie nicht in ihrer gesamten Schönheit betrachten, sondern Häppchen für Häppchen vorgehen. Aber fangen wir einfach mal an, so schlimm wird es schon nicht:

```
/**
 * Erstellt Liste aller Adapter, Bildschirmmodi und 3D Devices
 */
BOOL xD3D_Erstelle_Deviceliste(DWORD dwBreite, DWORD dwHoehe) {
   const DWORD      dwAnz_Devicetypen = 2;
   const TCHAR*     achDevicetypen[]  = { "HAL", "REF" };
   const D3DDEVTYPE adevDevicetypen[] = { D3DDEVTYPE_HAL,
                                          D3DDEVTYPE_REF };
   BOOL blnHAL_exists     = FALSE;
   BOOL blnHAL_kompatibel = FALSE;
```

wird fortgesetzt...

So, hier müssen wir auch schon gleich wieder bremsen. Es gibt auf dem Markt hunderte verschiedene 3D Devices. Direct3D unterteilt diese jedoch in drei verschiedene Kategorien (*die Hardwarehersteller müssen ihre Karte entsprechend einstellen so dass sie sich bei Direct3D richtig identifiziert*).

- `D3DDEVTYPE_HAL`

Die Kategorie **H**ardware **A**bstraction **L**ayer, oder kurz HAL, besagt dass das Device dazu in der Lage ist Dreiecke über Hardwareimplementierungen zu rendern. Die Grafikkarte selbst malt sozusagen die Grafik was wesentlich schneller ist als wenn Softwarealgorithmen das machen müssten. Die Berechnung der Beleuchtungseffekte und der 3D Transformationen (*Bewegung und Drehung von 3D Objekten*) wird bei dem HAL dann entweder über Software oder bei den teureren Karten ebenfalls schneller über Hardware erledigt. Letzteres sind die sogenannten **T**ransformation a**n**d **L**ighting, oder kurz TnL, 3D Karten also die Ferraris unter der Hardware.

- `D3DDEVTYPE_SW`
Die zweite Kategorie ist ein reines Software Device. Das bedeutet dass auf dem Computer gar keine 3D Hardware installiert ist, lediglich eine 2D Grafikkarte. In diesem Fall muss Direct3D das Rendern (*also das Malen der Dreiecke*) selbst durch Softwarealgorithmen durchführen was wesentlich langsamer ist als ein HAL. Dazu kommt noch das Problem, dass ein SW Device nicht alle Funktionalitäten von Direct3D ausführen kann. Entweder bauen wir also überall in unseren Code eine Fallunterscheidung zwischen HAL und SW ein oder wir ignorieren das SW Device einfach während der Enumeration und legen damit fest, dass unser Programm die Zusammenarbeit mit einem reinen SW Device ablehnen wird.

- `D3DDEVTYPE_REF`
Der dritte Device Typ ist der **Ref**erence Rasterizer. Dieser ist ebenfalls ein reines Software Device, bietet dafür aber volle Unterstützung der gesamten Direct3D Funktionalität. ABER: Dieser Device Typ ist lediglich zu Testzwecken implementiert weil er sehr langsam ist. Für den Fall dass einem während der Softwareentwicklung mal kein HAL zur Verfügung steht kann man seine Programme durch den `REF` testen da er wirklich alle Hardware Funktionalitäten ausführen kann. Microsoft rät aber in der DirectX Dokumentation dringend davon ab, ein `REF` Device in einem finalen Release zu verwenden.

Nun gut, wir einigen uns also darauf nur HAL und `REF` Devices in die Liste aufzunehmen da ein SW Device für ein 3D Spiel absolut keinen Sinn macht - es ist einfach zu langsam. In einem finalen Release sollte man aber nur das HAL verwenden. Dann machen wir mit unserer Funktion mal weiter. Wir haben noch schnell zwei `BOOL` Variablen deklariert in denen wir speichern werden ob wir während der Enumeration ein HAL gefunden haben und ob dieses für unsere Zwecke geeignet ist. Sobald diese Aussagen dann auf ein Device zutreffen setzen wir die Variablen auf `TRUE` und wissen, dass wir in unserer Liste wenigstens ein geeignetes Device gefunden haben. Anderenfalls müssen wir unser Programm leider beenden.

Aber weiter geht's mit unserer Funktion. Nun werden wir folgendes tun: Wir werden eine Schleife laufen lassen über die Anzahl aller installierten Adapter (*Grafikkarten*) auf dem Zielcomputer. Diese Anzahl liefert uns die Funktion `GetAdapterCount()` des Direct3D Hauptinterface Objektes. Dann erstellen wir uns einen Zeiger auf die nächste freie Position im globalen Array `g_Adapter[]` das ja unsere Adapterliste enthalten soll wenn wir hier fertig sind. Die nächste freie Position wird übrigens durch die globale Variable `g_dwAnz_Adapter` angezeigt, die ja zu Beginn auf 0 steht.
Im Verlauf der Schleife füllen wir dann alle notwendigen Informationen in die ausgewählte Position im globalen Array. Stellen wir aber fest dass ein Adapter ungeeignet ist, so erhöhen wir den Zähler `g_dwAnz_Adapter` nicht, so dass wir im nächsten Schleifendurchlauf wieder an derselben Indexposition die Werte des neuen Adapters speichern. Erst wenn ein Adapter in unserem Sinne geeignet ist, dann erhöhen wir den Zähler und der Adapter hat damit unwiderruflich einen Platz in unserer Liste erhalten:

xD3D_Erstelle_Deviceliste() Fortsetzug:

```
// Starte Schleife über alle Adapter (Grafikkarten) auf
// dem Zielcomputer. Im Normalfall ist das nur einer!
for(UINT iAdapter=0; iAdapter<g_lpD3D->GetAdapterCount(); iAdapter++) {
   // Zeiger auf die nächste freie Position im globalen Array
   XD3DADAPTERINFO* pAdapter = &g_Adapter[g_dwAnz_Adapter];
   // Identifier des aktuellen Adapters holen
   g_lpD3D->GetAdapterIdentifier(iAdapter, 0,
                                 &pAdapter->d3dAdapterIdentifier);

   // Aktueller Adapter hat noch keine 3D Devices
```

```
   pAdapter->dwAnz_Devices = 0;
   pAdapter->dwAkt_Device  = 0;

   // Arrays und Zähler für Enumeration bereitstellen
   D3DDISPLAYMODE aModus[100];
   D3DFORMAT      aFormat[20];
   DWORD          dwAnz_Formate = 0;
   DWORD          dwAnz_Modi    = 0;

   // Wie viele Modi hat der Adapter
   DWORD dwAnz_Adapter_Modi = g_lpD3D->GetAdapterModeCount(iAdapter);
```

wird fortgesetzt...

Danach setzen wir ein paar Startwerte für die entsprechenden Elemente und Zähler der Adapter Datenstruktur und erstellen noch Zähler für die Formate und Modi dieses Adapters. Dann beschaffen wir uns über die Funktion `GetAdapterModeCount()` des Direct3D Hauptinterface Objektes die Anzahl der Bildschirmmodi die von diesem Adapter ausgeführt werden können. Das sind beispielsweise die Bildschirmmodi 800x600 bei 8 Bit Farben oder 800x600 bei 16 Bit Farben usw.
Nun müssen wir uns mal ganz gut konzentrieren, denn jetzt geht es ans Eingemachte! Das Array `aModus[]` werden wir jetzt gleich füllen. Wir lassen dazu eine Schleife über alle verfügbaren Modi des Adapters laufen und speichern deren Daten in das Array um. Später, wenn wir die an diesen Adapter angeschlossenen Devices suchen, müssen wir dann alle Modi in diesem Array prüfen ob sie auch von dem Device und nicht nur von dem Adapter unterstützt werden. Erst dann können wir ein unterstütztes Format auch wirklich in die endgültige Liste in unserem globalen Device Array aufnehmen!!!

Im folgenden Codeteil werden wir also diese Schleife ausführen und alle Modi des Adapters ansehen. Durch die Direct3D Hauptinterface Funktion `EnumAdapterModes()` können wir uns den jeweils nächsten Modus des Adapters geben lassen. Dann unterziehen wir diesem Modus ein paar Tests, beispielsweise nehmen wir keinen Modus mit weniger als 640x400 Pixeln auf, das ist unter unserer Würde. Danach prüfen wir ob wir einen Modus für ein Adapter bereits in dem Array `aModus[]` haben. Bei unterschiedlichen Bildschirm Refresh Rates kann das passieren. Ist dies nicht der Fall, so übernehmen wir den Modus in das Array. Dasselbe tun wir dann mit dem Format eines Modus. Dieses Format gibt die Bittiefe des Modus an, also 8 Bit, 16 Bit oder 32 Bit (*jeweils in verschiedenen Variationen wie wir später noch sehen werden*). Das Format speichern wir dann in dem Array `aFormat[]`.

xD3D_Erstelle_Deviceliste() Fortsetzug:
(immer noch in der Schleife für Adapter)

```
// Schleife für die Enumeration aller Bildschirm
// Modi des aktuellen Adapters
for (UINT iMode=0; iMode<dwAnz_Adapter_Modi; iMode++) {
   // Display Mode Eigenschaften abfragen
   D3DDISPLAYMODE DisplayMode;
   g_lpD3D->EnumAdapterModes(iAdapter, iMode, &DisplayMode);

   // Ungeeignete Displaymodes verwerfen
   if( (DisplayMode.Width<640) || (DisplayMode.Height<400) )
      continue;

   // Ist der Modus schon vorhanden (zB.bei RefreshRates)?
   for (DWORD dwM=0L; dwM<dwAnz_Modi; dwM++) {
      if (( aModus[dwM].Width  == DisplayMode.Width ) &&
          ( aModus[dwM].Height == DisplayMode.Height) &&
          ( aModus[dwM].Format == DisplayMode.Format)   )
         break;
      } // for [AnzModi]
```

```
   // Falls nicht füge Modus der Liste hinzu
   if (dwM == dwAnz_Modi) {
      aModus[dwAnz_Modi].Width  = DisplayMode.Width;
      aModus[dwAnz_Modi].Height = DisplayMode.Height;
      aModus[dwAnz_Modi].Format = DisplayMode.Format;
      aModus[dwAnz_Modi].RefreshRate = 0;
      dwAnz_Modi++;

      // Existiert das Format des Modus schon
      for (DWORD dwF=0; dwF<dwAnz_Formate; dwF++) {
         if (DisplayMode.Format == aFormat[dwF])
            break;
         } // for [AnzFormate]

      // Falls nicht, füge Format der Liste hinzu
      if (dwF == dwAnz_Formate)
         aFormat[dwAnz_Formate++] = DisplayMode.Format;
      } // if [dwM=AnzModi]
   } // for [AnzAdapterModi]

// Sortiere Display-Modus Liste (nach Format, Breite, Höhe)
qsort(aModus, dwAnz_Modi, sizeof(D3DDISPLAYMODE),
      xD3D_Sortiere_Modi);
```

wird fortgesetzt...

So, wo stehen wir nun? Wir haben jetzt zwei lokale Listen. Eine für alle Modi (*Bildschirmauflösungen*) und eine für alle Formate (*Bittiefen*) die der aktuelle Adapter unterstützt, wir sind ja immer noch in der Schleife über alle Adapter. Nun sortieren wir durch die C Funktion `qsort()` die Liste der Modi. Ich setze hier mal voraus, dass jeder mit dieser Funktion vertraut ist. Wer das nicht ist, der soll einfach mal so hinnehmen dass das hier so funktioniert. Als Sortierungskriterium verwenden wir die Funktion `xD3D_Sortiere_Modi` die ich jetzt um Verwirrung zu sparen nicht zwischendurch hier ablichte. Die drei Zeilen kann sich aber jeder in dem Demo zu diesem Kapitel ansehen und verstehen.

Machen wir also weiter mit unserem Adapter. Wir haben nun seine Modi enumeriert, was tun wir jetzt? Richtig, wir untersuchen die 3D Devices die an diesen Adapter angeschlossen sind. Wir lassen also eine Schleife über alle Direct3D Device Typen laufen die wir zulassen wollen (*hier HAL und REF*) und untersuchen diese dann auf ihre Eignung.

xD3D_Erstelle_Deviceliste() Fortsetzug:
(immer noch in der Schleife für Adapter)

```
// Devices zur Liste des Adapters hinzufügen
for (int iDevice=0; iDevice<dwAnz_Devicetypen; iDevice++) {
   XD3DDEVICEINFO* pDevice;

   // Daten über das Device abfragen und speichern
   pDevice = &pAdapter->aDevices[pAdapter->dwAnz_Devices];
   pDevice->devTyp = adevDevicetypen[iDevice];

   g_lpD3D->GetDeviceCaps(iAdapter, adevDevicetypen[iDevice],
                          &pDevice->d3dCaps);

   pDevice->achName    = achDevicetypen[iDevice];
   pDevice->dwAnz_Modi = 0;
   pDevice->dwAkt_Mode = 0;
   pDevice->MultiSampleType = D3DMULTISAMPLE_NONE;
```

wird fortgesetzt...

Wir erzeugen den Pointer `pDevice` als direkte Referenz auf unser globales Array der Adapter und speichern für den Adapter den wir gerade bearbeiten eine Liste aller verfügbaren Devices. Wichtig ist dabei die Funktion `GetDeviceCaps()` welche uns das entsprechende Element der `XD3DDEVICEINFO` Datenstruktur mit den Capabilities (=*Eigenschaften*) des Device füllt. Hier können wir bei Bedarf nachsehen, was das Device alles kann und was nicht.

Als nächstes müssen wir folgendes tun: Für das Device müssen wir das Array `aFormat[]` durchlaufen und alle dort gespeicherten Formate die der Adapter darstellen kann daraufhin prüfen, ob das Device diese auch verwenden kann. Diese Prüfung übernmmt für uns die Funktion `CheckDeviceType()` des Direct3D Hauptinterface Objektes. Wenn diese Funktion fehlschlägt dann können wir das Format verwerfen und brauchen es nicht in die globale Liste aufzunehmen. Anderenfalls setzen wir unsere Arbeit fort und prüfen, ob wir hier ein HAL Device erwischt haben und merken uns dies in der `BOOL` Variablen.
Nun dreht sich alles um die Variable `blnFormat_Okay[]` in der wir speichern werden ob das entsprechende Format aus `aFormat[]` geeignet ist für unser HAL Device oder nicht. Hier gibt es vier Fälle zu unterscheiden:

(a) Unser Device ist ein pures TnL Device
(b) Unser Device ist ein TnL Device
(c) Unser Device macht TnL gemischt in Hard- und Software
(d) Unser Device erledigt TnL in Software

Hier rufen wir auch unsere Testfunktion `xD3DEnum_Device_Okay()` auf wo wir weitere Prüfungen für das Device vornehmen könnten...aber hier nicht tun, die Funktion gibt einfach immer `TRUE` zurück.

xD3D_Erstelle_Deviceliste() Fortsetzug:
(immer noch in der Schleife für Adapter
und noch in der Schleife für Devices)

```
// Untersuche alle Formate des Devices auf Eignung für das
// Device und die Applikation
BOOL      blnFormat_Okay[20];
DWORD     dwEgnschftn[20];
D3DFORMAT fmtDepthStencil[20];

// Format auf Eignung mit Device prüfen
for (DWORD f=0; f<dwAnz_Formate; f++) {
   blnFormat_Okay[f]  = FALSE;
   fmtDepthStencil[f] = D3DFMT_UNKNOWN;

   // Verwerfe Formate die das Device nicht rendern kann
   if (FAILED(g_lpD3D->CheckDeviceType(
                iAdapter,       // Für welchen Adapter
                pDevice->devTyp // HAL od. REF
                aFormat[f],     // Display Format
                aFormat[f],     // BackBuffer Format
                FALSE)))        // Windowed Mode
      continue;

   // Ist das 3D Device ein HAL Device
   if (pDevice->devTyp == D3DDEVTYPE_HAL)
      blnHAL_exists = TRUE;

   // Unterstützt (simuliert) das Device TnL in Hardware?
   if (pDevice->d3dCaps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) {
      // Reines Hardware Vertexprocessing
      if (pDevice->d3dCaps.DevCaps & D3DDEVCAPS_PUREDEVICE) {
```

```
      dwEgnschftn[f] = D3DCREATE_HARDWARE_VERTEXPROCESSING |
                       D3DCREATE_PUREDEVICE;
      // Weitere Tests auf Eignung
      if (SUCCEEDED(xD3DEnum_Device_Okay(&pDevice->d3dCaps,
                       dwEgnschftn[f], aFormat[f])))
         blnFormat_Okay[f] = TRUE;
      } // if [PureDevice]

   // Hardware Vertexprocessing
   if (blnFormat_Okay[f] == FALSE) {
      dwEgnschftn[f] = D3DCREATE_HARDWARE_VERTEXPROCESSING;
      // Weitere Tests auf Eignung
      if (SUCCEEDED(xD3DEnum_Device_Okay(&pDevice->d3dCaps,
                       dwEgnschftn[f], aFormat[f])))
         blnFormat_Okay[f] = TRUE;
      } // if [!PureDevice]

   // Gemischtes Vertexprocessing
   if (blnFormat_Okay[f] == FALSE) {
      dwEgnschftn[f] = D3DCREATE_MIXED_VERTEXPROCESSING;
      // Weitere Tests auf Eignung
      if (SUCCEEDED(xD3DEnum_Device_Okay(&pDevice->d3dCaps,
                       dwEgnschftn[f], aFormat[f])))
         blnFormat_Okay[f] = TRUE;
      } // if [Mixed]
   } // if [TnL (simul)Hw-Device]

// Falls nein bestätige TnL in Software
if (blnFormat_Okay[f] == FALSE) {
   dwEgnschftn[f] = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
   // Weitere Tests aus Eignung
   if (SUCCEEDED(xD3DEnum_Device_Okay(&pDevice->d3dCaps,
                    dwEgnschftn[f], aFormat[f])))
      blnFormat_Okay[f] = TRUE;
   } // if [SW Processing]

// Nun suchen wir geeignetes Z/Stencil Format des Device
if (blnFormat_Okay[f]) {
   if (!FindDepthStencilFormat(iAdapter, pDevice->devTyp,
                           aFormat[f], &fmtDepthStencil[f]))
      blnFormat_Okay[f] = FALSE;
   }
} // for [AnzFormate]
```

wird fortgesetzt...

Okay, nachdem wir also herausgefunden haben, dass wir unser Format für das Device akzeptieren müssen wir noch eine Sache erledigen bevor wir das Format tatsächlich zulassen: Wir müssen prüfen, ob unser Device auch einen DepthBuffer im entsprechenden Format erzeugen kann. DepthBuffer ist dabei nur eine andere Bezeichnung für einen Z-Buffer. Im obigen Codestück prüfen wir das durch die Funktion `FindDepthStencilFormat()` die wir uns später ansehen werden. Im Moment akzeptieren wir einfach mal diese Funktion und ihren Rückgabewert `TRUE` oder `FALSE`.

Are you still with me? Keine Panik wir sind fast da! Jetzt haben wir also herausgefunden welche Formate in dem Array `aFormat[]` für unser Device geeignet sind. Nun durchlaufen wir alle Formate in diesem Array und falls sie geeignet sind so kopieren wir die Daten in die Modusliste des globalen Device Arrays, auf die ja auch der Pointer `pDevice` zeigt, um sie ein für allemal in unserer Liste dingfest zu machen.

xD3D_Erstelle_Deviceliste() Fortsetzug:
(immer noch in der Schleife für Adapter
und noch in der Schleife für Devices)

```
// Wurde das aktuelle Format bestätigt, dann der Liste zufügen
// Für alle Modi des aktuellen Device...
for (DWORD m=0L; m<dwAnz_Modi; m++) {
   // ...für alle Formate des aktuellen Modi der Schleife...
   for (DWORD f=0; f<dwAnz_Formate; f++) {
      // ...Format ist enumeriertes Format des Adapters...
      if (aModus[m].Format == aFormat[f]) {
         // ...und es ist geeignet für unsere Zwecke
         if (blnFormat_Okay[f] == TRUE) {
            // Füge diesen Modus zu der Modi-Liste des Device
            pDevice->aModus[pDevice->dwAnz_Modi].dwBreite =
               aModus[m].Width;
            pDevice->aModus[pDevice->dwAnz_Modi].dwHoehe  =
               aModus[m].Height;
            pDevice->aModus[pDevice->dwAnz_Modi].Format   =
               aModus[m].Format;
            pDevice->aModus[pDevice->dwAnz_Modi].dwEgnschftn =
               dwEgnschftn[f];
            pDevice->aModus[pDevice->dwAnz_Modi].DepthStencilFormat =
               fmtDepthStencil[f];

            // ACHTUNG: Neuer Modus für aktuelles Device
            // gefunden und akzeptiert!
            pDevice->dwAnz_Modi++;

            if (pDevice->devTyp == D3DDEVTYPE_HAL)
               blnHAL_kompatibel = TRUE;
            } // if [Confirmed]
         } // if [Format=Format]
      } // for [AnzFormate]
   } // for [AnzModi]
```

wird fortgesetzt...

Und jetzt haben wir tatsächlich alles zusammen. Wir haben für den Adapter, in dessen Schleife wir uns immer noch befinden, alle verfügbaren Devices enumeriert und für jedes dieser Device, in deren Schleife wir ebenfalls noch sind, haben wir die verfügbaren Bildschirmmodi enumeriert.

Ah...moment, jetzt können wir bereits einen Modi auswählen. Dazu durchlaufen wir also alle enumerierten Modi eines Device und kontrollieren zunächst einmal, ob die Breite und Höhe des Modus den Wünschen entsprechen. Wir erinnern uns ja noch dunkel, dass unsere Wünsche der Funktion als Parameter gegeben wurden. Dann prüfen wir nach, ob wir ein 16 Bit Format für diesen Modus haben. Dafür gibt es drei Möglichkeiten:

(a) `D3DFMT_R5G6B5`: 5 Bit Rot + 6 Bit Grün + 5 Bit Blau
(b) `D3DFMT_X1R5G5B5`: 1 Bit leer + 5 Bit Rot + 5 Bit Grün + 5 Bit Blau
(c) `D3DFMT_A1R5G5B5`: 1 Bit Alpha + 5 Bit Rot + 5 Bit Grün + 5 Bit Blau

Welche der Möglichkeiten wir dabei wählen spielt eigentlich gar keine Rolle, wir nehmen das erst beste 16 Bit Format welches wir in einem Modus finden können. Dann brechen wir die Schleife ab und haben das Element `dwAkt_Mode` der Device Datenstruktur auf die Indexnummer des Arrays `aModus[]` der Datenstruktur gesetzt wo wir den passenden Modus finden.

xD3D_Erstelle_Deviceliste() Fortsetzug:

*(immer noch in der Schleife für Adapter
und noch in der Schleife für Devices)*

```
// Auswahl des gewünschten Modus (nur mit 16 Bit)
for (m=0; m<pDevice->dwAnz_Modi; m++) {
   if (pDevice->aModus[m].dwBreite == dwBreite &&
       pDevice->aModus[m].dwHoehe  == dwHoehe) {
      pDevice->dwAkt_Mode = m;
      // RxGyBz => x Bits für Rot, y für Grün, z für Blau
      // X1=1Bit ungenutzt, A1=1Bit für Alpha
      if (pDevice->aModus[m].Format == D3DFMT_R5G6B5   ||
          pDevice->aModus[m].Format == D3DFMT_X1R5G5B5 ||
          pDevice->aModus[m].Format == D3DFMT_A1R5G5B5   )
         break;
      } // if [Breite&Höhe stimmen]
   } // for [alle Modi des aktuellen Device]
```

wird fortgesetzt...

Jetzt müssen wir ein gefundenes Device endgültig in unserer globalen Liste verewigen. Wie oben beschrieben tun wir dies, in dem wir den globalen Zähler für diese Liste um 1 erhöhen um die aktuell gespeicherten Daten an der Indexposition zu behalten und nicht mehr zu überschreiben. Das tun wir aber nur unter der Voraussetzung, dass wir wenigstens einen gültigen Bildschirmmodus für das Device gefunden haben. Dasselbe machen wir dann mit dem Adapter. Dessen Indexposition im globalen Array wird nur dann gesichert, wenn wir ein gültiges Device (*mit einem gültigen Modus*) für diesen Adapter gefunden und bestätigt haben.

*xD3D_Erstelle_Deviceliste() Fortsetzug:
(immer noch in der Schleife für Adapter
und noch in der Schleife für Devices)*

```
      // Hat Device mind. 1 gültigen Modus dann Device in
      // Device-Liste des aktuellen Adapters speichern
      if (pDevice->dwAnz_Modi > 0) {
         pAdapter->dwAnz_Devices++;
            fprintf(Protokoll, " -> Device: %s\n", pDevice->achName);
         }
      } // for [Schleife der Devices beendet!!!]

   // Hat Adapter mehr als ein gültiges Device dann
   // speichern wir den Adapter in der globalen Liste
   if (pAdapter->dwAnz_Devices > 0) {
      g_dwAnz_Adapter++;
      fprintf(Protokoll, " -> Adapter akzeptiert: %s \n",
         pAdapter->d3dAdapterIdentifier.Description);
      }
   } // for [Schleife der Adapter beendet!!!]
```

Guess what? Wir haben jetzt unsere globale Liste mit allen notwendigen Informationen über sämtliche installierte HAL-fähige Grafikhardware und deren mögliche Bildschirmmodi fertig erstellt. Der finale Schritt unser Monsterfunktion ist es nun, ein geeignetes Device aus dieser Liste auszuwählen und die Indexposition unserer Auswahl zu speichern. Hier also das Ende unserer Funktion:

```
   // Falls wir keinen gültigen Adapter gefunden haben
   // melden wir Fehler. Dann muss die Applikation ein
   // anderes Format wählen (Modus, Fromat, SW etc.)
   if (g_dwAnz_Adapter == 0) {
      fprintf(Protokoll, "Fehler: Kein geeignetes Device \n");
      return FALSE;
      }
```

```
   // Wähle das erst beste Device aus Liste (enthält ja
   // nur geeignete). Das HAL muss vor dem REF stehen.
   for (DWORD a=0; a<g_dwAnz_Adapter; a++) {
      for (DWORD d=0; d<g_Adapter[a].dwAnz_Devices; d++) {
         g_Adapter[a].dwAkt_Device = d;
         g_dwAdapter = a;

         // Warnung ausgeben falls nur REF gefunden
         if (g_Adapter[a].aDevices[d].devTyp == D3DDEVTYPE_REF) {
            if (!blnHAL_exists)
               fprintf(Protokoll, "Fehler: Kein HAL \n");
            else if (!blnHAL_kompatibel)
               fprintf(Protokoll, "Fehler: Unkompatibel \n");
            }

         return TRUE;
         } // for [AnzDevices]
      } // for [AnzAdapter]

   return FALSE;
   } // xD3D_Erstelle_Deviceliste [Funktion zu Ende!!!]
/*----------------------------------------------------*/
```

Nun gibt die globale Variable `g_dwAdapter` an, an welcher Indexposition des globalen Arrays `g_Adapter[]` wir den ausgewählten Adapter finden können. Das Element `dwAkt_Device` unserer Adapter Datenstruktur sagt dann wiederum aus, an welcher Indexposition des Elementes `aDevices[]` dieser Struktur wir das ausgewählte Device finden. Den zu verwendenden Modus haben wir ja bereits weiter oben für jedes Device in der Liste festgelegt.

Woah...fertig enumeriert, jetzt kann der Spass wieder beginnen....oh, da war doch noch was. Wir müssen noch einen Blick auf die Funktion `FindDepthStencilFormat()` werfen.

Der Z Buffer und sein Sinn

Was ist ein Z Buffer (*auch DepthBuffer gennant*)? Stellen wir uns das einmal bildlich vor, wir haben ein 3D Objekt welches wir am Bildschirm rendern wollen. Dazu rechnet der Computer das Objekt um in 2D Koordinaten auf dem Bildschirm und rendert diese indem er alle Pixel auf dem Bildschirm, die das Objekt belegt, mit der entsprechenden Farbe des Objektes an dieser Stelle färbt. Das was passiert wenn wir mehrere Objekte haben?
Unsere 3D Hardware muss irgendwie entscheiden können, welche Objekte andere verdecken. Es wäre ziemlich blöd, wenn man ein Objekt durch eine Mauer hindurch sehen könnte, nur weil wir erst die Mauer rendern und dann das Objekt welches eigentlich hinter der Mauer liegt und durch diese verdeckt werden sollte. Und hier haben wir auch schon eine mögliche Lösung für das Dilemma: Wir können alle unsere Objekte de Entfernung nach sortieren und die am entferntesten Objekte zuerst malen, dann die näheren usw. Das nennt sich Painters Algorithmus weil ein Maler dieselbe Technik verwendet um ein Gemälde zu malen.

Dabei gibt es jedoch einige Probleme auf die ich hier nicht näher eingehen möchte, jedenfalls ist diese Methode nie 100 % korrekt. Und hier kommt der Z Buffer ins Spiel. Statt die Objekte zu ordnen rendern wir sie einfach wild durcheinander. Die 3D Hardware hat aber nun zusätzlich zu dem Front- und BackBuffer noch einen Z-Buffer in derselben Bildschirmabmessung angelegt. Immer wenn ein Pixel auf dem Bildschirm gemalt wird, so berechnet die 3D Engine die Entfernung genau dieses einen Pixels vom Betrachter und speichert diesen Wert an der korrespondierenden Stelle im Z Buffer. Immer wenn nun ein Pixel gemalt werden soll, so wird automatisch zuerst die betroffene Stelle im Z Buffer geprüft. Ist der dort gespeicherte Entfernungswert grösser als der des Pixels der jetzt gemalt werden soll so ist alles okay. Der neue Pixel liegt optisch betrachtet wirklich vor dem alten, wird gemalt und schreibt nun seinen Entfernungswert an diese Stelle in den Z Buffer.

Ist der Entfernungswert des alten Pixels im Z Buffer aber kleiner als der Wert des nun zu malenden Pixels dann haben wir den Fall, dass der bereits gemalte Pixel optisch vor dem neuen Pixel liegt und diesen verdeckt, daher wird der neue Pixel einfach ignoriert und nicht gemalt. Diese Methode funktioniert 100 % perfekt selbst für die verwinkeltsten 3D Modelle. Kommen wir nun zu der Tiefe des Z Buffers. Je mehr Bit ein Z Buffer hat (16, 24, 32) desto genauere Kommazahlen kann der Z Buffer speichern und um so genauer wird er funktionieren. ABER: Je mehr Bit desto mehr Speicherplatz im VRAM belegt der Z Buffer und dieser Speicher ist für gewöhnlich sehr begrenzt. Es empfiehlt sich daher bei einem 16 Bit Z Buffer zu bleiben.

Nun kurz zu dem StencilBuffer. Dieser ist dem Z Buffer ähnlich aufgebaut, wird aber für vollkommen andere Effekte genutzt. Unter anderem kann man damit Schatten in Echtzeit berechnen und rendern lassen und noch diverse andere Dinge. Das geht aber weit über das hinaus was wir uns hier ansehen wollen und daher verzichten wir auf einen StencialBuffer und setzen ihn auf 0 Bit.

Aber zurück zu unserer Auswahlfunktion. Bevor wir einfach blind einen Z Buffer von unserem Device ansehen müssen wir...ja genau, die überhaupt verfügbaren Z Buffer Modi enumerieren.

```
/**
 * Suche Z-/Stecil-Buffer der verfügbar ist und
 * die geforderten Eigenschaften besitzt.
 */
BOOL FindDepthStencilFormat(UINT iAdapter,
                            D3DDEVTYPE DeviceType,
                            D3DFORMAT TargetFormat,
                            D3DFORMAT* pDepthStencilFormat) {
   // 16 Bit Depth-Buffer und 0 Bit Stencil-Buffer
   if ((g_dwMinDepthBits<=16) && (g_dwMinStencilBits==0)) {
      if (SUCCEEDED(g_lpD3D->CheckDeviceFormat(iAdapter, DeviceType,
                                                TargetFormat,
                                                D3DUSAGE_DEPTHSTENCIL,
                                                D3DRTYPE_SURFACE,
                                                D3DFMT_D16))) {
         if (SUCCEEDED(g_lpD3D->CheckDepthStencilMatch(iAdapter,
                                                DeviceType,
                                                TargetFormat,
                                                TargetFormat,
                                                D3DFMT_D16))) {
            *pDepthStencilFormat = D3DFMT_D16;
            return TRUE;
            }
         }
      } // 16 Z Bits und 0 Stencil Bits

   return FALSE;
   } // FindDepthStencilFormat
/*------------------------------------------------*/
```

Ich habe mir hier die künstlerische Freiheit genommen, die Funktion etwas zu kürzen. Im Original (*siehe Download*) werden noch mehrere Fälle abgehandelt, diese sind jedoch alle nahezu identisch mit diesem hier. Lediglich die abgetesteten Bit Tiefen und die Formatkonstante ändern sich.
Hier prüfen wir nur ob das Device die Kombination eines 16 Bit Z Buffers und eines 0 Bit StencilBuffers ausführen kann. Dieses Format hat bei Direct3D die Bezeichnung `D3DFMT_D16`. Die beiden blau markierten Funktionen erledigen diese Tests für uns. Die erste Funktion testet ob das gewünschte Format von dem Device überhaupt unterstützt wird. Die zweite Funktion testen dann, ob wir dieses Format zusammen mit dem gewählten Format des FrontBuffers (*also des Bildschirmmodus*) bei diesem Device verwenden können. Sind diese Tests erfolgreich so füllen wir unseren Call-by-Reference Pointer `pDepthStencilFormat` mit dem entsprechenden Format und geben `TRUE` zurück, anderenfalls `FALSE`.

Initialisieren von Direct3D

Gosh! Gleich haben wir's. Erinnern wir uns zurück an die Funktion `xD3D_initialisieren()`, dort haben wir bereits gesehen dass wir nun, nach der Erstellung der Liste mit aller verfügbaren Hardware, die Funktion `xD3D_Objekte_initialisieren()` aufrufen. Diese dient nun endlich, endlich dazu Direct3D zu starten und zwar mit genau der von uns ausgewählten Hardware. Glaubt es oder nicht, hier wird man sehen, dass die Arbeit mit Direct3D eigentlich ganz leicht ist. Also nehmen wir uns die Funktion Stück für Stück vor:

```
/**
 * Init. des Devices mit einem BackBuffer
 */
BOOL xD3D_Objekte_initialisieren(void) {
   D3DPRESENT_PARAMETERS d3dpp;
   HRESULT               hr;

   // Nimm die Auserwählten aus der globalen Liste
   XD3DADAPTERINFO* pAdapter = &g_Adapter[g_dwAdapter];
   XD3DDEVICEINFO*  pDevice  = &pAdapter->aDevices[pAdapter->dwAkt_Device];
   XD3DMODEINFO*    pMode    = &pDevice->aModus[pDevice->dwAkt_Mode];

   // Lege die Variablen für das Direct3D Device fest
   ZeroMemory(&d3dpp, sizeof(D3DPRESENT_PARAMETERS));
   d3dpp.Windowed               = FALSE;
   d3dpp.BackBufferCount        = 1;
   d3dpp.MultiSampleType        = pDevice->MultiSampleType;
   d3dpp.SwapEffect             = D3DSWAPEFFECT_FLIP;
   d3dpp.EnableAutoDepthStencil = TRUE;
   d3dpp.AutoDepthStencilFormat = pMode->DepthStencilFormat;
   d3dpp.hDeviceWindow          = hwnd;
   d3dpp.BackBufferWidth        = pMode->dwBreite;
   d3dpp.BackBufferHeight       = pMode->dwHoehe;
   d3dpp.BackBufferFormat       = pMode->Format;
   // WICHTIG: Ohne dies Flag können wir BackBuffer nicht ver-
   // riegeln, also nicht selbst darauf malen!!!
   d3dpp.Flags = D3DPRESENTFLAG_LOCKABLE_BACKBUFFER;

   // Erstelle das eigentliche Direct3D Device
   hr = g_lpD3D->CreateDevice(g_dwAdapter, pDevice->devTyp, hwnd,
                              pMode->dwEgnschftn, &d3dpp,
                              &g_lpD3DDevice);
   if (FAILED(hr)) {
      fprintf(Protokoll, "Fehler: CreateDevice() failed \n");
      return FALSE;
      }

   // Globale Variable mit tatsächlichem BackBuffer füllen
   g_lpD3DDevice->GetBackBuffer(0, D3DBACKBUFFER_TYPE_MONO, &g_lpD3DSBack);

   // Licht, Cullmodus und Z-Buffer aktivieren
   g_lpD3DDevice->SetRenderState(D3DRS_AMBIENT,  D3DCOLOR_XRGB(255,255,255));
   g_lpD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
   g_lpD3DDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
   g_lpD3DDevice->SetRenderState(D3DRS_ZENABLE,  D3DZB_TRUE);

   return TRUE;
```

```
   } // xD3D_Objekte_initialisieren
/*----------------------------------------------------------------*/
```

Wow, das ist alles? Ja ist es! Wir wählen zunächst die entsprechende Kombination aus Adapter, Device und Modus welche wir bei der Enumeration als würdig betrachtet haben. Alles was wir dann tun müssen ist die folgende Struktur mit sinnvollen Werten zu füllen:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
   // Bildschirmauflösung und Anzahl an Backbuffern
   UINT      BackBufferWidth;
   UINT      BackBufferHeight;
   D3DFORMAT BackBufferFormat;
   UINT      BackBufferCount;
   // Unwichtig für uns
   D3DMULTISAMPLE_TYPE MultiSampleType;
   // Unwichtig für uns
   D3DSWAPEFFECT SwapEffect;
   // HWND unseres Fensters
   HWND          hDeviceWindow;
   // Fenstermodus (sonst Vollbild)
   BOOL          Windowed;
   // Z Buffer erzeugen?
   BOOL          EnableAutoDepthStencil;
   // Bit Format des Z Buffers
   D3DFORMAT     AutoDepthStencilFormat;
   // BackBuffer durch uns bearbeitbar?
   DWORD         Flags;
   // Unwichtig für uns
   UINT          FullScreen_RefreshRateInHz;
   UINT          FullScreen_PresentationInterval;
   } D3DPRESENT_PARAMETERS;
```

Mit dieser Datenstruktur haben wir dann eine Schablone für Direct3D erstellt, wie unsere Device Objekt aussehen soll. Nun erzuegn wir durch den Aufruf der Funktion `CreateDevice()` des Direct3D Hauptinterface Objektes unser Direct3D Device. Dieser Funktion geben wir an, welchen Adapter wir verwenden wollen, den Device Typ (*HAL oder REF*), ein Handle auf unser Fenster für das das Device ist, den Bildschirmmodus und natürlich die eben gefüllte Datenstruktur. Der letzte Parameter ist dann ein Direct3D Device Interface Objekt welches wir von Direct3D mit dem erzeugten Device füllen lassen. Die globale Variable `g_lpD3DDevice` ist nun ein gültiges Direct3D Device welches die gesamte 3D Arbeit für uns erledigen wird. Insbesondere das Rendern von Dreiecken mit Texturen.
Der erste Schritt ist es nun, dass wir uns über dieses Device Objekt einen weiteren Pointer auf den automatisch erzeugten BackBuffer erstellen. Das brauchen wir wenn wir selbst Daten wie beispielsweise Bilder in den BackBuffer schreiben wollen. Daher musste er auch als verriegelbar definiert werden.

Renderstates des Device

Bei der Arbeit mit einem Direct3D Device verläuft vieles über die sogenannten Renderstates. Wie wir eben gesehen haben setzen wir bereits bei der Initialisierung die wichtigsten dieser Renderstates fest. Aber wir werden im nächsten Kapitel noch etwas genauer sehen, wie die Arbeit mit einem Direct3D Device genau funktioniert. Die Funktion `SetRenderState()` des Direct3D Device Interface legt einen Renderstate auf einen bestimmten Wert fest.

`D3DRS_AMBIENT`
Eine der wichtigsten Sachen in einem 3D Programm ist das Licht! Ambientes Licht nennt man das Licht welches einfach ohne bestimmte Richtung überall mit gleicher Helligkeit vorhanden ist. Das kommt zustanden wenn einfallende Lichtstrahlen der Sonne oder einer Lampe von Wänden, Decke, Tischen und allen anderen

Objekten der realen Welt so oft hin und her reflektiert werden, dass einfach überall ein bisschen Licht ist. Mit diesem Renderstate setzen wir die Farbe und Intensität für unser ambientes Licht fest. Das Makro `D3DCOLOR_XRGB` nimmt Werte von 0 bis 255 für den Rot-, Grün- und Blauanteil des ambienten Lichtes auf. Je kleiner die Zahl um so dunkler die jeweilige Farbe. In diesem Beispiel haben wir also helles weisses Licht.

- `D3DRS_LIGHTING`

Damit Direct3D das Licht auch berechnet müssen wir dem Device explizit mitteilen, dass wir das Licht einschalten wollen. Das machen wir mit diesem Renderstate. Nun wird Direct3D beim Rendern eines jeden Pixels das entsprechende ambiente Licht berücksichtigen.

- `D3DRS_CULLMODE`

Jede Medaille hat zwei Seiten...so auch jedes Dreieck. Wenn wir ein 3D Modell haben, so ist es aber überflüssig beide Seiten der einzelnen Dreiecke des Modells zu berechnen und zu malen, denn die *Innenseite* des Modells kann man ja eh nicht sehen. Wenn wir ein Dreieck aus drei Punkten haben so definieren wir einfach eine der beiden Seiten als Vorderseite und eine der beiden Seiten als Rückseite. In meinem 3D Grafik Grundlagentutorial auf dieser Homepage (*siehe Link im Menü links*) habe ich dazu etwas ausführlichere Erläuterungen gegeben. Kurz gesagt müssen wir einfach immer dieselbe Seite als Vorder- bzw. Rückseite definieren wozu wir den Normalenvektor des Dreiecks verwenden.
Wichtig ist aber folgendes. Die Richtung des Normalenvektors hängt von der Reihenfolge der einzelnen Punkte des Dreiecks ab. Definieren wir die Punkte im Uhrzeigersinn oder dagegen. Entsprechend kann man das Culling von Direct3D einstellen, denn dieses Culling sagt aus, welche der beiden Seiten eines Dreiecks Direct3D malen und welche es weglassen (*cullen*) wird. Der Wert `D3DCULL_NONE` sagt dass Direct3D beide Seiten malen wird was meistens unerwünscht ist. Der Wert `D3DCULL_CW` bedeutet dass Direct3D diejenige Seite des Dreiecks bei der die Punkte in **C**lock**w**ise (*Uhrzeigersinn*) Reihenfolge sind als Rückseite betrachtet und diese nicht malt. Entsprechend besagt `D3DCULL_CCW` dass Seiten mit Punktreihenfolge in **C**ounter**c**lock**w**ise (*Gegenuhrzeigersinn*) weggeculled werden. Auch hier keine Panik, dazu kommen wir im nächsten Kapitel noch einmal.

- `D3DRS_ZENABLE`

Über diesen Renderstate aktivieren wir unseren Z Buffer. Wir haben den Z Buffer zwar über die obige Struktur bereits automatisch erstellen lassen, dennoch müssen wir ihn auch aktivieren. Für gewisse Spezialeffekte kann es nämlich nötig sein, den Z Buffer kurzzeitig zu deaktivieren, und das funktioniert über diesen Renderstate.

Okay, war's das jetzt endlich? Zum Laufen lassen des Programms ja, sonst nein.

Aufräumen nach der Party

Wie immer kommt am Morgen danach irgendwann das böse Erwachen. Wenn wir unser Programm also beenden, dann sollten wir auch ordentlich hinter uns aufräumen. Im Sinne von DirectX heisst das vor allem den blockierten Speicherplatz wieder an Windows zurück zu geben. Bei DirectX Interface Objekten funktioniert das über die Funktion `Release()`.

```
/**
 * Beim Beenden des Spiels muss auch der Speicher
 * der Direct3D Objekte freigegeben werden.
 */
void xD3D_beenden(void) {
   if (g_lpD3DSBack) {
      g_lpD3DSBack->Release();
      g_lpD3DSBack = NULL;
      }
   if (g_lpD3DDevice) {
      g_lpD3DDevice->Release();
      g_lpD3DDevice = NULL;
      g_lpD3D->Release();
```

```
      g_lpD3D = NULL;
      }
   } // xD3D_beenden
/*----------------------------*/
```

Now it's over. Wir haben es tatsächlich geschafft, Direct3D ist jetzt voll und vor allem sauber in unser Projekt integriert. Wir sehen zwar nur einen schwarzen Bildschirm vor sich, aber der hat es faustdick hinter den Ohren...was sein zukünftiges Potential angeht. In der Protokolldatei seines Projektes kann man nach dem Lauf des Programms übrigens nachlesen, welche Hardware aktiviert wurde. Das habe ich in der Funktion `xD3D_Objekte_initialisieren()` noch heimlich untergebracht, Ihr werdet aber keine Probleme haben diese Ausgabe zu verstehen da sie nur bereits hier besprochenes wiederholt.

Und hier gibt es den Code des *gesamten* Tutorials zum Download: Projektdateien

---

Weiter geht's zum Kapitel 4...

SPIELEENTWICKLUNG MIT DIRECT3D IM - KAPITEL 4

---

von Stefan Zerbst

"Du musst lernen wie man Danke sagt und sich dabei ganz tief bückt, und wie man Essensreste von fremden Tellern leckt. Du musst wissen wie man lächelt während man den Boden küsst und dass hier Freiheit nicht viel mehr als ein Wort zum Träumen ist."
König der Blinden, Die Toten Hosen

Ein Dreieck rendern

Na, sitzt uns der Schreck vom letzten Kapitel noch im Nacken? Aber jetzt können wir uns wieder beruhigen, denn nachdem Direct3D erst einmal initialisiert ist wird der Rest *so einfach wie Pfannkuchen essen*... In diesem Kapitel werden wir uns ein wenig damit beschäftigen, wie man eine 3D Welt auf einen 2D Bildschirm bringt. Dazu werden wir alle notwendigen Grundlagen lernen, um das `Direct3DDevice8` Interface Objekt in den Griff zu bekommen und es dazu zu bringe das zu machen was wir von ihm wollen. Und am Ende dieses Kapitels wollen wir ein Dreieck auf den Bildschirm rendern.
Um hier gleich ein wenig Enttäuschung vorzubeugen: Nachdem man es einmal geschafft hat ein 3D Dreieck auf den 2D Monitor zu bringen weiss man eigentlich das wichtigste was es über 3D Engines zu wissen gibt. Selbst wenn ein Dreieck nicht wie ein besonders *cooles* 3D Modell aussieht so macht es doch keinen Unterschied, ob wir nun ein Dreieck rendern oder 500 Dreiecke die dann ein schickes 3D Modell ergeben, oder?!

ACHTUNG: Wir müssen unser Projekt nun zu der Datei `d3dx8.lib` linken, denn diese enthält die DX Funktionen von Direct3D. Diese sind eigentlich nicht ein direkter Bestandteil von DirectX, sondern eine Art Beispiel-Implementierung um die Verwendung von Direct3D einfacher zu machen. Ich werde aber so weit wie möglich auf diese Funktionen verzichten und zeigen, wie wir alles von Hand machen können. Schliesslich sind wir hier um zu lernen, und nicht um die Aufrufe von *Fremd-Funktionen* anzukucken! Ich wede lediglich beim Laden von Texturen auf die DX Funktionen zurückgreifen, da das Laden von BMP Dateien in verschiedenen Auflösungen hier einfach den Rahmen des Tutorials sprengen würde.

Initialisieren der Szene

Okay, wir haben jetzt ein `Direct3DDevice8` Interface Objekt erzeugt und ich hatte bereits angedeutet, dass dieses eine grosse Last der zu erledigenden Arbeit tragen wird. Damit unsere Ausgabe auf dem Bildschirm auch irgendwie sinnvoll aussehen wird müssen wir zunächst noch einen Schritt weiter gehen und das Device auf unsere konkreten Wünsche hin ausrichten. Allgemein bezeichnet man seine 3D Umgebung, also das gesamte Programm, als Szene. Daher benötigen wir nun eine Funktion die unsere Szene initialisiert. Eigentlich tun wir dort aber nix weiter als das Device Objekt konkret einzustellen.

Das wichtigste hierbei ist nun die Frage, wie bekommen wir unsere Daten von 3D auf 2D? Im 3D Grafik Grundlagen Tutorial (*siehe Link im Menü links*) habe ich das auch etwas ausführlicher beschrieben, hier also noch einmal eine Kurzfassung:
Der Vorgang nennt sich **Projektion**, wir projizieren unsere Daten von einem 3D Koordinatensystem in ein 2D Koordinatensystem. Im 3D Raum haben alle Objekte die drei Koordinaten (x,y,z) die ihre Position auf den drei Achsen angeben. Im 2D Raum, wie beispielsweise dem Bildschirm, haben alle Objekte aber nur die zwei Koordinaten (x,y) und das z fehlt. Stop! Das ist nicht so ganz richtig, denn wenn wir 3D Szenen auf 2D projizieren so sehen sie immer noch 3D aus. Jeder kennt sicherlich die Techniken um solche Bilder auf dem Papier zu malen. Entweder malen wir alle Linien *die nach hinten gehen* im 45 Grad Winkel und um die Hälfte verkürzt oder wir malen einen Punkt auf das Blatt auf den alle diese Linien zulaufen.
Die z Koordinate geht also nicht wirklich verloren, sondern sie wird bei dem Vorgang der Projektion in die x und y Koordinaten integriert. Wenn wir unsere Szene jetzt für Direct3D initialisieren dann müssen wir dem Device

mitteilen, wie es diese Projektion durchführen soll. Dazu müssen wir dem Device eine **Projektionsmatrix** bekannt machen und das geht wie folgt:

```
g_lpD3DDevice->SetTransform(D3DTS_PROJECTION, &matProj);
```

Die Funktion `SetTransform()` des `Direct3DDevice8` Interface übernimmt diesen Job für uns. Wie wir später noch sehen werden gibt es drei verschiedene Arte von Transformationen die wir für das Device einstellen können, die Projektion ist nur eine davon. Im ersten Parameter der Funktion geben wir daher den Bezeichner `D3DTS_PROJECTION` an, damit das Device *weiss* dass wir ihm nun eine Projektionsmatrix bekannt machen werden. Der zweite Parameter enthält dann die Projektionsmatrix vom Typ `D3DMATRIX` die folgende Elemente enthält:

```
struct {
   float _11, _12, _13, _14;
   float _21, _22, _23, _24;
   float _31, _32, _33, _34;
   float _41, _42, _43, _44;
   };
```

Wem Matrizen bisher noch nie begegnet sind, obwohl das in der Schule früher oder später der Fall sein sollte, der braucht sich nun nicht so viele Gedanken darüber zu machen. Eine Matrix ist sozusagen eine Tabelle aus Zahlen mit einer gewissen Anzahl aus Zeilen und Spalten. Wie man oben sieht werden diese durchnummeriert mit zweistelligen Zahlen. Die erste Zahl gibt die Zeilennummer an und die zweite Zahl die Spaltennummer. Bevor wir uns weiter über Matrizen den Kopf zerbrechen kommt hier noch eine ganz besondere Matrix die man kennen sollte:

```
1   0   0   0
0   1   0   0
0   0   1   0
0   0   0   1
```

Diese Matrix nennt sich Einheitsmatrix, denn für sie ist charakteristisch dass alle Einträge ausser der Hauptdiagonalen 0 sind während die Hauptdiagonale nur Einträge mit 1 hat. Bei der Arbeit mit Direct3D haben alle Matrizen übrigens genau vier Zeilen und vier Spalten, was auch seinen Grund hat wie wir im nächsten und übernächsten Kapitel einsehen werden.

Aber zurück zu unserer Szene und der Projektionsmatrix. Wir wissen jetzt was eine Matrix ist und wie wir eine solche zur Verwendung für die Projektion bei Direct3D anmelden können. Was wir noch nicht wissen ist, wie erstellen wir so eine Projektionsmatrix? Die Antwort ist denkbar einfach, wir nehmen die folgende Funktion die diese Arbeit für uns erledigt:

```
BOOL xD3D_Projektions_Matrix(D3DMATRIX *mat,  // Speicheradresse
                             float fFOV,      // Sichtfeld
                             float fAspect,   // Bildschirmratio
                             float fNearPlane,// Sichtweite von
                             float fFarPlane);// Sichtweite bis
```

Erst mal keine Panik, wir werden diese Funktion ein paar Abschnitte weiter unten auch selbst implementieren und dann genau sehen was dort passiert. Hier sind wir aber erst bei der Initialisierung unserer Szene die ich erst mal beenden möchte. Wie wir sehen übergeben wir unserer Funktion diverse Parameter, allen voran die Speicheradresse `mat` an der die Funktion die fertige Projektionsmatrix speichern wird. Der zweite Parameter gibt das gewünschte Sichtfeld des Spielers an. Wenn der Spieler auf den Monitor blickt und die 3D Welt sieht dann kann man sich das so vorstellen als ob ein Mensch durch ein Fenster sieht. Für den Parameter `fFOV` geben wir dann den Sichtwinkel an, den der Mensch beim Blick aus dem Fenster hat. Das hängt natürlich von der Entfernung zum Fenster ab, aber ein Winkel von ca. 45 Grad ist ein ganz guter Durchschnittswert. Aber Vorsicht, bei der Arbeit mit Direct3D rechnet man mit Winkel in der Einheit RAD und nicht Grad. Aber wenn man den RAD Winkel mit 57 multipliziert erhält man den (*ungefähren*) Wert in Grad. Wir werden für `fFOV` also 0.8 RAD verwenden weil `0.8*57 = ca.45` ist.

Als nächstes müssen wir der Funktion die Aspektratio unseres Bildschirms mitteilen. Das klingt zwar hochtechnisch, ist aber nix anderes als der Verhältnis von `Höhe / Breite` des Bildschirms, aber dazu gleich mehr. Die letzten zwei Parameter geben schliesslich die Sichtweite des Spielers an und zwar einmal in welcher Entfernung vom Spieler die Sichtweite beginnt und in welcher Entfernung sie endet. Unser Funktionsaufruf wird also wie folgt aussehen:

```
D3DMATRIX matProj;

xD3D_Projektions_Matrix(&matProj, //  Speicheradresse
                        0.8f,     //  Sichtfeld
                        fAspekt,  //  Bildschirmratio
                        1.0f,     // Sichtweite  von
                        1000.0f); // Sichtweite bis
```

Hey...da hat sich die Variable `fAspekt` noch reingeschummelt, woher kommt die denn? Also, eben hatte ich zwar gesagt dass wir als Ratio das Verhältnis von Bildschirmhöhe zu Bildschirmbreite verwenden, das ist aber nicht ganz korrekt. Selbst wenn wir ein Vollbild Programm entwickelt so kann es unter Umständen gewünscht sein, dass die 3D Ansicht eben *nicht* über den ganzen Bildschirm geht, sondern dass irgendwo rechts, links, oben oder unten noch ein Menü ist oder Anzeigeinstrumente oder was auch immer. Man unterschiedet daher strikt zwischen dem Bildschirm und dem sogenannten **Viewport**. Der Viewport ist der Bereich des Bildschirms auf dem die 3D Grafik angezeigt wird. In unserem Fall hat der Viewport dieselbe Abmessung wie der Bildschirm, da wir die 3D Ausgabe auf dem ganzen Bildschirm sehen wollen. In Direct3D gibt es für den Viewport die folgende Datenstruktur:

```
typedef struct _D3DVIEWPORT8 {
   DWORD X;       // Rechter Rand
   DWORD Y;       // Oberer Rand
   DWORD Width;   // Breite
   DWORD Height;  // Höhe
   float MinZ;    // Z Min Wert
   float MaxZ;    // Z Max Wert
   } D3DVIEWPORT8;
```

Wie man sieht geben die ersten beiden Elemente die obere, linke Ecke des Viewports an, bei uns ist das (0,0) denn bei der Erzeugung des Device Objektes hat Direct3D automatisch einen Viewport in der Grösse und Position des Fensters (*hier Vollbild*) erzeugt. Die beiden folgenden Elemente geben die Breite und Höhe an, also bei uns 800, 600. Die beiden letzten Elemente betreffen den Z Buffer, denn dieser Z Buffer gehört ja schliesslich zum Viewport. Hier kann man angeben, welche Werte im Z Buffer als Minimalwert und welche als Maximalwert zugelassen sind. Für gewöhnlich verwendet man die Werte 0.0f und 1.0f so dass alle Werte im Z Buffer als Kommazahlen zwischen 0 und 1 liegen.
Uns interessiert nun aber lediglich die Höhe und die Breite um `fAspekt` zu berechnen. Daher bemühen wir die entsprechende Funktion des Device um uns eine `D3DVIEWPORT8` Struktur mit den Daten des aktiven Viewports füllen zu lassen:

ACHTUNG: Die Funktion IDirect3DDevice8::GetViewport ist anscheinend auf fast keiner Grafikkarte (*ausser natürlich mal wieder meiner*) so implementiert, dass sie fehlerfrei funktioniert. Daher sollte man einen entsprechenden Fall auch vorsehen!!!

```
D3DVIEWPORT8 d3dViewport;
float        fAspekt

d3drval = g_lpD3DDevice->GetViewport(&d3dViewport);
if (FAILED(d3drval))
   fAspekt = ((float)SCREEN_HOEHE) / SCREEN_BREITE;
else
   fAspekt = ((float)d3dViewport.Height) / d3dViewport.Width;
```

Nun gut, unser Device ist jetzt eigentlich bereit 3D Grafik zu rendern. Aber wir werden uns noch eines ansehen, bevor wir dazu kommen. Geht auch schnell, versprochen.

```
typedef struct _D3DMATERIAL8 {
   D3DCOLORVALUE Diffuse;
   D3DCOLORVALUE Ambient;
   D3DCOLORVALUE Specular;
   D3DCOLORVALUE Emissive;
   float         Power;
   } D3DMATERIAL8;
```

Direct3D definiert die obige Struktur, um sogenannte Materialien festzulegen. Ein Material definiert das Erscheinungsbild einer Oberfläche eines Objektes. Es gibt glatte Materialien wie beispielsweise metallische Oberflächen und es gibt rauhe Materialien wie Holz oder so etwas. Das Erscheinungsbild diese Oberflächen hängt massgeblich damit zusammen, wie sie das einfallende Licht reflektieren. Wir hatten ja bereits das ambiente Licht besprochen. Daneben gibt es noch das diffuse Licht, spekuläres Licht und das Licht welches ein Objekt selbst emittieren kann (*wenn es glüht oder leuchtet*). In der obigen Struktur können wir diese Lichtwerte in den entsprechenden `D3DCOLORVALUE` Objekten einstellen, welche jeweils die `float` Elemente A R G B (*Alpha, Rot, Grün, Blau*) haben. Ich möchte jetzt hier nicht weiter in die verschiedenen Arten von Lichttypen abdriften, das liegt auch ausserhalb des Scopes dieses Tutorials. Aber das DirectX SDK bietet in der Doku einen Abschnitt über die entsprechenden Direct3D Lichttypen.
Wir werden jetzt einfach ein Materialobjekt erstellen und dem Device zuweisen. Dieses Material wird dann für alle Objekte verwendet die von dem Device gerendert werden. So lange bis wir ein anderes Material für das Device festlegen. Der Code dazu ist selbsterklärend, daher kommt hier gleich die gesamte Funktion zur Initialisierung unserer Szene:

```
/**
 * Initialisieren der Scene
 */
BOOL xD3D_Szene_initialisieren(void) {
   HRESULT d3drval;

   // 1.Schritt: Viewport Daten abfragen
   D3DVIEWPORT8 d3dViewport;
   d3drval = g_lpD3DDevice->GetViewport(&d3dViewport);
   if(FAILED(d3drval)) {
      fprintf(Protokoll, "Fehler: GetViewport() failed \n");
      return FALSE;
      }

   // 2.Schritt: Perspektivische Projektionsmatrix festlegen
   D3DMATRIX matProj;
   float fAspekt = ((float)d3dViewport.Height) / d3dViewport.Width;

   xD3D_Projektions_Matrix(&matProj, // Speicheradresse
                           0.8f,     // Sichtfeld
                           fAspekt,  // Bildschirmratio
                           1.0f,     // Sichtweite von
                           1000.0f); // Sichtweite bis

   // Projektionsmatrix einstellen
   g_lpD3DDevice->SetTransform(D3DTS_PROJECTION, &matProj);

   // 3.Schritt: Rasterizations Material bestimmen
   D3DMATERIAL8 mtrl;
   ZeroMemory(&mtrl, sizeof(mtrl));
```

```
   mtrl.Ambient.r  = 1.0f; // 100% R und G reflektieren
   mtrl.Ambient.g  = 1.0f; // ->Objekt strahlt Gelb ab
   mtrl.Ambient.b  = 0.0f;
   mtrl.Emissive.r = 1.0f; // Objekt erzeugt rotes Licht
   mtrl.Emissive.g = 0.0f;
   mtrl.Emissive.b = 0.0f; // ->Gelb+Rot = Orange

   g_lpD3DDevice->SetMaterial(&mtrl);

   // 4.Schritt: Geometrie erstellen
   if (!xD3D_Dreieck_erzeugen()) {
      fprintf(Protokoll, "Fehler: Dreieck_erzeugen() failed \n");
      return FALSE;
      }

   fprintf(Protokoll, "Szene_init: successful \n");
   return TRUE;
   } // xD3D_Szene_initialisieren
/*------------------------------------------------------------*/
```

Der letzte Schritt in der Initialisierung unserer Szene ist dann die Erzeugung der Geometrie für unser Dreieck. Das klingt zwar sehr spannend, ist aber nicht mehr als die Festlegung von drei 3D Punkten im Raum die die Ecken des Dreiecks angeben werden. Obwohl...es bisschen mehr steckt schon dahinter. Sehen wir uns das also mal genauer an.

Anlegen der Geometrie

Wir brauchen eigentlich nicht viel mehr als eine Datenstruktur für die einzelnen Punkte eines 3D Modells, sei es beispielsweise ein einfaches Dreieck. Da wir die gesamte Arbeit des Renders von 3D Geometrie aber später auf dem Direct3D Device abladen müssen wir uns hier etwas kooperativ zeigen. Direct3D definiert bestimmte Elemente die wir in unsere Datenstruktur einbinden müssen, beispielsweise die drei Koordinaten eines Punktes. Unsere Datenstruktur sieht nun also wie folgt aus:

```
// Vertex Struktur (untransformiert, unbeleuchtet)
typedef struct D3DVERTEX_TYPE {
   float     x;  // D3DFVF_XYZ
   float     y;
   float     z;
   D3DVECTOR vN; // D3DFVF_NORMAL
   float     tu; // D3DFVF_TEX1
   float     tv;
   } D3DVERTEX;
```

Als erstes sehen wir hier dass die Eckpunkte im Englischen Vertex (*Mrz: Vertices*) heissen. Diese Bezeichnung sollten wir auch verwenden, da wir dann einige Funktionen oder Bezeichner von Direct3D besser verstehen können, ebenso wie die englisch Doku oder andere Tutorials. Als zweites erkennen wir die Direct3D Definitionen die ich auskommentiert hinter die Elemente geschrieben habe. `D3DFVF_XYZ` steht beispielsweise für **D**irect**3D** **F**lexible**V**ertex**F**ormat **XYZ** Coordinates und besagt dass wir drei float Werte definieren müssen die die x, y und z Koordinate (*in dieser Reihenfolge*) des Vertex angeben. Das nächste Element ist der Normalenvektor des Vertex, doch dazu später mehr. Die letzten zwei Elemente sind die Texturkoordinaten des Vertex, auch dazu später mehr.
Nun müssen wir uns noch eine Definition erstellen aus der Direct3D schlau werden kann, wie unsere Datenstruktur genau aussieht, bzw. was für Elemente sie enthält.

```
#define D3DFVF_VERTEX (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
```

Auf diese Weise erstellen wir uns sozusagen den Bezeichner `D3DFVF_VERTEX` den wir dem Device vor dem Rendern von `D3DVERTEX` Vertices angeben müssen. Damit weiss das Device dann in welcher Reihenfolge was für Elemente in unserer Vertex Struktur definiert sind und kann diese Daten dann zum Rendern der Vertices an der richtigen Position, mit der richtigen Beleuchtung und den richtigen Texturfarben rendern.

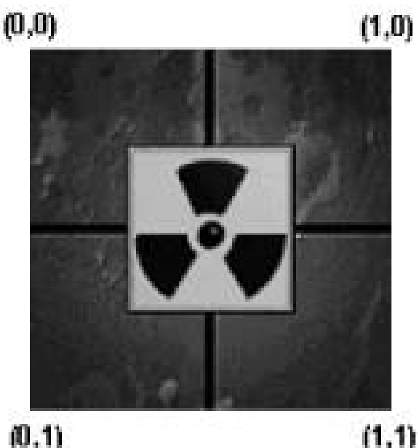
Der Datentyp `D3DVECTOR` von Direct3D dient dazu, einen Vektor zu definieren. Ein Vektor (*für alle die das in Mathe auch noch nicht hatten*) ist sozusagen ein Stück einer Linie mit einer bestimmten Länge und einer Ausrichtung im 3D Raum. Man definiert dazu einfach nur x, y und z Koordinaten eines Punktes und der Vektor ist das genau das Linienstück welches vom Nullpunkt des Koordinatensystems zu diesem Punkt läuft. Den Normalenvektor eines Vertex brauchen wir, um die Beleuchtung des Punktes zu berechnen. Glücklicherweise ist das aber auch der Job von Direct3D und wir brauchen uns darum nicht zu kümmern. Als Normalenvektor bezeichnet man einen Vektor übrigens, wenn er im 90 Grad Winkel zu einer Fläche steht.
Hä...wie kann ein Vektor im 90 Grad Winkel zu einem unendlich kleinen Vertex stehen? Natürlich gar nicht, bei der Benennung und Berechnung dieses Vektors schummelt man etwas. Zur Berechnung der Beleuchtung wird Direct3D für jeden Vertex einen Intensitätswert des einfallenden Lichtes berechnen. Wenn zwischen drei oder mehr Vertices dann eine Fläche am Bildschirm gerendert wird so verwendet Direct3D die Lichtwerte jedes einzelnen Vertex und interpoliert diese um die Fläche entsprechend mit einer fliessenden Beleuchtung zwischen den berechneten Vertexlichtwerten zu versehen. Den Normalenvektor eines Vertex berechnet man dann einfach, indem man den Durchschnittsvektor aus den Normalenvektoren aller Flächen berechnet die an den Vertex angrenzen.
Puh...viel Theoriekrams hier. Keine Panik, wir werden das nicht weiter vertiefen oder implementieren. Aber ich dachte mir es ist gut wenigstens die Grundbegriffe zu kennen. Nun aber zurück zu unserem Dreieck:

```
D3DVERTEX          g_avTri[3];
LPDIRECT3DTEXTURE8 g_lpTextur;

/**
 * Die Geometrie für das globale Dreieck erstellen
 */
BOOL xD3D_Dreieck_erzeugen(void) {
   HRESULT   hr;
   D3DVECTOR vNormal;
   D3DVERTEX v1, v2, v3;

   vNormal.x =  0.0f;
   vNormal.y =  0.0f;
   vNormal.z = -1.0f;

   v1.x  = 0.0f;  // Mitte
   v1.y  = 1.0f;  // Oben
   v1.z  = 3.0f;
   v1.vN = vNormal;
   v1.tu = 0.5f;
   v1.tv = 0.0f;

   v2.x  = 1.0f;  // Rechts
   v2.y  = -1.0f; // Unten
   v2.z  = 3.0f;
   v2.vN = vNormal;
   v2.tu = 1.0f;
   v2.tv = 1.0f;

   v3.x  = -1.0f; // Links
   v3.y  = -1.0f; // Unten
   v3.z  = 3.0f;
   v3.vN = vNormal;
   v3.tu = 0.0f;
```

```
   v3.tv = 1.0f;

   g_avTri[0] = v1;
   g_avTri[1] = v2;
   g_avTri[2] = v3;

   hr = D3DXCreateTextureFromFileA(g_lpD3DDevice, // Direct3D Device
                                   "textur.bmp",  // Grafikdatei
                                   &g_lpTextur);  // Textur Objekt
   if (FAILED(hr)) {
      fprintf(Protokoll, "Fehler: CreateTexture() failed \n");
      return FALSE;
      }

   return TRUE;
   } // xD3D_Dreieck_erzeugen
/*----------------------------------------------------------*/
```

Das globale Array `g_avTri[3]` enthält die drei Vertices die später mal unser Dreieck ergeben sollen. Das Objekt `LPDIRECT3DTEXTURE8` ist der Direct3D Datentyp in den wir eine Grafikdatei laden können die wir als Textur verwenden wollen. Also die Grafik die beim Rendern auf das Dreieck gelegt wird um es schöner aussehen zu lassen. Zum Laden der Grafik verwenden wir die Funktion `D3DXCreateTextureFromFileA()` aus der `d3dx8.lib` Bibliothek.
Nun zur Erstellung der eigentlichen Geometrie. Die z Koordinate aller drei Punkte ist dieselbe, nämlich 3. Die z Achse von Direct3D verläuft in positiver Richtung vom Spieler weg und in negativer Richtung direkt hinter den Spieler. Die z Koordinate +3 liegt also in drei Einheiten Entfernung vor dem Spieler, welcher am Punkt (x=0, y=0, z=0) des 3D Universums steht. Die anderen Werte der Vertices sind dann entsprechend ausgelegt, so dass das Dreieck direkt in der Mitte das Bildschirms erscheinen wird. Ihr könnt aber gerne noch mit den Koordinaten herumspielen, um die Auswirkungen von anderen Werten zu sehen. Nun noch einmal zur Vorderseite und Rückseite des Dreiecks und dem Culling von Rückseiten. Wie man sieht ist das Dreieck in der Reihenfolge v1, v2, v3 definiert, also die Punkte oben, rechts und unten. Damit sind die Punkte im Uhrzeigersinn von (*von unserer Seite des Dreiecks aus gesehen*) die Vorderseite des Dreiecks und werden gerendert. Würden wir nun auf der andern Seite des Dreiecks stehen dann würden wir die Punkte v1, v2, v3 als oben, links und rechst interpretieren (*also natürlich spiegelverkehrt*) und sie hätten die Reihenfolge gegen den Uhrzeigersinn. Daher wird diese Seite des Dreiecks nicht gerendert. Das hat jetzt nichts damit zu tun, dass wir diese Seite gar nicht physisch sehen können. Aber wenn wir das Dreieck im nächsten Kapitel rotieren, dann werden wir sehen dass es tatsächlich nur eine Seite hat und die andere unsichtbar ist, auch wenn sie dann zu uns zeigt.

Ach ja, die Texturkoordinaten, gleich nach der Enumeration eines meiner Lieblingsthemen. Ist aber eigentlich gar nicht so schwer, wenn man es einmal verstanden hat. Ein Bild sagt aber immer mehr als tausend Worte:



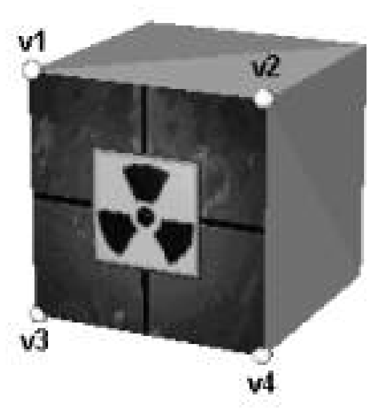


Abbildung 1: Berechnung der Texturkoordinaten

Auf der linken Seite sehen wir ein BMP Bild welches wir als Textur geladen haben. In diesem Kontext sprechen wir dann von den sogenannten Texelkoordinaten tx und ty auf diesem 2D Bild. Wie man sieht sind diese immer so definiert dass sie zwischen 0.0f und 1.0f laufen, egal ob das Bild nun 32x32 oder 256x64 Pixel gross ist. Wenn wir nun ein 3D Modell, so wie hier den Würfel, rendern so müssen wir Direct3D für jeden Vertex die Texturkoordinaten namens u und v mitteilen. Um die Sache etwas einfacher zu machen haben wir hier nur eine Seite des Würfels mit Textur belegt, nämlich die Seite aus den Vertices v1, v2, v3, v4. Um die Textur wie gezeigt auf die Seite zu malen müssen wir nichts weiter tun, als die entsprechenden Texelkoordinaten auszulesen. Der Vertex v1 hat also die Texturkoordinaten (0.0, 0.0), v2 hat (1.0, 0.0) und so weiter.
Nun haben wir aber dummerweise ein Dreieck und kein Viereck! Wie man sieht haben wir die Texturkoordinaten der beiden Fussvertices unseres Dreiecks so festgelegt wie hier die Vertices v3 und v4. Wir haben aber das Problem mit der Spitze des Dreiecks, diese müssten wir ja Textur.technisch betrachtet irgendwo in der Mitte zwischen den Texturkoordinaten der Vertices v1 und v2 des Würfels anordnen. Mit den Werten von (0.0, 0.0) wäre die Texturgrafik von rechts bis in die Mitte des Dreiecks verzehrt und bei (1.0, 0.0) wäre sie nach links verzehrt, ihr könnt das gerne ausprobieren! Und es ist dann tatsächlich so einfach wie man es sich nur vorstellen kann: Die Spitze des Dreiecks liegt doch genau in der Mitte, also sind die Texturkoordinaten der Spitze (0.5, 0.0). Damit wird die Grafik sauber über das Dreieck gemalt und es fehlen halt nur die oberen Ecken weil die Grafik ja ein Viereck ist.

Das ist aber noch nicht die ganze Wahrheit: Man kann eine Textur beispielsweise auf einer Fläche wiederholen. Wir können die Textur von 0 mal über 1/2 mal über 2, 3, 4 oder 5 mal und so weiter auf der Fläche anzeigen lassen. Entweder in horizontal (u) wiederholt oder vertikal (v) oder gar in beide Richtungen mit gleichen oder verschiedenen Weiderholungswerten. Dasselbe gilt für eine Verschiebung der Textur. Beispielsweise könnten wir in obigem Bild die Wiederholung und Verschiebung der Textur so einstellen, dass wir nur das helle Rechteck mit dem Radioaktiv-Symbol auf der Fläche sehen können. Dazu müssten wir die Textur ungefähr 0.5 mal wiederholen und um ca 0.5 verschieben. Beides sowohl in u als auch in v Richtung. Doch wie drücken wir das in Texturkoordinaten aus? Ganz einfach, so:

```
u = Verschiebung_x + Basis_u * Wiederholung_u;
v = Verschiebung_y + Basis_v * Wiederholung_v;
```

Der entsprechende Basiswert bezieht sich auf den ursprünglichen Wert zwischen 0.0 und 1.0 den die Texturkoordinate ohne Verschiebung und Wiederholung hat. Auch damit könnt Ihr ja ein wenig rumspielen. damit haben wir also unsere Geometrie für das Dreieck erstellt und als kleine Übung solltet Ihr wirklich versuchen ein Rechteck zu erstellen, mit Textur zu versehen und diese dann zu verschieben und zu wiederholen. Das ist gar nicht so schwer, denn ein Rechteck rendert Ihr aus zwei Dreiecken.
Man muss also die Koordinaten des Spitze des Dreiecks zu (-1, 1, 3) verschieben und einen Punkt v4 bei (1, 1, 3) erzeugen. Die Texturkoordinaten könnt Ihr dann ganz leicht aus dem obigen Beispiel mit dem Würfel ersehen. Das Array `g_avTri[6]` enthält dann drei Einträge mehr und zwar v1, v4, v3 damit die Vertices auch

brav im Uhrzeigersinn bleiben. Beim Rendern (*siehe unten*) muss man dann noch angeben, dass man zwei und nicht mehr nur ein Dreieck rendern will. Okay, auch wenn's jetzt wie in der Schule klingt aber das solltet Ihr wirklich als kleine Hausaufgabe machen, denn: *Selber coden macht schlau!*

Rendern eines Dreiecks mit Textur

Und nun lüften wir das magische Geheimnis um das sich Tausende von Fragezeichen ranken: Wie male ich ein Dreieck mit Direct3D? Ich traue mich gar nicht dass hier zu zeigen, denn es ist so lächerlich simpel dass man sich frag, warum man nicht schon seit frühester Kindheit 3D Engines entwickelt ;)

```
/**
 * Ausgabe eines Rechtecks mit Textur als Testobjekt
 */
BOOL xD3D_Dreieck_mit_Textur_rendern(void) {
   // Selbstdefiniertes Vertexformat bekannt machen
   g_lpD3DDevice->SetVertexShader(D3DFVF_VERTEX);

   // Textur für das Device festlegen
   g_lpD3DDevice->SetTexture(0, g_lpTextur);

   // Szene starten
   g_lpD3DDevice->BeginScene();
      // Vertices rendern
      g_lpD3DDevice->DrawPrimitiveUP(
                D3DPT_TRIANGLELIST, // Echte Dreiecke
                1,                  // 1 Dreieck zum Rendern
                &g_avTri[0],        // Anfang der Daten
                sizeof(D3DVERTEX)); // Grösse eines Vertex
   g_lpD3DDevice->EndScene();

   return TRUE;
   } // xD3D_Dreieck_mit_Textur_rendern
/*-----------------------------------------------*/
```

Als erstes verwenden wir die Funktion `SetVertexShader()` des Device Objektes um Direct3D mitzuteilen dass wir nun unser selbstdefiniertes Vertexformat verwenden werden. Damit weiss das Device, aus welchen Daten in welcher Reihenfolge sich unser Vertex zusammensetzt.
Danach teilen wir dem Device unsere Textur durch die Funktion `SetTexture()` mit. Ebenso wie bei einem Material gilt hier, dass ab jetzt alle 3D Grafik, die wir über das Device rendern, mit dieser Textur versehen werden so lange bis wir ein anderes `LPDIRECT3DTEXTURE8` Objekt oder `NULL` festlegen. Der erste Parameter der Funktion bezieht sich übrigens auf die Textur Stufe des Device. Es stehen aktuell 8 Stufen zur Verfügung die für verschiedene Spezialeffekte genutzt werden können, wenn man mehrere Texturen gleichzeitig auf ein Objekt legen will. So weit sind wir aber noch lange nicht, daher verwenden wir nur die 0. Stufe welche die *normale* Textur beinhaltet.

Ach ja, in diesem Beispielcode würde es natürlich ausreichen, wenn wir Textur und VertexShader einmal bei der Initialisierung setzen und nicht immer wieder beim Rendern. Wenn wir es aber irgendwann mal mit mehr als einer Textur und/oder mehr als einem VertexShader zu tun haben so müssen wir in jeder entsprechenden Renderfunktion die richtigen Objekte für das Device einstellen, darum fangen wir hier gleich richtig damit an!

Und jetzt haben wir alles zusammen. Durch die Funktion `BeginScene()` teilen wir dem Device mit, dass wir jetzt endlich etwas rendern wollen und nach dem Rendern rufen wir dann entsprechend `EndScene()` auf, um dem Device mitzuteilen dass es sich jetzt wieder entspannen kann. Ach ja...das Stück dazwischen. Eigentlich erklärt sich alles von selbst, die Funktion `DrawPrimitiveUP()` dient dazu, grafische Primitive zu malen (*wobei das Englische Primitive nix mit dem deutschen Adjektiv primitiv zu tun hat*).
Direct3D ist dazu in der Lage, verschiedene Typen von Primitiven zu malen, beispielsweise Punkte, Linien und

eben Dreiecke. Dazu gibt es noch Unterscheidungen zwischen beispielsweise Dreieckslisten oder Dreieck-Fächern. Aber wir bleiben hier auf dem Teppich bei dem Bezeichner D3DPT_TRIANGLELIST, welcher aussagt dass wir einzelne Dreiecke aus jeweils drei Vertices unseres Arrays rendern wollen. Als zweites geben wir an, wie viele Dreiecke wir rendern wollen (*hier kommt für die Hausaufgabe eine 2 hin*) und danach geben wir die Stelle an, wo unsere Vertexdaten beginnen. Als letztes muss Direct3D noch wissen, wie gross unsere Datenstruktur ist. Das ist alles, nun wird unser Dreieck gerendert...damit ist es aber noch nicht am Bildschirm sichtbar!

Wir erinnern uns ja noch dunkel, dass wir einen BackBuffer haben. In genau diesem ist das Dreieck jetzt gerendert und wir müssen nun dafür sorgen, dass Front- und BackBuffer gewechselt werden. Das ist aber eine Arbeit die wir in der Hauptschleife erledigen werden.

Aufrufen des Renderns in der Hauptschleife

Zugegebenermassen ist unsere 3D Szene bisher noch recht trostlos. Wir haben lediglich ein Dreieck, welches auch noch nicht einmal sehr 3D'ig aussieht, eher platt 2D. Nichtsdestotrotz handelt es sich bei unserem Programm bereits um ein vollwertiges 3D Programm und wir beginnen damit, unser Projekt auch auf bewegte 3D Grafik einzurichten. Beispielsweise die Rotation des Dreiecks die wir im nächsten Kapitel in Angriff nehmen werden.

Dazu gehört zuerst, dass wir in jedem Frame den BackBuffer löschen müssen. Warum ist das nötig? Nun, wir malen lediglich ein Dreieck in den BackBuffer und wechseln diesen dann zum FrontBuffer. Dann malen wir in den jetzigen BackBuffer wieder das Dreieck...auch kein Problem...und wechseln wieder die Buffer für den nächsten Frame. Nun haben wir aber wieder den Buffer, den wir zuerst als BackBuffer hatten und in diesem ist noch das alte Bild aus dem ersten Frame gespeichert. Wenn wir das Dreieck aber nun an einer anderen Stelle malen, dann sehen wir noch die Reste des alten Bildes. Wir müssen also den Buffer in jedem Frame löschen um den neuen Frame ordentlich malen zu können. Wie gesagt in unserem Beispiel hier ist das eigentlich noch unnötig, da wir das Dreieck immer exakt an dieselbe Stelle in den Buffern rendern. Aber spätestens im nächsten Kapitel brauchen wir das.

Und wie löscht man einen Buffer? Auch das ist ganz einfach, man malt den Buffer einfach mit einer bestimmten Farbe aus und dafür gibt es die Device Funktion Clear():

```
g_lpD3DDevice->Clear(0, NULL,                      // Total löschen
                     D3DCLEAR_TARGET |             // RenderTarget leeren
                     D3DCLEAR_ZBUFFER,             // Z-Buffer leeren
                     D3DCOLOR_XRGB(50,10,10),// RenderTarget Füllwert
                     1.0f,                         // Z-Buffer Füllwert
                     0);                           // Stencil Füllwert
```

Die ersten beiden Parameter geben eine Anzahl an Rechtecken (*Param1*) beziehungsweise ein Array mit der entsprechenden Anzahl an Rechtecken (*Param2*) an. Diese Rechtecke sind die Bereiche auf dem BackBuffer die gelöscht werden sollen. Geben wir hier 0 und NULL an so wird der gesamte BackBuffer gelöscht. Die Auswahl bestimmter Bereiche kann unter Umständen für Spezialeffekte benötigt werden.
Der nächste Parameter gibt an, welche Elemente des BackBuffers wir löschen wollen. Wir haben hier definiert dass wir das **RenderTarget** (*also den eigentlichen BackBuffer mit den grafischen Daten*) löschen wollen, ebenso wie den Z Buffer. Alternativ könnten wir auch den StencilBuffer durch die zusätzliche Angabe von D3DCLEAR_STENCIL löschen.
Der dritte Parameter gibt dann an, mit welchem Farbwert der BackBuffer gefüllt werden soll. Ich habe hier ein sehr dunkles Rot gewählt. Natürlich kann man von Tiefschwarz bis Hellweiss alle Farben benutzen die einem gefallen. Der vorletzte Parameter definiert mit welchen Wert der Z Buffer gefüllt wird. Ich hatte ja bereits weiter oben gesagt, dass die Werte im Z Buffer in der Regel von 0.0f bis 1.0f laufen. Ein Wert von 1.0f im Z Buffer für einen Pixel des RenderTargets bedeutet sozusagen eine Entfernung von Unendlich. Und jeder neue Pixel an dieser Stelle der zu einem 3D Objekt gehört welches näher am Spieler als Unendlich ist darf gerendert werden. Damit haben wir also den gesamten Z Buffer so gefüllt, dass jeder Pixel zunächst wenigstens einmal beschrieben werden kann.

Dasselbe gilt für den Parameter des StencilBuffers. Wir haben hier das Löschen dieses SpezialBuffers nicht aktiviert (*wir haben ja bei der Initialisierung im letzten Kapitel auch nur 0 Bit = NIX für den StencilBuffer reserviert*) und daher wird der Wert in diesem Parameter ignoriert.

Okay, damit ist der BackBuffer gelöscht. Jetzt rufen wir unsere Renderfunktion für das Dreieck mit Textur auf. Doch wie bekommen wir nun eigentlich den BackBuffer in den FrontBuffer und umgekehrt? Das machen wir wie folgt:

```
g_lpD3DDevice->Present(NULL, NULL, NULL, NULL);
```

Die ganzen Parameter der Funktion dienen verschiedenen, für uns uninteressanten, Zwecken. Wir können beispielsweise nur Teile des BackBuffers präsentieren oder ein Fensterhandle angeben um ein anderes Fenster als unsere zu benutzen. Aber im Normalfall setzen wir alle vier Parameter auf `NULL` und sind zufrieden. Direct3D trägt übrigens auch Sorge dafür, dass unsere globale Variable `g_lpD3DSBack` immer auf den aktuellen BackBuffer zeigt, auch wenn unsere beiden Buffer (*Front- und Back-*) ständig gewechselt werden. Und hier ist der neue Code unserer Hauptschleife:

```
case SPIEL_LAEUFT:
   {
   // Löschen des BackBuffer
   g_lpD3DDevice->Clear(0, NULL,   // Total löschen
         D3DCLEAR_TARGET |         // RenderTarget leeren
         D3DCLEAR_ZBUFFER,         // Z-Buffer leeren
         D3DCOLOR_XRGB(50,10,10),  //  RenderTarget Füllwert
         1.0f, 0);                 // Z-,Stncl Füllwert

   // Rendere das Dreieck in den BackBuffer
   xD3D_Dreieck_mit_Textur_rendern();

   // Tausche Front- und BackBuffer
   g_lpD3DDevice->Present(NULL, NULL, NULL, NULL);
   } break;
```

Nach so viel Spass muss es auch wieder ein Tief geben oder? Also zurück zur harten Realität der Mathematik und Theorie.

Die Projektionsmatrix

Lange, lange habe ich dieses Thema vor mir hergeschoben. Aber wen ich bisher nicht mit all den Goodies des Renderns blenden konnte der hat natürlich bemerkt, dass wir unsere Projektionsmatrix immer noch nicht erstellt haben. Die DirectX Dokumentation sagt aus, dass man für die Projektionsmatrix den folgenden Ansatz verwenden soll:

```
w    0    0     0
0    h    0     0
0    0    Q     1
0    0  -QZn    0
```

Die Variable Z_n in dieser Matrix ist einfach die Mindestentfernung ab der der Spieler sehen kann, auch *Near Clipping Plane* genannt. Für die Berechnung der anderen Variablen `w`, `h` und `Q` gibt es verschiedene Möglichkeiten. In der Version 7 von DirectX gab es noch andere Formeln als jetzt in der neuen Version 8, aber ich verwende noch die alte Version, welche natürlich immer noch funktioniert:

```
/**
 * Erstellt Projektionsmatrix für perspekt.Projektion
 * anhand der übergebenen Daten. Basiert auf der
 * D3DUtil.cpp Funktion des DirectX 7 SDK's.
```

```
 */
BOOL xD3D_Projektions_Matrix(D3DMATRIX *mat,  // Speicheradresse
                              float fFOV,      // Sichtfeld
                              float fAspect,   // Bildschirmratio
                              float fNearPlane,// Sichtweite von
                              float fFarPlane) // Sichtweite bis
   {
   if (fabs(fFarPlane-fNearPlane) < 0.01f)
      return FALSE;
   if (fabs(sin(fFOV/2)) < 0.01f)
      return FALSE;

   float w = fAspect * ( cosf(fFOV/2)/sinf(fFOV/2) );
   float h = 1.0f * ( cosf(fFOV/2)/sinf(fFOV/2) );
   float Q = fFarPlane / ( fFarPlane - fNearPlane );

   ZeroMemory(mat, sizeof(D3DMATRIX));
   (*mat)._11 = w;
   (*mat)._22 = h;
   (*mat)._33 = Q;
   (*mat)._34 = 1.0f;
   (*mat)._43 = -Q*fNearPlane;

   return TRUE;
   } // xD3D_Projektions_Matrix
/*-----------------------------------------------*/
```

Hierbei handelt es sich um finsterste Mathematik die wir besser nicht hinterfragen werden. Das wichtigste für uns ist folgendes: Wir können nun eine Matrix für die korrekte Projektion mit Direct3D erstellen indem wir ein paar simple Werte angeben. Das haben wir ja weiter oben bereits gesehen. Und was noch wichtiger ist: Durch diese Funktion kann man die Projektionsmatrix bei Bedarf auch schnell ändern. Im Normalfall wird man die Projektionsmatrix zwar während des Programms nicht mehr ändern müssen...aber wann ist die Welt schon mal normal.
Für Spezialeffekte kann dies unter Umständen schon mal nötig werden. Beispielsweise kann man die 3D Ansicht zoomen indem man den Wert des Parameters `fFOV`, also den Blickwinkel des Betrachters, einengt. Also von dem aktuellen Wert von 0.8f auf einen Wert von ca. 01.f, für alle die ein Fernglas programmieren wollen.

Peek statt Get

Kommen wir noch einmal zum Nachrichtenverkehr unter Windows. Ein Windowsprogramm soll natürlich möglichst wenig Rechenzeit kosten, daher ist die Funktion `GetMessage()` eine kleine Fussangel für uns. Wenn wir diese Funktion aufrufen, dann geht sie in die Nachrichtenwarteschlange unseres Fensters und holt die nächste dort wartende Nachricht ab. Dann geht es weiter im normalen Verlauf der Hauptschleife.
Aber was ist nun wenn in der Warteschlange keine Nachricht wartet...? Dann haben wir ein Problem, denn die Funktion wartet dann so lange an der Wartschleife bis dort eine Nachricht ankommt. So lange steht unser Programm aber still und erst wenn wir eine Nachricht auslösen (*in diesem Fall haben wir nur das Drücken der Escapetaste als Mitte zum Zweck*) läuft unser Programm weiter.

Ziemlich doof denn unser Programm soll eben nicht auf Nachrichten warten. Es soll diese verarbeiten falls sie kommen und anderenfalls soll unser Programm seine Hauptschleife immer wieder durchlaufen bis wir das Programm beenden. Und genau das erreichen wir durch die Funktion `PeekMessage()` durch die wir die Funktion `GetMessage()` nun ersetzen werden. In der folgenden Syntax sagt diese Funktion aus, dass wir eine Nachricht abholen und damit aus der Warteschlange entfernen.

```
PeekMessage(&message, NULL, 0, 0, PM_REMOVE);
```

Alles andere was den Nachrichtenverkehr angeht bleibt in unserem Programm wie gehabt. Nun könnt Ihr den Code laden und das Programm austesten. Wenn das Dreieck nun etwas gelblich erscheint, obwohl die Textur normale Farben hat, so sollte man nicht vergessen dass das definierte Material die Farbe des Objektes auch beeinflusst. Auch hier sei wieder zum rumspielen mit dem Werten angeregt!

Und hier gibt es den Code des *gesamten* Tutorials zum Download: Projektdateien

---

Weiter geht's zum Kapitel 5...

von Stefan Zerbst

"Gentlemen walk but never run."
Englishman in New York, Sting

Das Dreieck dreht sich

Wow...das ist doch schon was. Im letzten Kapitel haben wir gelernt ein Dreieck mit Textur auf den Bildschirm zu rendern. Doch jetzt wird es noch besser. In diesem Kapitel werden wir uns ansehen, wie wir Dreiecke beziehungsweise jedes nur erdenkliche 3D Objekt rotieren und im 3D Raum verschieben können. Diese grundlegenden Techniken brauchen wir für jedes 3D Objekt, ob nun unser einfaches Dreieck hier oder einen komplexen Raumjäger. Diese besteht schliesslich aus der Sicht von Direct3D auch nur auf vielen kleinen Dreiecken.

Die Einheitsmatrix

Was wären wir ohne die gute alte Einheitsmatrix? Sie ist sozusagen das neutrale Element in der Welt der Matrizen. Wenn wir eine Matrix mit einer anderen multiplizieren (*und das werden wir nachher tun müssen*) und eine der beiden Matrizen ist die Einheitsmatrix, so kommt als Ergebnis wieder genau die andere Matrix heraus. Die Einheitsmatrix verändert das Ergebnis einer Matrizenmultiplikation also nicht. Laber, laber, laber...hier ist eine Funktion die aus einer übergebenen `D3DMATRIX` eine Einheitsmatrix erstellt:

```
inline void xUtil_Einheitsmatrix(D3DMATRIX *mat) {
   mat->_11 = mat->_22 = mat->_33 = mat->_44 = 1.0f;
   mat->_12 = mat->_13 = mat->_14 = mat->_41 = 0.0f;
   mat->_21 = mat->_23 = mat->_24 = mat->_42 = 0.0f;
   mat->_31 = mat->_32 = mat->_34 = mat->_43 = 0.0f;
   } // xUtil_Einheitsmatrix
```

Ich habe mal in einer Spieleprogrammierungs-Newsgroup gelesen, dass die Arbeit mit Matrizen für 3D Grafik schwer sein soll. Wer hier auch dieser Meinung ist, der hebt bitte den Finger *hehe*

Sämtliche Einträge der Matrix werden auf 0 gesetzt, lediglich die Hauptdiagonale wird mit 1 Einträgen belegt. Das ist auch schon alles, um eine schicke Einheitsmatrix zu erhalten. Aber warum zum Teufel brauchen wir eigentlich bei der Programmierung von 3D Grafik diese Matrizen? Man hört das an jeder Ecke und alle schrecken einen mit Schauermärchen über die Komplexität der Matrizenrechnung ab.

Erst mal zum Punkt Komplexität. In der Matrizenrechnung tut man nix anderes als zu multiplizieren und zu addieren und das lernt jeder in der Grundschule. Alles andere hat damit zu tun, dass man leicht mit den Zeilen und Spalten der Matrizen durcheinander kommt, aber im Endeffekt kann man alles ganz fix im Kopf rechnen oder bei etwas komplizierteren Zahlen den guten alten Taschenrechner verwenden.
Nun zur zweiten Frage: Warum brauchen wir diesen ganzen Matrizenkrams? Wenn wir uns gleich daran machen unser Objekt zu bewegen dann gibt es zwei Arten von Bewegungen. Zum einen können wir das Objekt in unserer Welt hin und her schieben und zum anderen können wir das Objekt rotieren. Dazu müssen wir je nach Bewegungsart alle Vertices unseres Modells mit gewissen Formeln umrechnen, wobei diese Formeln dann die Verschiebung oder Rotation um eine bestimmte Achse beinhalten. Die Vertices mit den Originalkoordinaten des Modells nennt man **lokale Koordinaten** (*z.B. die ursprünglichen drei Vertices unseres Dreiecks*). Nach der Anwendung der eben erwähnten Formeln sind die Vertices dann an der Position im 3D Raum in der sie in Relation zu allen anderen Objekten des 3D Raumes nach ihrer Bewegung auch stehen

sollten. Daher nennt man diese Koordinaten dann **Weltkoordinaten**.
Bei Direct3D funktioniert das aber rein äusserlich etwas anders. Die lokalen Koordinaten unserer Vertices bleiben immer erhalten. Wir berechnen die entsprechenden Formeln die für die aktuelle Drehung und Verschiebung eines Objektes im aktuellen Frame notwendig sind und weisen diese Formeln dann dem Direct3D Device zu. Alle Vertices die wir jetzt über das Device rendern werden dann von dem Device entsprechend mit genau diesen Rotationen und genau dieser Verschiebung von lokalen in Weltkoordinaten umgerechnet und dann gerendert. Cool, oder? Und gar nicht schwer.

Nun zu den konkreten Formeln und zurück zur Frage nach den Matrizen. Es gibt zwar auch *normale* Formeln für die notwendigen Berechnungen, ohne die Verwendung von Matrizen. Diese Matrizen haben aber einen entscheidenden Vorteil. Wir benötigen eine Formel für die Verschiebung des Objektes auf alle drei Achsen des 3D Raumes. Dazu benötigen wir je eine Formel für die Rotation um jeweils eine der drei Achsen. Damit haben wir schon vier Formeln. Nun müssen wir jede dieser vier Formeln einzeln auf jeden unserer Vertices hetzen um diesen in Weltkoordinaten umzurechnen.
Das klingt nicht nur aufwendig, sondern ist es auch. Eine bessere Alternative bieten die Matrizen. Wir können nämlich für jede dieser vier Formeln ebenfalls eine geeignete Matrix finden, die bei der Multiplikation mit einem Vertex genau dasselbe Ergebnis liefert. An sich ist das noch nicht dramatisch, der Clou ist aber, dass wir viele Matrizen zu einer einzigen Matrix zusammen multiplizieren können und die Ergebnismatrix dann immer noch dieselben Informationen enthält. Konkret gesagt: Wir werden die drei Matrizen für die Rotationen einmal pro Frame erstellen, dann zusammen multiplizieren, die Verschiebung des Objektes einbeziehen und dann diese Matrix dem Device zuweisen.
Und jetzt senden wir unsere Vertices in lokalen Koordinaten an das Device. Anstatt auf jeden Vertex vier Formeln zu hetzen muss das Device jeden Vertex lediglich mit dieser einen Matrix multiplizieren. Dadurch werden enorm viele Rechnungen eingespart und unsere 3D Engine wird superschnell. Aber es wird noch besser. Die Projektionsmatrix, die wir dem Device ja schon mitgeteilt haben, kann von dem Device ebenso zu der Weltmatrix multipliziert werden um die Projektion auch noch in die Megamatrix mit einzubinden. Und wenn wir später zu der Bewegung des Spielers selbst kommen dann werden wir lernen, auch die Bewegung des Spielers noch in diese Matrix zu stopfen. Mehr Berechnungen kann man schon gar nicht sparen.

Verschieben eines Objektes

Die Verschiebung eines Objektes im 3D Raum ist denkbar einfach. Wir addieren einfach zu allen Vertices eines Objektes die gewünschte Verschiebung auf der entsprechenden Koordinate der korrespondierenden Achse. Oder auf C Deutsch für unser Dreieck:

```
for (int i=0; i<3; i++) {
   g_Tri[i].x += n_Verschiebung_auf_x_Achse;
   g_Tri[i].y += n_Verschiebung_auf_y_Achse;
   g_Tri[i].z += n_Verschiebung_auf_z_Achse;
   }
```

Wir hatten aber gesagt dass wir die lokalen Koordinaten nicht ändern wollen und statt dessen eine Matrix für die Verschiebung erstellen die wir dem Device zuweisen können. Diese Matrix sieht wie folgt aus:

```
    1    0    0    0
    0    1    0    0
    0    0    1    0
    tx   ty   tz   1
```

Wir nehmen einfach die Einheitsmatrix und platzieren die Verschiebung auf der x, y und z Achse in der letzten Zeile. Jeder Vertex den wir nun mit diese Matrix multiplizieren wird korrekt wie gewünscht auf den 3 Achsen verschoben.

Rotieren eines Objektes

Beim Rotieren eines Objektes wird es schon etwas wilder. Hier kommt ein wenig Trigonometrie ins Spiels und das klingt nicht nur nach den Kreisfunktionen sin und cos, sondern hat auch wirklich etwas damit zu tun. Ich spare mir an dieser Stelle die Herleitung der folgenden Matrizen und gebe sie einfach mal so an. Aber man kann tatsächlich mathematisch erklären, wie diese Matrizen zustande kommen und dass sie korrekt für eine Rotation sorgen.

Diese Matrix hier dreht einen Vertex mit dem wir sie multiplizieren auf der x Achse und `a` RAD:

```
    1     0        0       0
    0   cos(a)  sin(a)    0
    0  -sin(a)  cos(a)    0
    0     0        0       1
```

Entsprechend können wir uns eine Funktion schreiben mit deren Hilfe wir so eine Rotationsmatrix erzeugen können:

```
void xUtil_RotationsmatrixX(D3DMATRIX *mat, float fRads)
   {
   xUtil_Einheitsmatrix(mat);
   (*mat)._22 =  cosf(fRads);
   (*mat)._23 =  sinf(fRads);
   (*mat)._32 = -sinf(fRads);
   (*mat)._33 =  cosf(fRads);
   } // xUtil_RotationsmatrixX
```

Ja, das ist alles was wir beherrschen müssen um ein Objekt auf der x Achse zu drehen. Gehen wir gleich weiter zu der y Achse, was erstaunlicherweise auch nicht komplizierter ist. Hier die Matrix um einen Vertex um `a` RAD um die y Achse zu drehen:

```
  cos(a) 0 -sin(a) 0
    0    1    0    0
  sin(a) 0  cos(a) 0
    0    0    0    1
```

Und die Funktion um die Matrix zu erzeugen:

```
void xUtil_RotationsmatrixY(D3DMATRIX *mat, float fRads)
   {
   xUtil_Einheitsmatrix(mat);
   (*mat)._11 =  cosf(fRads);
   (*mat)._13 = -sinf(fRads);
   (*mat)._31 =  sinf(fRads);
   (*mat)._33 =  cosf(fRads);
   } // xUtil_RotationsmatrixY
```

Auch nix schlimmes passiert bisher. Also her mit der Matrix für die Rotation eines Vertex um die z Achse.

```
  cos(a) sin(a)  0    0
 -sin(a) cos(a)  0    0
    0      0     1    0
    0      0     0    1
```

Zu guter Letzt die Funktion zur Erstellung dieser Rotationsmatrix:

```
void xUtil_RotationsmatrixZ(D3DMATRIX *mat, float fRads)
   {
   xUtil_Einheitsmatrix(mat);
   (*mat)._11 =  cosf(fRads);
```

```
   (*mat)._12 =  sinf(fRads);
   (*mat)._21 = -sinf(fRads);
   (*mat)._22 =  cosf(fRads);
   } // xUtil_RotationsmatrixZ
```

Und das war's, jetzt sind wir fast schon am Ende des Themas Rotation! Um aber auch mehr als nur eine Rotationsachse je Frame zuzulassen müssen wir nun auch noch lernen, wie wir die Rotationsmatrizen zusammen multiplizieren können.

Multiplizieren von Matrizen

Dies ist nun endlich die Stelle wo unser Gehirn ein wenig zum Denken angeregt wird. Das Multiplizieren von Matrizen ist etwas komplexer als die bisherigen Operationen, aber nicht viel mehr. Es läuft alles auf Multiplikation und Addition hinaus. Wir wollen nun eine Matrix A mit einer Matrix B zu der Ergebnismatrix C zusammen multiplizieren. Ein Eintrag c_{ij} (*i=Zeile, j=spalte*) in der Matrix C ergibt sich dann aus der Multiplikation der i'ten Zeile der Matrix A mit der j'ten Spalte der Matrix B.
Und wie multipliziere wir eine Zeile mit einer Spalte? Auch das ist ganz einfach. Betrachten wir die Zeile und die Spalte jeweils als einen Zahlenstrang, dann multiplizieren wir die beiden ersten Elemente der Stränge miteinander, addieren dazu die Multiplikation der beiden zweiten Elemente usw. Nur am Rande: *Aus diesem Grund muss Matrix A so viele Spalten haben wie Matrix B Zeilen hat. Bei uns haben aber alle Matrizen je vier Zeilen und Spalten, da gibt es also kein Problem. In diesem Fall hat auch die Ergebnismatrix C wieder vier Zeilen und Spalten.*
Und wem das zu schnell ging noch einmal ein kurzes Life-Fire Beispiel mit 2x2 Matrizen:

```
  |a11  a12|   |b11  b12|   |a11*b11+a12*b21  a11*b12+a12*b22|
  |        | * |        | = |                                |
  |a21  a22|   |b21  b22|   |a21*b11+a22*b21  a21*b12+a22*b22|
```

Easy, wenn man kurz drüber nachdenkt. Hier ist eine etwa verschärfte Version im C Code die wir verwenden werden. Hier muss man eventuell etwas näher hinschauen, um die Berechnung zu erkennen oder man verwendet die Funktion einfach so...oder schreibt seine eigene.

```
/**
 * Multiplikation zweier Matrizen: A*B = C
 */
void xUtil_MatrixMult(D3DMATRIX *C, D3DMATRIX *A, D3DMATRIX *B) {
   float* pA = (float*)A;
   float* pB = (float*)B;
   float  pM[16];

   ZeroMemory(pM, sizeof(D3DMATRIX));

   for (WORD i=0; i<4; i++)
      for (WORD j=0; j<4; j++)
         for (WORD k=0; k<4; k++)
            pM[4*i+j] += pA[4*i+k] * pB[4*k+j];

   memcpy(C, pM, sizeof(D3DMATRIX));
   } // xUtil_MatrixMult
```

So, jetzt können wir auch Matrizen multiplizieren. Und soll ich Euch etwas sagen? Das ist alles was wir über Matrizenrechnung wissen müssen, um eine schicke 3D Engine zu programmieren. Ich suche übrigens immer noch die Stelle wo das schwer sein soll...

Die Weltmatrix

Jetzt sind wir mit 3D Code bis an die Zähne bewaffnet und es wird Zeit, diesen auch einzusetzen und auf unser Dreieck loszulassen. In der Funktion `xD3D_Dreieck_mit_Textur_rendern()` werden wir jetzt für jeden Frame die Rotation und Verschiebung für das Dreieck berechnen und ausführen. Damit lernen wir die zweite Verwendung für die Funktion `SetTransform()` des Device kennen. Neben der Einstellung der Projektionsmatrix können wir durch den Bezeichner `D3DTS_WORLD` die nun berechnete Weltmatrix für das Device festlegen, mit deren Hilfe alle nun bei dem Device eintreffenden lokalen Koordinaten in Weltkoordinaten umgerechnet werden.
Ich habe für dieses Beispiel die globalen Variablen `g_fRotX`, `g_fRotY`, `g_fRotZ` und `g_fMove` deklariert. Die ersten drei legen die aktuelle Rotation unseres Dreiecks fest und werden mit 0.0f RAD (*keine Rotation*) initialisiert. Die vierte Variable dient dazu, das Objekt auf der z Achse zu verschieben.

Eines habe ich bisher noch verschwiegen. Wenn wir ein 3D Objekt rotieren, so wird es natürlich um seinen lokalen Mittelpunkt rotiert. Für unser Dreieck ist dieser lokale Mittelpunkt (*so wie für alle Objekte*) der Punkt (0,0,0). Allerdings haben wir die Koordinaten des Dreiecks so angelegt, dass alle Vertices bei 3.0 auf der z Achse liegen, also ein gutes Stück vom lokalen Mittelpunkt verschoben sind. Das führt dazu, dass unser Dreieck sich etwas merkwürdig rotieren lässt, es läuft nämlich eine Kreisbahn um den Spieler herum. Das liegt daran, dass sich das Dreieck um seinen lokalen Ursprung (0,0,0) dreht. Daher habe ich alle z Koordinaten der Vertices auf 0.0 gesetzt. So sollte das eigentlich auch für jedes Objekt sein, es muss mit seiner Geometrie in seinem lokalen Nullpunkt zentriert sein. Nun dreht sich das Dreieck sauer um seine eigenen Achsen wie wir das wollen. Soll das Dreieck weiterhin drei Einheiten vor dem Spieler sein, so setzen wir `g_fMove` einfach auf 3.0 und führen nach der Rotation eine Verschiebung auf der z Achse durch. Und hier der neue Code in der Funktion:

```
D3DMATRIX matRotX, matRotY, matRotZ,
          matTemp, matWelt;

// Haben wir einen vollen Kreis (verhindert
// Überlauf der Variablen bei Maximalwert
if (g_fRotY >= (360.0f / (2*PI)))
   g_fRotY = 0.0f;

// In jedem Frame um ca.2 Grad rotieren
g_fRotY += 0.06f;

// Objektrotationen
xUtil_RotationsmatrixX(&matRotX, g_fRotX);
xUtil_RotationsmatrixY(&matRotY, g_fRotY);
xUtil_RotationsmatrixZ(&matRotZ, g_fRotZ);

xUtil_MatrixMult(&matTemp, &matRotX, &matRotY);
xUtil_MatrixMult(&matWelt, &matTemp, &matRotZ);

// Objektverschiebung
matWelt._41 = 0.0f;
matWelt._42 = 0.0f;
matWelt._43 = g_fMove;

g_lpD3DDevice->SetTransform(D3DTS_WORLD, &matWelt);
```

Zuerst lernen wir hier auch noch die Umrechnung von Grad in RAD ganz genau: (Grad / 2*Pi) ergibt die Anzahl an RAD für den entsprechenden Wert in Grad. In diesem Beispiel achten wir darauf, dass wir die Rotation immer wieder auf 0 Grad setzen, wenn das Dreieck um 360 Grad rotiert wurde. Das ist zunächst einmal unnötig, denn 360 Grad sind gleich 0 Grad und wenn wir zweimal 36 Grad gedreht haben dann ist das ebenfalls gleich 0 Grad. Auf diese Weise verhindern wir aber, dass unsere Variable irgendwann mal *überläuft* weil wir sie immer grösser machen.
Als nächstes sehen wir, dass wir lediglich um die y Achse drehen. Ihr könnt aber auch die anderen beiden

Rotationswerte beliebig setzen und sehen, wie das rüberkommt, kein Problem. Danach erstellen wir die drei Rotationsmatrizen für die aktuellen Drehwinkel und Multiplizieren die Matrizen zusammen. Danach legen wir noch schnell die Verschiebung in der Ergebnismatrix fest (*dafür brauchen wir keine eigene Matrix und keine Multiplikation*) und haben alles beisammen.

Noch zwei Dinge: Ich habe noch schnell in den Code eingebaut, dass das Dreieck ein Stückchen auf der z Achse hin und her gleitet. Aber der entsprechende Code ist kinderleicht und Ihr werdet keine Probleme damit haben.
Zum anderen kann man jetzt endlich sehen, dass das Dreieck wirklich nur eine Seite hat. Wenn wir Rückseite nach vorne rotiert ist, dann sehen wir dass wir sie nicht sehen. So, jetzt ladet den Code um spielt etwas damit rum...

Und hier gibt es den Code des *gesamten* Tutorials zum Download: Projektdateien

---

Weiter geht's zum Kapitel 6...

SPIELEENTWICKLUNG MIT DIRECT3D IM - KAPITEL 6

von Stefan Zerbst

"You did what you did to me. Now it's history I see."
Alphaville's Big in Japan, Guano Apes

3D Modelle laden und rendern

Sind Dreiecke langweilig, oder was? Nachdem wir unsere Hausaufgaben nun brav erledigt haben kommen wir endlich zu den coolen Aspekten der 3D Spieleprogrammierung. In diesem Kapitel werden wir lernen ein 3D Modell aus einer `*.x` Datei in unser Programm zu laden, zu verschieben und zu rotieren. Wie man nach dem letzten Kapitel bestimmt schnell einsehen kann ist es nicht sehr einfach, die Geometrie eines Objektes komplexer als ein Würfel von Hand (*im Quellcode*) zu erstellen und dann noch mit Texturkoordinaten zu versehen.
Dazu verwendet man dann besser 3D Modellierungssoftware mit deren Hilfe an per Mausklicks wie in einem Zeichenprogramme seine Modelle entwerfen kann. So ein Programm ist beispielsweise AC3D (*siehe Link im Menü auf der linken Seite*). Auf die Erstellung eines solchen 3D Modells will ich hier nicht weiter eingehen. Es sei nur so viel erwähnt dass im DirectX SDK auch der Konverter conv3ds.exe enthalten ist mit dessen Hilfe man `*.3ds` Dateien des Programms 3D Studio in X File Dateien konvertieren kann. Das ist deshalb interessant weil es überall im Internet 3D Modelle in diesem Format zu finden gibt.

Ich setze hier voraus, dass wir bereits unsere Modelle in entsprechenden Dateien als X Files vorliegen haben und diese nun in unser Projekt laden wollen. Das Modell welches ich hier verwende ist der Cat Raumjäger des Spiels Wing Captain 2.0 aus meinem ersten Buch. Die Geschichte dieses Modells ist auch schon recht lang. Als ich vor Urzeiten (*also Ende 98/Anfang 99*) André LaMothe's Buch "Black Art of 3D Game Programming" durchgearbeitet habe um 3D Programmierung zu lernen, da habe ich für die Demoprogramme des Buchs selbst ein Modell basteln wollen. Also griff ich zu Stift und Papier und konstruierte auf dem harten Weg (*durch Nachdenken*) die erste Version des Cat Raumjägers durch Aufmalen der einzelnen Vertices auf einem Koordinatensystem auf Papier. Das geht also...ist aber nicht besonders empfehlenswert!
Einzig die Triebwerksbehausungen und das Seitenleitwerk fehlten der damaligen Version. Für das Tutorial Wing Captain 1.0 auf dieser Homepage habe ich den Cat Raumjäger dann etwas aufgepeppt, war aber schlau genug nun AC3D dafür zu verwenden, jedoch fehlten der Cat noch vernünftige Texturen und die Laser. In der aktuellen Version hat die Cat nun endlich vernünftige Texturen, die beiden Laserkanonen wurden unter dem Rumpf montiert und die Flügelspitzen wurden noch ein wenig nach unten gezogen. Und hier ist die Cat:

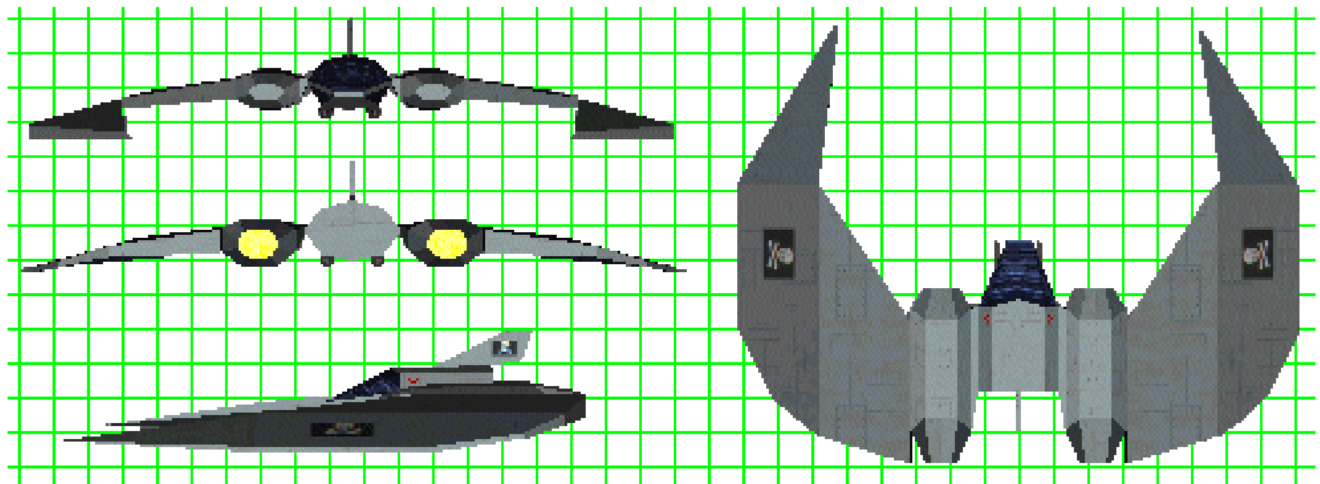


Abbildung 1: Der Cat Raumjäger

Okay, ich gebe es gerne zu. Als ich die erste Version dieses Modells per Stift und Papier entworfen habe war Chris Robers "Wing Commander III - Heart of the Tiger" gerade ziemlich angesagt. **;-)**

Laden eines X File Modells

Der erste Schritt besteht natürlich darin das als X File gespeicherte 3D Modell zu laden. Idealerweise würde man an dieser Stelle seine eigenen Datenstrukturen entwickeln in denen man die Vertices, Materialien, Texturen usw. des Modells speichert. Diese Strukturen kann man dann für das Kollisionssystem seines Projektes ideal abstimmen. Da das aber ein ziemlich aufwendiger Vorgang ist werden wir hier die abgespeckte Variante des Ladens vollziehen und uns einiger Strukturen der DirectX DX Bibliothek bedienen. Die erstgenannte *Hardcore*-Variante finden Interessierte im zweiten Band meiner Bücherreihe.

```
typedef struct X3DMODELL_TYP {
   DWORD               dwNumMaterials;
   D3DMATERIAL8       *pMeshMaterials;
   LPDIRECT3DTEXTURE8 *pMeshTextures;
   LPD3DXMESH          pMesh;
   } X3DMODELL;
```

Schwups, da ist auch schon die erste Datenstruktur. Wir definieren uns eine Struktur die alle DirectX DX Elemente beinhaltet die zum Laden eines X File Modells notwendig sind. Das `D3DMATERIAL8` Objekt kennen wir ja schon, ebenso das Texturobjekt, die brauche ich also nicht weiter zu erklären. Daneben gibt es noch einen Zähler für die Anzahl der Materialien. Beim Laden des X Files wird beispielsweise jede neue Textur als ein neues Material definiert und die einzelnen Teile des Modells werden dann nach ihren Materialien geordnet gerendert. Warum das gut ist dazu kommen wir später noch, aber wir sehen hier schnell ein, dass wir diesen Zähler nützlich finden könnten. Das wirklich neue Objekt in dieser Struktur ist das `LPD3DXMESH` Objekt. Mit Hilfe einiger weniger DX Funktionen können wir die Geometrie des X Files in dieses Objekt laden. Also fangen wir an:

```
/**
 * Lädt die angegebene X File Datei in ein X3DMODELL Objekt
 */
X3DMODELL xMod_lade_XFile(char* achName) {
   LPD3DXBUFFER pD3DXMtrlBuffer;
   HRESULT      hr;
   X3DMODELL    X;

   X.dwNumMaterials = 0L;
   X.pMeshMaterials = NULL;
   X.pMeshTextures  = NULL;
   X.pMesh          = NULL;

   // Lade die X File Datei mit der DX Funktion
   hr = D3DXLoadMeshFromX(achName, D3DXMESH_SYSTEMMEM,
                          g_lpD3DDevice, NULL,
                          &pD3DXMtrlBuffer, &X.dwNumMaterials,
                          &X.pMesh);
   if (FAILED(hr)) {
      fprintf(Protokoll, "Fehler: DXLoadX(\"%s\") failed \n", achName);
      if (hr==E_OUTOFMEMORY)
         fprintf(Protokoll, " -> OUT_OF_MEMORY \n");
      return X;
      } // if
```

wird fortgesetzt...

Die Funktion `D3DXLoadMeshFromX()` erledigt hier die gesamte Arbeit für uns. Als ersten Parameter geben wir

der Funktion den Namen der zu ladenden Datei an und als zweites sagen wir ihr in welchem Speicher (*System oder Videospeicher*) wir das Objekt platzieren wollen. Den dritten Parameter können wir ohne weiteres als das Direct3D Device identifizieren auf dem wir das Modell später rendern wollen. Der vierte Parameter wird von uns nicht benötigt, könnte aber mit Informationen über benachbarte Flächen des Objektes gefüllt werden.
Nun wird es etwas spannender denn der folgende Parameter ist ein Objekt des Typs `LPD3DXBUFFER` das die Funktion dann mit Informationen über die verwendeten Materialien des Objektes füllt. Diese Funktion ist nämlich nur dazu da, die Geometrie zu laden. Die Materialien müssen wir noch gesondert behandeln, dazu packen wir sie in dieses Objekt. Der nächste Parameter wird dann von der Funktion mit der Anzahl von gefundenen Materialien gefüllt. Zu guter Letzt haben wir dann auch das `D3DXMESH` Objekt in das die Funktion die Geometrie des X File Modells steckt.

In dem `D3DXBUFFER` Buffer Objekt stecken nun alle Informationen über die verwendeten Materialien (*Texturen und Lichtreflexionswerte*) in Form von `D3DXMATERIAL` Objekten. Sehen wir uns also an wie das aussieht:

```
struct D3DXMATERIAL {
   D3DMATERIAL8 MatD3D;
   LPSTR        pTextureFilename;
   };
```

Okay, das ist einfach genug. Das DX Material Objekt enthält also einfach ein normales Direct3D Material Objekt und den Namen der Grafikdatei die als Textur die zu diesem Material gehört. Wir werden also im folgenden einen `D3DXMATERIAL` Zeiger erzeugen und auf die Daten des eben gefüllten `D3DXBUFFER` Objektes setzen. Das tun wir mit der Funktion `GetBufferPointer()`. Danach können wir über diesen Pointer alle Materialien durchlaufen und die Informationen in die entsprechenden Felder unserer Datenstruktur `X3DMODELL` umspeichern. Dazu müssen wir aber erst noch den Speicher für diese Felder allokieren:

xMod_lade_XFile() Fortsetzug:

```
   // Zeiger auf Materialdaten setzen
   D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)
                    pD3DXMtrlBuffer->GetBufferPointer();

   // Speicher allokieren
   X.pMeshMaterials = (D3DMATERIAL8*)malloc(X.dwNumMaterials *
                                         sizeof(D3DMATERIAL8));
   X.pMeshTextures = (LPDIRECT3DTEXTURE8*)malloc(X.dwNumMaterials *
                                              sizeof(LPDIRECT3DTEXTURE8));
   if (!X.pMeshMaterials || !X.pMeshTextures) {
      fprintf(Protokoll, "Fehler: malloc() Mesh Elemente \n");
      return X;
      }
```

wird fortgesetzt...

Aber jetzt können wir die Materialien endlich umspeichern. Man beachte dass das Setzen des Pointers noch kein Umspeichern der Daten bewirkt.

xMod_lade_XFile() Fortsetzug:

```
   // Speichere alle Materialien um und lade Texturen
   for (DWORD i=0; i<X.dwNumMaterials; i++) {
      X.pMeshMaterials[i] = d3dxMaterials[i].MatD3D;

      // Im X File ist amb.und diff.Licht nur eins
      X.pMeshMaterials[i].Ambient = X.pMeshMaterials[i].Diffuse;

      hr = D3DXCreateTextureFromFile(g_lpD3DDevice,
                          d3dxMaterials[i].pTextureFilename,
```

```
                        &X.pMeshTextures[i]);
      if (FAILED(hr)) {
         X.pMeshTextures[i] = NULL;
         fprintf(Protokoll, "Error: %s DXTexture(\"%s\") failed\n",
                 achName, d3dxMaterials[i].pTextureFilename);
         return X;
         } // if
      } // for
   // Aufräumen nicht vergessen!!!
   pD3DXMtrlBuffer->Release();

   return X;
   } // xMod_lade_XFile
/*------------------------------------------------------------*/
```

Der einzige Fallstrick hier ist lediglich, dass das X File Format keine Unterscheidung zwischen ambientem und diffusem Licht macht (*was auch immer das ist*). Folglich definiert die X File Datei nur einen Lichtwert während die Direct3D Material Datenstruktur nach zwei Werten verlangt. Daher kopieren wir den Wert einfach in beide Felder. Dann laden wir flux die Textur und alles ist okay. Damit haben wir das 3D Modell offiziell in unser Programm geladen.

Rendern eines X File Modells

Jetzt haben wir das 3D Modell geladen. Und was tun wir nun, damit unser eben frisch geladenes 3D Modell am Bildschirm erscheint und damit genau so aussieht wie es uns die folgende Abbildung beispielhaft zeigt?

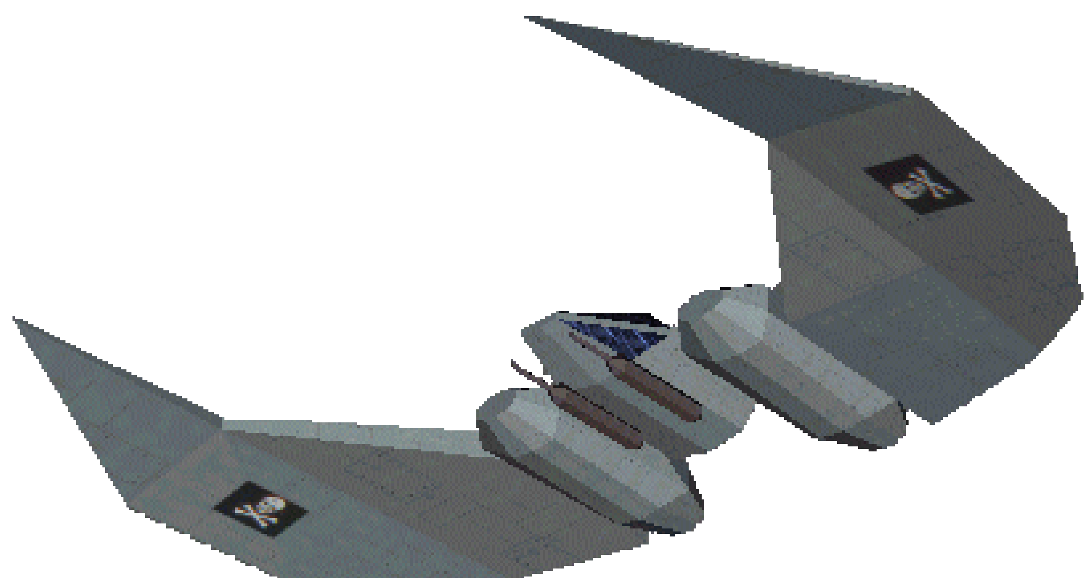


Abbildung 2: Das gerenderte 3D Modell

Das war ein langer einleitender Satz, aber ein bisschen was wollte ich unter dieser Überschrift auch schreiben denn das Rendern eines Modells in unserer `X3DMODELL` Struktur ist ein Kinderspiel. Seht selbst:

```
/**
 * Rendert das DXMESH Objekt im X3DMODELL
 */
BOOL xMod_rendere(X3DMODELL* pX) {
   // Durchlaufe alle Materialien und rendere jeweils alle
   // Vertices die ein Material verwenden durch DrawSubset()
   for (DWORD i=0; i<pX->dwNumMaterials; i++) {
      // Material und Textur für Device einstellen
```

```
      g_lpD3DDevice->SetMaterial(&pX->pMeshMaterials[i]);
      g_lpD3DDevice->SetTexture(0, pX->pMeshTextures[i]);
      // Rendern
      pX->pMesh->DrawSubset(i);
      }
   return TRUE;
   } // xMod_rendere
/*-------------------------------------------------------*/
```

Ein Witz, oder? Wir laufen einfach durch alle Materialien des Objektes und setzen jeweils das Material und die Textur für das Device. Dann rufen wir die Funktion `DrawSubset()` auf und das Objekt wird gerendert.
Nun zum Thema Geschwindigkeit. Jedes Setzen einer Eigenschaft des Direct3D Device Objektes kostet Zeit. Ob wir nun das Material einstellen, oder die Textur oder eine Matrix für das Device ist ganz egal. Noch mehr Zeit kostet es, die entsprechende Draw Funktion des Device aufzurufen die Vertices rendern soll. Es gibt also nix schlimmeres für ein Device als wenn wir blind einfach je ein Dreieck des Modells nehmen würden (*mal angenommen wir könnten das*), dessen Material und Textur festlegen und dann das eine Dreieck rendern lassen würden. Viel intelligenter, und von besonderer Wichtigkeit für die Geschwindigkeit unseres Programms, ist es wenn man alle Dreiecke des Modells nach den Materialien (*und damit auch nach den Texturen*) ordnet und dann immer alle Dreiecke auf einen Sitz rendert die dasselbe Material (*und dieselbe Textur*) verwenden. Genau das passiert aber schon beim Laden eines X File Modells durch die entsprechende DX Funktion die wir verwendet haben. Daher laufen wir durch alle Materialien und rendern die sogenannten Subsets die jeweils alle Dreiecke des Modells enthalten die sich ein Material und eine Textur teilen.

Leider hat dieses `D3DXMESH` Objekt aber anscheinend das Problem, dass identische Materialien im X File 3D Modell nicht als solche erkannt werden, wenn sie mehrfach (und nicht als Referenz) definiert sind. Es kommt nun also darauf an, dass auch die X File Datei vernünftig vorliegt. Für die Beispieldatei dieses Kapitels trifft da natürlich zu, es kann aber sein dass man in diesem Programm bei einigen Modellen bereits Performanceeinbrüche bemerken wird. Das liegt dann genau daran, dass das 3D Modell zu viele verschiedene Materialien definiert hat (auch wenn sie alle dieselben Eigenschaften und Texturen haben) und zu wenig Dreiecke bei einem Aufruf der entsprechenden Draw Funktion des Device liefert.

Aufräumen eines X File Modells

Am Ende unseres Programms müssen wir noch dafür Sorge tragen, dass wir auch jeden belegten Speicher wieder brav freigeben und auch die akquirierten DirectX Objekte wieder in die Freiheit entlassen. Hier ist die Funktion die das für unsere 3D Modelle tun wird.

```
/**
 * Gibt Speicher und alle DirectX Objekte des X3DMOD frei
 */
void xMod_kill(X3DMODELL X) {
   if (X.pMeshMaterials) {
      free(X.pMeshMaterials);
      X.pMeshMaterials = NULL;
      }

   if (X.pMeshTextures) {
      for (DWORD i=0; i<X.dwNumMaterials; i++ ) {
         if (X.pMeshTextures[i])
            X.pMeshTextures[i]->Release();
         } // for
      free(X.pMeshTextures);
      X.pMeshTextures = NULL;
   } // if

   if(X.pMesh)
```

```
      X.pMesh->Release();
   } // xMod_Kill
/*----------------------------------------------*/
```

Kapseln des 3D Modells

Eigentlich könnten wir jetzt bereits 3D Modelle aus X Files laden und zur Anzeige bringen. Es ist aber immer ratsam in einem Spiel zwischen den folgenden zwei Dingen strikt zu unterscheiden: Zum einem gibt es (a) das 3D Modell mit Geometrie und zum anderen gibt es (b) das Objekt mit Position im Raum, Lebensenergie usw. Daher verwenden wir hier schon mal die folgende Datenstruktur um diese Trennung deutlich zu machen:

```
typedef struct XOBJEKT_TYP {
   D3DMATRIX matWelt;
   D3DVECTOR vPos;
   float     fRotX,
             fRotY,
             fRotZ;
   X3DMODELL Modell;
   } XOBJEKT;
```

Wie wir sehen können ist das 3D Modell, welches wir am Bildschirm sehen werden, nur ein kleiner Teil des gesamten Objektes. An erster Stellen speichern wir für ein Objekt seine Weltmatrix. Dazu kommen die Position des Objektes in Weltkoordinaten und die Winkel der aktuellen Rotation des Objektes. Nun können wir dem Objekt in jedem Frame die gewünschte Rotation zuweisen und seine Position verschieben. Dann berechnen wir die Weltmatrix und speichern sie auch in dem Objekt und rendern schliesslich das 3D Modell des Objektes. Und wenn wir schon dabei sind unsere Daten sinnvoll einzukapseln dann entwickeln wir auch gleich noch ein paar Hilfsfunktionen die die Objektstrukturen einfacher verwendbar machen. Namentlich ist das erst mal eine Funktion mit der wir ein solches `XOBJEKT` Objekt unseres Programms initialisieren und mit sinnvollen Startwerten belegen können.

```
/**
 * Initialisiert ein XOBJEKT mit lokalen Achsen, Rota-Winkeln,
 * lädt das XMODELL und legt die Position fest.
 */
XOBJEKT xUtil_init_XObjekt(char *achName, float fX,
                           float fY, float fZ) {
   XOBJEKT XObj;

   // X File in XOBJEKT laden
   XObj.Modell = xMod_lade_XFile(achName);

   // Lade das 3D Modell in das XOBJEKT
   if (!XObj.Modell.pMesh) {
      fprintf(Protokoll, "Fehler: xMod_laden() failed\n");
      return XObj;
      }
   // Weltmatrix des Objektes
   xUtil_Einheitsmatrix(&(XObj.matWelt));

   // Rotationswinkel des Objektes
   XObj.fRotX = 0.0f;
   XObj.fRotY = 0.0f;
   XObj.fRotZ = 0.0f;

   // Position des Objektes
   XObj.vPos.x = XObj.matWelt._41 = fX;
```

```
   XObj.vPos.y = XObj.matWelt._42 = fY;
   XObj.vPos.z = XObj.matWelt._43 = fZ;

   return XObj;
   } // xUtil_init_XObjekt
/*----------------------------------------------------*/
```

Diese Funktion erstellt ein `XOBJEKT` mit den Rotationswinkeln von jeweils 0 auf allen Achsen. Die Position des Objektes kann über die Parameter der Funktion bestimmt werden und wird auch gleich in der Weltmatrix des Objektes mit vermerkt. Ach ja, das X File 3D Modell wird hier auch gleich mit geladen.

In einem echten 3D Spiel wird man dann noch Felder für Lebensenergie des Objektes, Wegpunkpfade, gewähltes Ziel usw. in diese Struktur packen. Daher hielt ich es für ratsam, diese Trennung zwischen Objekt und Modell auch hier gleich einzuführen, damit wir uns ja nix anderes angewöhnen.

Bewegung und Rotation des Objektes

Last not least werde ich noch schnell demonstrieren wie wir unser Objekt einfach drehen und verschieben können. Natürlich bedarf es dazu nicht mehr viel denn das notwendige Wissen haben wir uns ja schlauerweise schon im letzten Kapitel erarbeitet. Wir entwickeln also eine elegante Funktion mit deren Hilfe wir bequem die Rotation udn Verschiebung für ein Objekt festlegen können. Hier ist sie:

```
/**
 * Setzt das angegebene Objekt auf die Rota- und Pos der
 * Parameter die in den Objekt Elementen gespeichert werden.
 */
void xUtil_transformiere_XObjekt(float fRotX, float fRotY, float fRotZ,
                                 float fPosX, float fPosY, float fPosZ,
                                 XOBJEKT *pXObj) {
   D3DMATRIX matRotX, matRotY, matRotZ,
             matTemp, matWelt;

   // Rotationsmatrizen erzeugen
   xUtil_RotationsmatrixX(&matRotX, fRotX);
   xUtil_RotationsmatrixY(&matRotY, fRotY);
   xUtil_RotationsmatrixZ(&matRotZ, fRotZ);

   // Weltmatrix für Rotationen erstellen
   xUtil_MatrixMult(&matTemp, &matRotX, &matRotY);
   xUtil_MatrixMult(&matWelt, &matTemp, &matRotZ);
   (*pXObj).matWelt = matWelt;

   // Position verschieben
   (*pXObj).vPos.x = fPosX;
   (*pXObj).vPos.y = fPosY;
   (*pXObj).vPos.z = fPosZ;

   // Verschiebung in Weltmatrix speichern
   (*pXObj).matWelt._41 = pXObj->vPos.x;
   (*pXObj).matWelt._42 = pXObj->vPos.y;
   (*pXObj).matWelt._43 = pXObj->vPos.z;
   } // xUtil_transformiere_XObjekt
/*-----------------------------------------------*/
```

Diese Funktion können wir einfach in jedem Frame aufrufen und die absoluten Rotationswinkel und die absoluten Positionskoordinaten im Weltkoordinatensystem angeben die unser Objekt nun haben soll. Diese

Funktion erzeugt daraus die korrekte Weltmatrix und speichert sie mit dem Objekt. Es ist unter Umständen etwas zu viel Overkill (*unnötige Berechnung*) wenn wir immer alle Rotationsmatrizen berechnen und setzen. Manchmal wird ein Objekt gar nicht erst rotiert. Aber das ist ja hier nur eine Beispielimplementierung und daher sind wir mit Optimierungsgedanken hier nicht sehr verschwenderisch.

Rendern eines Beispielobjektes

Und nun, endlich, endlich, werden wir das Objekt auch rendern. Dazu habe ich konsequent auch wieder eine Funktion für die Einkapselung geschrieben, auch wenn die etwas lächerlich kurz ist. Wir müssen ja lediglich die Weltmatrix setzen und dann die Renderfunktion des 3D Modells des Objektes aufrufen:

```
void xUtil_render_XObjekt(XOBJEKT *pXObj) {
   // Verschiebung und Rotation für Device einstellen
   g_lpD3DDevice->SetTransform(D3DTS_WORLD, &pXObj->matWelt);

   g_lpD3DDevice->BeginScene();
   // 3D Modell des XOBJEKT rendern
   xMod_rendere(&pXObj->Modell);
   g_lpD3DDevice->EndScene();
   } // xUtil_render_XObjekt
```

Damit ist unsere Einkapselung komplett und wir können das `XOBJEKT` wie folgt verwenden.

```
// global
XOBJEKT xCat; // 3D Modell

// In Phase I: Erstelle ein Testobjekt
xCat = xUtil_init_XObjekt("cat.x", 5.0f, 3.0, 30.0);
if (!xCat.Modell.pMesh) {
   fprintf(Protokoll,"SpielInit: Testobjekt failed \n");
   g_blnBeenden = TRUE;
   return FALSE;
   }

// In Phase II: Hauptschleife
xUtil_transformiere_XObjekt(fRotX, fRotY, fRotZ,
                            xCat.vPos.x, xCat.vPos.y,
                            xCat.vPos.z, &xCat);
xUtil_render_XObjekt(&xCat);

// In Phase III: Objekte freigeben
if (xCat.Modell.pMesh)
   xMod_kill(xCat.Modell);
```

Für die Rotation des Objektes benutze ich übrigens die gleiche Technik wie bei dem Dreieck im letzten Kapitel. Die drei Variablen hier sind aber lokale Variablen der `WinMain()` Funktion. Als Position muss man natürlich jeweils die aktuelle Position des Objektes verwenden plus eventueller Änderungen in der Position. Aber ihr solltet sowieso wieder genug Boden gefunden haben, um ein wenig herum zu spielen. Ach, noch ein Wort. Es ist natürlich unschön die Rotation und Verschiebung eines Objektes direkt in der Hauptschleife zu machen. Normalerweise hat man eine eigenen Datei für Funktion die die künstliche Intelligenz und alle anderen Berechnungen eines solche `XOBJEKT` Objektes durchführen. Aber wie wollten ja alles so einfach und übersichtlich wie möglich halten, gell.

Und hier gibt es den Code des *gesamten* Tutorials zum Download: Projektdateien

---

Weiter geht's zum Kapitel 7...

SPIELEENTWICKLUNG MIT DIRECT3D IM - KAPITEL 7

von Stefan Zerbst

"You wanna dance?"
Duke Nukem 3D, Duke Nuke

Bewegung des Spielers

In diesem Kapitel werden wir nun endlich lernen wie wir es auch dem Spieler ermöglichen können, sich in unserem virtuellen Universum frei zu bewegen. Bisher haben wir ja nur gesehen wie wir 3D Modelle oder Objekte laden beziehungsweise erzeugen und bewegen können. So lange sich er Spieler aber nicht in seiner Umgebung bewegen kann, können wir wohl kaum von einem interessanten Programm, geschweigedenn einem Spiel, reden, oder?

Beginnen wir also mit etwas Basiswissen in Sachen 3D Engine bevor wir uns die Direct3D implementierungs-technischen Details ansehen. Es ist für den Computer unmöglich den Spieler von der Position (0,0,0) weg zu bewegen oder ihn gar rotieren zu lassen, egal um welche Achse. Das würde die Projektion von 2D auf 3D und noch einige Andere mathematische Formeln total durcheinander bringen und ist damit für uns unmöglich! Hm...in jedem 3D Spiel kann der Spieler sich aber bewegen, warum geht das da? Nun ist es mal wieder Zeit für mein Lieblingszitat an dieser Stelle: *Wenn der Prophet nicht zum Berg komm, dann muss der Berg eben zum Propheten kommen!* Für uns heisst das, dass wir mal wieder mathematisch etwas tricksen müssen.

Einstein hatte bereits in seiner Relativitätstheorie festgelegt, dass alles relativ ist **:-)** insbesondere aber die (*gleichförmig beschleunigte*) Bewegung. Für ein System aus zwei Objekten ist es vollkommen egal welches von beiden sich bewegt, oder ob sich beide bewegen. Jedes der beiden Objekte hat dieselbe Wahrnehmung, egal ob sich Objekt 1 von Objekt 2 weg bewegt und Objekt 2 ruht oder umgekehrt. Man denke nur an das typische Beispiel wenn man im Zug sitzt der am Bahnhof hält und ein Zug auf dem Nachbargleis steht. So lange man ausser den beiden Zügen nichts sehen kann (*den Bahnsteig o.a.*) ist es bei der Abfahrt schwer zu unterscheiden ob der eigene Zug fährt oder der Zug neben einem. Was für uns dabei wichtig ist, ist dass es einfach keinen Unterschied in der visuellen Wahrnehmung macht.

Viel Blabla, kommen wir zurück zum Thema. In der 3D Grafik nennt man den Spieler nicht Spieler, sondern **Kamera**. Wir sprechen also davon, dass wir jetzt die Kamera in unserer virtuellen Welt bewegen wollen, denn schliesslich ist die Ansicht am Monitor nicht immer das, was der Spieler durch die Augen seiner Spielfigur sehen würden. Denken wir nur an Aussenansichten bei Simulationen oder gar die verschiedenen Ansichten bei *Lara Croft*.
Okay, aber wir haben gerade gelernt, dass man die Kamera nicht in der virtuellen Welt bewegen oder rotieren kann. Der Trick ist nun folgender: Wenn der Spieler sich einen Schritt nach vorne bewegt dann unterscheidet sich das optisch in keiner Weise davon, dass die Kamera an ihrer Position (0,0,0) stehen bleibt und sich die **gesamte** Welt inklusive aller 3D Objekte, Modelle, des Terrains oder des Indoor Levels um einen Schritt auf den Spieler zu bewegt.

Cool, oder? Eine 3D Engine macht also folgendes. Der Spieler gibt an, dass es sich um x, y und z Einheiten auf den jeweiligen Achsen bewegen möchte und dass er sich um die Werte a, b und c auf den jeweiligen Achsen drehen will. Die Engine wird dann erst wie im letzten Kapitel besprochen die lokalen Koordinaten aller Objekte in Weltkoordinaten umrechnen um die Bewegung und Rotation der Objekte selbst durchzuführen. Danach werden alle Objekte in der Welt um die Werte -x, -y und -z auf den jeweiligen Achsen verschoben und um die Werte -a, -b und -c auf den jeweiligen Achsen rotiert.
Wir modifizieren also alle Objekte in der Welt mit der inversen Bewegung und der inversen Rotation des Spielers beziehungsweise der Kamera. Wenn der Spieler sich um 10 Grad nach rechts drehen will und fünf

Einheiten vorwärts gehen möchte, so bleibt er stehen und die Welt dreht sich um 10 Grad nach links und bewegt sich um fünf Einheiten auf den Spieler zu. So einfach ist das. Und noch ein Fachwort zum Schluss: Wir kennen ja bereits das lokale Koordinatensystem der Objekte, ebenso wie das Weltkoordinatensystem und das Projektionskoordinatensystem (=*Bildschirmkoordinaten*). Die Umrechnung der Objekte gemäss der Bewegung des Spielers nach der Verschiebung zu Weltkoordinaten und vor der Projektion in Bildschirmkoordinaten ist nun das vierte Koordinatensystem welches wir kennen lernen und man nennt dies die Kamerakoordinaten.

Was ist vDir, vUp und vRight?

Jetzt haben wir schon so viel über lokale Koordinatensysteme gesprochen da wollen wir sie uns auch einmal bildlich ansehen. Dazu bemühen wir die **Abbildung 1** um ein paar Definitionen festzulegen.

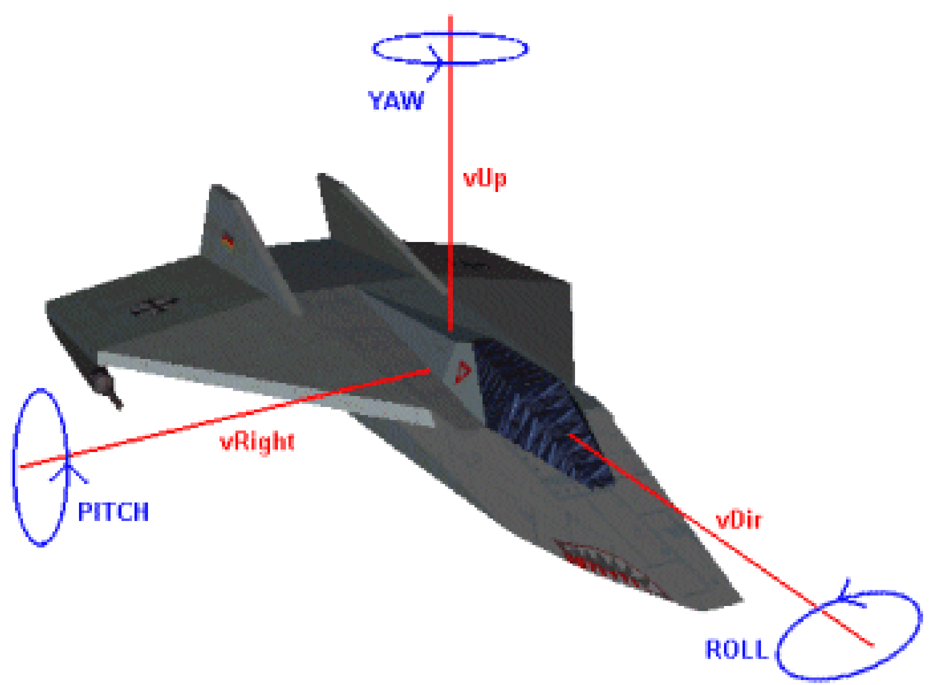


Abbildung 1: Drei Achsen und drei Rotationen

Jedes 3D Objekt in unserer virtuellen 3D Welt im Computer hat seine eigenen Koordinatenachsen, so auch die Kamera. Die drei Achsen bezeichnet man gewöhnlich als **Direction** (=*Richtungsvektor*), **Right** (=*Rechtsvektor*) und als **Up** (=*Hochvektor*). Diese Vektoren sind von elementarer Wichtigkeit für die korrekte Bewegung eines Objektes. Bisher wissen wir doch nur, wie wir ein Objekt auf den Weltachsen X, Y und Z verschieben können. Sobald wir unser Objekt aber ein Stückchen rotiert haben ist es nicht mehr an den Weltachsen ausgerichtet. Wenn wir unser Modell jetzt vorwärtsbewegen wollen dann wissen wir nicht, entlang welcher Achse wir das Modell verschieben müssen, denn die Z Achse ist das garantiert nicht mehr da das Modell etwas gedreht ist. Es würde sich sonst seitwärts bewegen.
Aus diesem Grund speichern wir für jedes 3D Objekt die oben gezeigten Achsen und immer wenn wir das Modell rotieren, dann rotieren wir diese Achsen mit dem Modell. So können wir jederzeit feststellen, in welche Richtung wir das Objekt bewegen müssen, da wir die rotierten Achsen haben. Um da wir gerade am Rotieren sind sehen wir uns auch gleich die Namen an, die man für die Rotation normalerweise verwendet. Drehen wir um die X Achse dann nennt man das **Pitch**, drehen wir um die Y Achse dann nennt man das **Yaw** und die Drehung um die Z Achse heisst **Roll**.

Im folgenden werden wir dieses System für die Kamera implementieren, dann wird das vielleicht etwas klarer. Dann sehen wir auch, wie wir ein Objekt tatsächlich in die richtige Richtung bewegen können, auch wenn es total *verdreht* im Raum ist.

Eine Datenstruktur für die Kamera

Was ist das Wichtigste für ein Programm? Richtig, die gut sitzenden Datenstrukturen. Wir beginnen unsere Arbeit also wie immer mit der Definition einer guten Datenstruktur wir unsere Zwecke:

```
typedef  struct  XKAMERA_TYP  {
    float     fYaw,
              fPitch,
              fRoll;
    D3DVECTOR vUp,
              vDir,
              vRight;
    D3DVECTOR vPos;
    } XKAMERA;
```

Diese Struktur bietet eigentlich keine Überraschungen. Wir speichern jeweils die drei notwendigen Drehwinkel und die dazugehörigen drei Achsen des Kameraobjektes. Zu guter letzt braucht die Kamera auch noch eine Position im 3D Universum. Und da wir gerade dabei sind ist sicherlich auch eine Funktion ganz nützlich mit deren Hilfe wir die Werte für unsere Kamera beim Start des Spiels einmalig initialisieren können:

```
// Globales Kameraobjekt
XKAMERA xKam;

/**
 * Diese Funktion initialisiert die Kamera auf die angegebene
 * Startposition und Drehwinkel von jeweils 0 RAD.
 */
void xKamera_aktivieren(float fPosX, float fPosY, float  fPosZ) {
   xKam.fPitch = 0.0f;
   xKam.fRoll  = 0.0f;
   xKam.fYaw   = 0.0f;
   xKam.vRight = xUtil_Vector(1.0f, 0.0f, 0.0f);
   xKam.vUp    = xUtil_Vector(0.0f, 1.0f, 0.0f);
   xKam.vDir   = xUtil_Vector(0.0f, 0.0f, 1.0f);
   xKam.vPos   = xUtil_Vector(fPosX, fPosY, fPosZ);
   } // xKamera_aktivieren
```

Hier passiert nix dramatisches, die Achsen des Kameraobjektes werden korrekt eingestellt und die Drehwinkel auf jeweils 0 RAD gesetzt. Ich habe übrigens noch eine Hilfsfunktion geschrieben mit deren Hilfe man einen Vektor schnell initialisieren kann, das spart lästige Tiparbeit. Hier ist sie:

```
inline D3DVECTOR xUtil_Vector(float fX, float fY, float fZ) {
   D3DVECTOR vVec;
   vVec.x = fX;
   vVec.y = fY;
   vVec.z = fZ;
   return vVec;
   }
```

Die Funktion tut zwar nix weiter als die Werte in ein `D3DVECTOR` Objekt zu speichern aber das spart halt einige Zeilen beim Tippen des Quelltexten und kommt unserer Faulheit zu gute. Okay, dann gehen wir gleich weiter im Text und sehen uns an, wie Direct3D das mit den Kamerakoordinaten realisiert.

Direct3D und die Kamerakoordinaten

Das ganze Geheimnis haben ich weiter oben schon verraten. Wir brauchen natürlich eine Matrix für die Kameradaten die wir dem Direct3D Device zuweisen. Die Transformation der Weltkoordinaten der Objekte in Kamerakoordinaten wird dann von dem Device übernommen und wir brauchen uns keine Gedanken darum zu

machen. Es ist natürlich nötig diese Kameramatrix immer dann neu zu erstellen und dem Device mitzuteilen wenn die Kamera sich bewegt. Das werden wir also in jedem Frame unseres Spiels machen (*wenn der Spieler die Möglichkeit hat die Kamera zu bewegen*). Und hier ist die Kameramatrix die das Device benötigt:

```
   vRight.x       vUp.x       vDir.x    0
   vRight.y       vUp.y       vDir.y    0
   vRight.z       vUp.z       vDir.z    0
-vPos*vRight   -vPos*vUp   -vPos*vDir   1
```

In die Matrix müssen wir also die entsprechenden Einträge unserer drei Achsenvektoren der Kamera eintragen. Die letzte Zeile der Kameramatrix besteht dann jeweils aus dem Negativen der Multiplikationen zweier Vektoren miteinander, um die Position der Kamera auch mit in Betracht zu ziehen. Wir können hier sehen, dass wir die entsprechenden Vektoren alle in unserer Kamerastruktur vorliegen haben. Das einzige was uns jetzt noch fehlt ist das Wissen, wie man zwei Vektoren miteinander multipliziert.
Das ist jedoch ein Kinderspiel und wir schreiben hier gleich eine entsprechende Funktion für unsere Hilfsbibliothek. Diesen Rechenvorgang bezeichnet man übrigens als Punktprodukt zweier Vektoren, was nix anderes als ein neues Wort für Multiplikation ist (*schliesslich kann man Skalare [normale Zahlen] auch als eindimensionale Vektoren interpretieren*). Das tut man deshalb weil es noch eine andere Art von Rechenoperation benutzt die man eventuell mit der Multiplikation verwechseln könnte. Und jene heisst Kreuzprodukt. Aber dazu kommen wir erst später.

```
inline float xUtil_Punktprodukt(D3DVECTOR *v1, D3DVECTOR *v2) {
   return (v1->x * v2->x +
           v1->y * v2->y +
           v1->z * v2->z);
   }
```

Easy, man multipliziert einfach die jeweiligen Koordinaten der beiden Vektoren miteinander, addiert die Ergebnisse der Multiplikation und hat damit das Endergebnis des Punktproduktes. Jetzt können wir also eine Funktion schreiben die uns aus den Daten der `XKAMERA` Datenstruktur die Matrix für das Direct3D Device erstellt:

```
/**
 * Nimmt die Rotationsdaten und Position der Kamera und erstellt
 * daraus die Kameramatrix und aktiviert sie für das D3D Device.
 */
void xKamera_Update(void) {
   D3DMATRIX matView;

   // Werte nur einmal berechnen!
   float fSinYaw   = sinf(xKam.fYaw),
         fCosYaw   = cosf(xKam.fYaw);
   float fSinPitch = sinf(xKam.fPitch),
         fCosPitch = cosf(xKam.fPitch);
   float fSinRoll  = sinf(xKam.fRoll),
         fCosRoll  = cosf(xKam.fRoll);

   // Igitt: Rotation von Hand:
   xKam.vRight.x = fCosYaw*fCosRoll + fSinYaw*fSinPitch*fSinRoll;
   xKam.vRight.y = fSinRoll*fCosPitch;
   xKam.vRight.z = fCosYaw*fSinPitch*fSinRoll - fSinYaw*fCosRoll;

   xKam.vUp.x = fSinYaw*fSinPitch*fCosRoll - fCosYaw*fSinRoll;
   xKam.vUp.y = fCosRoll*fCosPitch;
   xKam.vUp.z = fSinRoll*fSinYaw + fCosRoll*fCosYaw*fSinPitch;

   xKam.vDir.x = fCosPitch*fSinYaw;
```

```
   xKam.vDir.y = -fSinPitch;
   xKam.vDir.z = fCosPitch*fCosYaw;

   // Erzeuge die View Matrix für Direct3D
   matView._11 = xKam.vRight.x;
   matView._21 = xKam.vRight.y;
   matView._31 = xKam.vRight.z;
   matView._41 = -xUtil_Punktprodukt(&xKam.vPos, &xKam.vRight);
   matView._12 = xKam.vUp.x;
   matView._22 = xKam.vUp.y;
   matView._32 = xKam.vUp.z;
   matView._42 = -xUtil_Punktprodukt(&xKam.vPos, &xKam.vUp);
   matView._13 = xKam.vDir.x;
   matView._23 = xKam.vDir.y;
   matView._33 = xKam.vDir.z;
   matView._43 = -xUtil_Punktprodukt(&xKam.vPos, &xKam.vDir);
   matView._14 = 0.0f;
   matView._24 = 0.0f;
   matView._34 = 0.0f;
   matView._44 = 1.0f;

   // Aktiviere die View Matrix für das Device
   g_lpD3DDevice->SetTransform(D3DTS_VIEW, &matView);
   } // xKamera_Update
/*------------------------------------------------------------*/
```

Oh mein Gott...was passiert denn hier. Rollen wir den Code von hinten nach vorne auf. Ganz am Ende erstellen wir die gewünschte Matrix so wie oben dargelegt aus den entsprechenden Vektoren und den Punktprodukten. Dann setzen wir diese Matrix für das Direct3D Device mit der Funktion `SetTransform()` und dem Bezeichner `D3DTS_VIEW`. Damit haben wir die dritte (nach `D3DTS_WORLD` und `D3DTS_PROJECTION`) und letzte Verwendung dieser Funktion kennengelernt.
Nun zu dem Igitt-Code darüber. Im vorletzten Kapitel haben wir doch gelernt, wie man ein Objekt mittels der drei Rotationsmatrizen rotieren kann. Eigentlich könnten wir jetzt daherkommen und beispielsweise den `vRight` Vektor nacheinander mit den drei Matrizen multiplizieren um ihn entsprechend auf den drei Achsen zu rotieren. Aber es geht auch schneller. Statt drei Matrixrechnungen können wir das Ergebnis dieser Berechnungen in einer allgemeinen Formel vorher berechnen. Damit haben wir dann eine grosse Matrix in der alle drei Rotationsmatrizen gespeichert sind. Mit dieser einen Matrix müssen wir dann unseren Vektor einmal multiplizieren und haben alles was wir wollen.
Es gibt aber auch einen anderen Weg. Wir können den Vektor auf dem Papier erst mit einer, dann mit der zweiten und dann mit der dritten Matrix von Hand multiplizieren und sehen, was dabei heraus kommt. In diese allgemeinen Formeln müssen wir dann in jedem Frame nur noch die entsprechenden Rotationswinkel einsetzen. Blablabla...genau das ist in obigem Code passiert und damit wir auch alle sicher sein können dass wir das verstanden haben machen wir das mal beispielhaft für den `vRight` Vektor. Das sieht zwar unheimlich hässlich aus, spart uns aber einerseits in jedem Frame den Vektor durch drei Matrizenmultiplikationen zu hetzen was uns Geschwindigkeitszuwachs bringt. Andererseits lernen wir so endlich mal, auch etwas von Hand anzufassen und ein Feeling dafür zu bekommen **;-)**

Noch ein Detail bevor es los geht. Wenn wir einen Vektor mit einer Matrix multiplizieren dann kommt dabei wieder ein Vektor heraus. Wir haben bereits gelernt, wie wir Matrizen miteinander multiplizieren, oder? Ein Vektor ist nichts anderes als eine Matrix die nur eine Zeile hat, also ein Spezialfall der Matrixmultiplikation. FOlglich gelten auch hier die Regeln der Matrixmultiplikation. Also los geht's, wir haben die folgenden Dinge gegeben:

```
vRight = (1,0,0)

mRoll (Rota-Matrix z Achse Rollen) =
  cos(r) sin(r)  0    0
```

```
 -sin(r) cos(r)  0    0
    0      0     1    0
    0      0     0    1

vRight * mRoll = (1*cos(r) - 0*sin(r) + 0*0,
                  1*sin(r) + 0*cos(r) + 0*0,
                  1*0 + 0*0 + 1*0)

vNeu1 = (cos(r), sin(r), 0)
```

Wir haben nun den Vektor `vNeu1` erstellt, welcher dem Vektor `vRight` entspricht der um `r` RAD auf der z Achse gedreht wurde. Machen wir weiter, als nächstes binden wir die Rotation um die x Achse ein:

```
mPitch (Rota-Matrix x Achse Pitch) =
    1    0       0       0
    0   cos(p)  sin(p)   0
    0  -sin(p)  cos(p)   0
    0    0       0       1

vNeu1 * mPitch = (cos(r)*1 + sin(r)*0 + 0*0,
                  cos(r)*0 + sin(r)*cos(p) - 0*sin(p),
                  cos(r)*0 + sin(r)*sin(p) + 0*cos(p))

vNeu2 = (cos(r), sin(r)*cos(p), sin(r)*sin(p))
```

Okay, damit enthält `vNeu2` den Vektor `vRight` der um `r` RAD auf der z Achse gerollt wurde und um `p` RAD auf der x Achse gepitcht ist. Im letzten Schritt konkatenieren wir diesen Vektor mit dem Yaw auf der y Achse:

```
mYaw (Rota-Matrix y Achse Yaw) =
  cos(y) 0 -sin(y) 0
    0    1    0    0
  sin(y) 0  cos(y) 0
    0    0    0    1

vNeu2 * mYaw = (cos(r)*cos(y) + sin(r)*cos(p)*0 + sin(r)*sin(p)*sin(y),
                cos(r)*0 + sin(r)*cos(p)*1 + sin(r)*sin(p)*0,
                cos(r)*-sin(y) + sin(r)*cos(p)*0 + sin(r)*sin(p)*cos(y)

vErg.x = cos(r)*cos(y)+sin(r)*sin(p)*sin(y)
vErg.y = sin(r)*cos(p)
vErg.z = -sin(y)*cos(r)+sin(r)*sin(p)*cos(y)
```

Damit entspricht `vErg` also dem Vektor `vRight` der um `r` RAD auf der z Achse gerollt wurde, um `p` RAD auf der x Achse gepitcht und dann noch um `y` Grad auf der y Achse geyawed ist. So sind alle drei Rotationen in einem allgemeinen Vektor zusammen gefasst und man muss nur noch jeweils die entsprechenden Rotationswinkel angeben.

Vergleichen wir das Ergebnis mit dem obigen Quellcode so werden wir sehen, dass das genau die Formel ist die wir für `vRight` verwenden. Dasselbe Spielchen kann man nun für die beiden anderen Vektoren der Kamera machen, als kleiner Hinweis sei mir aber gestattet, dass die Reihenfolge bei der Matrixrotation für die endgültige Formel eine Rolle spielt. Diese Rechenoperation ist nicht *kommutativ* wie wir Mathematiker sagen **;-)**, wer die Formeln also nachrechnen will wird daher mit hoher Wahrscheinlichkeit auch andere Ergebnisse erhalten.

So zu guter Letzt haben wir noch die jeweiligen sin und cos Werte in Variablen reingerechnet, weil dies nicht gerade die schnellsten Rechenoperationen sind die ein Computer durchführen kann. Da wir aber die Werte mehrfach in den Formeln verwenden berechnen wir sie einmal am Anfang und speichern sie zur Wiederverwendung. Klasse! Jetzt haben wir so viel über Optimierung gelernt und unsere Rechenfähigkeiten noch ein wenig trainiert, dass es für ein Kapitel genug Arbeit ist. Jetzt kann der Spass wieder Einzug halten.

Bewegung und Rotation der Kamera

Keine Panik, von jetzt ab ist der Rest des Codes so einfach wie...genau, Pfannkuchen essen. Beginnen wir damit unsere Kamera um eine gewisse Anzahl an RAD zu drehen:

```
/**
 * Rotiert die Kamera um angegebenen Winkel auf den 3 Achsen.
 */
void xKamera_rotieren(float fRotX, float fRotY, float fRotZ) {
   xKam.fPitch += fRotX;
   xKam.fYaw   += fRotY;
   xKam.fRoll  += fRotZ;
   }
/*----------------------------------------------------------*/
```

**Hehe* wenn das nicht einfach ist. Wir addieren die entsprechenden Rotationswerte einfach zu unserem globalen Kameraobjekt. Damit haben wir für unser Kamera gespeichert um wie viele RAD sie auf den entsprechenden Achsen gedreht ist und unsere eben so ausführlich behandelte Funktion wird diese Werte dann verwenden um die korrekte Rotationsmatrix zu erstellen.

Kommen wir nun dazu wie wir die Kamera bewegen können. Ebenso wie es drei mögliche Rotationen gibt, so gibt es natürlich auch drei Bewegungen für die Kamera. Sie kann sich entlang des Richtungsvektors bewegen, entlang des Rechtsvektors (*seitlich gleiten*) und natürlich entlang des Hoch Vektors (*hoch und runter gleiten*). Und damit haben wir auch schon das Rätsel gelöst wie wir die Kamera bewegen. Wir müssen die Position der Kamera einfach entlang des entsprechenden Vektors der Kamera verschieben:

```
/**
 * Verschiebt die Kamera um die angegebenen Werte auf den
 * 3 Achsen rechts/links, hoch/runter und vor/zurück.
 */
void xKamera_verschieben(float fX, float fY, float fZ) {
   // Entlang der x Achse (rechts vs links)
   xKam.vPos.x += xKam.vRight.x * fX;
   xKam.vPos.y += xKam.vRight.y * fX;
   xKam.vPos.z += xKam.vRight.z * fX;

   // Entlang der y Achse (hoch vs runter)
   xKam.vPos.x += xKam.vUp.x * fY;
   xKam.vPos.y += xKam.vUp.y * fY;
   xKam.vPos.z += xKam.vUp.z * fY;

   // Entlang der z Achse (vor vs zurück)
   xKam.vPos.x += xKam.vDir.x * fZ;
   xKam.vPos.y += xKam.vDir.y * fZ;
   xKam.vPos.z += xKam.vDir.z * fZ;
   } // xKamera_verschieben
/*------------------------------------------------------*/
```

Et voiá, wir haben damit alles zusammen was wir für das Umsetzen der Bewegung des Spielers in unserer 3D Welt benötigen. Fehlt nur noch ein winziges Detail, nämlich das Einbauen der Funktionen in unseren Code. Zuerst erzeugen wir die Kamera in der Funktion `Spiel_Initialisieren()`:

```
// Aktiviere die Kamera mit Position
xKamera_aktivieren(0.0f, 0.0f, 0.0f);
```

In jedem Frame des Spiels müssen wir dann auch entsprechend die Kamera mit den neuesten Daten updaten,

um die View Matrix zu erstellen und für das Device zu aktivieren. In dem entsprechenden `case` der Hauptschleife kommt also der folgende Aufruf:

```
// Bewegung des Spielers umsetzen
xKamera_Update();
```

Yo Baby, damit sind wir so weit komplett. Sehen wir uns nun noch schnell an wie wir die Eingaben der Tastatur abfangen und so bearbeiten, dass wir das Drücken einer bestimmten Taste mit einer Bewegung der Kamera verbinden.

Abfrage der Tastatur

Und das ist nun wirklich kinderleicht. In der Funktion `WindowProc()` haben wir doch bereits eine Tastaurabfrage eingebaut. Im `case` von `WM_KEYDOWN` haben wir bisher zwar nur die Escape Taste `VK_ESCAPE` stehen, aber das können wir beliebig erweitern. Das folgende Codestück zeigt beispielsweise wie wir die Kamera durch Drücken der Pfeiltasten rechts oder links herum drehen können:

```
case VK_LEFT: {
   xKamera_rotieren(0.0f, -0.1f, 0.0f); // Links drehen
   return 0;
   } break;
case VK_RIGHT: {
   xKamera_rotieren(0.0f, 0.1f, 0.0f);  // Rechts drehen
   return 0;
   } break;
```

Wenn wir die normalen Buchstabentasten abfragen wollen dann machen wir das beispielsweise für das gleiten nach rechts oder links wie folgt:

```
case 'x':
case 'X': {
   xKamera_verschieben(-0.1f, 0.0f, 0.0f); // Links gleiten
   return 0;
   } break;
case 'c':
case 'C': {
   xKamera_verschieben(0.1f, 0.0f, 0.0f);  // Rechts gleiten
   return 0;
   } break;
```

Eigentlich sollte es unter Windows auch die Bezeichner `VK_X` und so weiter geben...gibt es aber bei mir irgendwie nicht. Also fragen wir die Buchstaben wie eben gezeigt ab. Nun sind wir aber wirklich am Ende und Ihr könnt dann im Projekt nachsehen, welche Tasten ich noch mit Funktionen belegt habe um alle Bewegungen der Kamera zu demonstrieren. Das Projekt beinhaltet noch die Cat aus dem letzten Kapitel die am Schirm gerendert wird. Schliesslich kann man bei einem absolut leeren Screen nicht sehr gut sehen, wie man sich drehen und bewegen kann.

Zum Abschluss noch eine gute Nachricht. Das nächste Kapitel besteht nur aus reinem *Pfannkuchenessen** und bietet einen kleinen Auffrischer in der Arbeit mit Pointern sowie ein paar weitere 3D Mathematik Goodies die wir im übernächsten Kapitel dringend brauchen werden, mit denen wir uns aber vorher bewaffnen werden da das übernächste Kapitel eh schon lang genug ist. Dann geht es nämlich endlich um das Rendern von 3D Indoor Level für unseren eigenen Quake Clone...

*Wer als erster eine Nachricht unter der Topic "Pfannkuchen" in mein Forum postet und errät woher genau ich das Pfannkuchenzitat habe und wer es geprägt hat der bekommt ein Exemplar meines zweiten Buches gratis sobald es fertig ist. eMails gelten **nicht** und es ist ausschliesslich die von **mir** als "Zerbie" gestartete Topic zu verwenden, der Rechtsweg ist ausgeschlossen, usw.

Und hier gibt es den Code des ***gesamten*** Tutorials zum Download: Projektdateien

Weiter geht's zum Kapitel 8...

von Stefan Zerbst

"Rydell had a theory about virtual real estate. The smaller and cheaper the physical site of a given operation, the bigger and cheesier the website. According to this theory, Selwyn FX was probably operating out of rolled-up newspaper."
All Tomorrow's Parties, by William Gibson

Pointer, Arrays und 3D Goodies

Die Schreckensmeldung gleich zuerst: In diesem Kapitel wird es kein Demoprogramm und keinen Download geben. Dieses Kapitel erfüllt einzig und allein den Zweck uns auf das vorzubereiten was da kommen wird. Und das ist das nächste Kapitel wo wir unseren eigenen Quake Clone programmieren werden....hey, hiergeblieben! Erst solltet Ihr dieses Kapitel lesen, denn hier werden einige Techniken besprochen die ich im nächsten Kapitel voraussetzen und nicht mehr erklären werde. Dann können wir uns nachher auch das wesentliche hinter einer Quake-like Engine konzentrieren.

Zuerst geht es hier darum die Arbeit mit Pointern und Arrays noch einmal etwas zu erläutern da ich mal davon ausgehe dass hier auch viele Anfänger mitlesen für die das noch etwas wackeliger C Boden ist. Danach geht es darum ein paar weitere mathematische Grundlagen zu liefern die wir für 3D und speziell für Direct3D benötigen. Es geht von dem Kreuzprodukt über den View Frustrum bis hin zu Bounding Box Culling. Diese Techniken braucht man natürlich nicht nur für eine 3D Indoor Engine, sondern für jede Art von 3D Programm, Weiterlesen lohnt sich also (*wie immer*).

Pointer und Arrays, Cousins in C

Ich gehe mal stark davon aus dass wir alle wissen was ein Array in C ist und wie man es verwenden kann. Nun widmen wir uns aber dem Problem warum wir in fortgeschrittenen Programmen immer weniger Arrays verwenden sollten und lieber auf Pointer zurückgreifen. Um der Motivation willen sehen wir uns Beispiele für eine 3D Indoor Engine an. **:-)**
Unsere Leveldaten sind nichts anderes als eine Liste von Vertices aus denen sich die Wände, Decken und Böden unseres Levels zusammensetzen,oder. Also brauchen wir ein Array für diese Vertices:

```
D3DVERTEX avVertexliste[10000];
```

Okay, wie wir sehen haben wir bereits hier ein Problem. Wir haben erst mal 10'000 Felder in dem Array angegeben aber woher wissen wir ob das reicht? Ein Array muss immer statisch allokierten Speicher verwenden, dass heisst nix anderes als dass wir zur Compile Zeit (*also wenn wir das Programm schreiben*) bereits eine Zahl wie 10'000 für die Arraygrösse angeben müssen und keine Variable wie beispielsweise `nAnzahl`. Probiert das ruhig mal mit dem Compiler aus, er wird dann einen Fehler auswerfen.
Okay, höre ich da jemand sagen: "ist doch kein Problem." Wenn wir unsere Leveldaten laden und die Vertices aus der Leveldatei in das Array laden dann dürfen wir halt auf maximal 10'000 Felder des Arrays zugreifen, ab dem 10'001 Feld laufen wir Gefahr eine ungültige Speicheradresse zu treffen und das Programm damit zu crashen. Also tun wir folgendes:

```
int g_Zähler = 0;

while (Leveldatei != End_of_File) {
```

```
   if (g_Zähler >= 10'000) {
      fprintf(Protokoll, "Fehler, zu viele Vertices");
      break;
      }

   avVertexliste[g_Zähler] = Lies_Vertex_aus_Datei();
   g_Zähler++;
   } // while
```

Super, damit verhindern wir dass unser Programm abstürzt aber wir schränken gleichzeitig unsere Leveldaten auf einen beliebigen Wert ein. Wenn es im Level mehr als 10'000 Vertices gibt so werden diese von unserer Engine einfach ignoriert. Das ist nicht sehr nett. Was tut ein schlauer Programmierer also? Er setzt den Wert für das Array so hoch, dass es auch für grosse Level reicht, beispielsweise auf 999'999. Aber auch das ist nicht ohne Tücken, vor allem ist es aber unschön. Wenn unser Level nun 50'000 Polygone hat dann haben wir fast 950'000 mal Speicher in der Grösse von `sizeof(D3DVERTEX)` für unser Programm blockiert der aber nie genutzt wird. Wenn wir das an mehreren Stellen in unserem Programm so machen dann geht uns selbst bei 128 MB RAM der Speicher aus bevor wir überhaupt die erste Textur laden können. Die Lösung bieten also Pointer an:

```
D3DVERTEX *pvVertexliste=NULL;
```

Einem Pointer können wir zur Laufzeit des Programms Speicher zuweisen, also nicht schon dann wenn wir das Programm tippen, sondern erst dann wenn das Programm läuft. Wir können also tatsächlich den benötigten Speicher in einer Variablen halten:

```
nAnz_Verts = Lade_Anzahl_aus_Leveldatei();

pvVertexliste = (D3DVERTEX*)malloc(nAnz_Verts*sizeof(D3DVERTEX));
if (!pvVertexliste)
   fprintf(Protokoll,"Fehler: Nicht genug Speicher\n");
   return;
   }
```

Woah...slow down. Als erstes laden wir die Anzahl der wirklich verwendeten Vertices aus der Leveldatei. Beispielsweise kann eine Leveldatei als ersten Eintrag eine Zahl anbieten die die Anzahl der Vertices in der Datei angibt. Diese Zahl laden wir dann zur Laufzeit des Programms und die Funktion `malloc()` dient dann dazu, uns Speicher für den Pointer `pvVertexliste` bereitzustellen. Wir müssen der Funktion einfach nur sagen wie viel Speicher. Das errechnen wir als die Grösse des benötigten Datentyps und die Anzahl der Elemente die wir in den Speicherbereich laden wollen. Der Rückgabewert der Funktion ist ein `void` Pointer auf den bereitgestellten Speicherbereich im RAM des Computers, wir casten diesen Pointer also in den von uns gewünschten Typ. Danach ist es wichtig zu prüfen, ob der Funktionsaufruf erfolgreich war. Das `if` Statement prüft dann, ob der Pointer immer noch den Wert `NULL` hat, das ist dann ein Fehler da kein Speicher bereitgestellt werden konnte.

Alles klar, jetzt haben wir den notwendigen Speicher. Wie aber laden wir unsere Daten in diesen Pointer? Das geht denkbar einfach wie wir hier sehen können:

```
D3DVERTEX *pVert = pvVertexliste;

for (int i=0; i<nAnz_Verts; i++) {
   *pVert = Lies_Vertex_aus_Datei();
   pVert++;
   } // for
```

Wir erschaffen uns einfach einen zweiten Zeiger den wir auf den Anfang des gültigen Speicherbereiches setzen in den wir unsere Vertices laden wollen. Dann durchlaufen wir alle Vertices in der Datei und speichern immer einen Vertex an der Speicheradresse auf die `pVert` zeigt. Eben dieses `pVert` können wir dann durch

`++` um genau `sizeof(D3DVERTEX)` verschieben, so dass wir den nächsten Vertex an die nächste Stelle im Speicherbereich laden. Für den einen oder anderen ist das eventuell nix neues, aber ich dachte mir dass wir das besser noch einmal wiederholen. Und hier noch ein offenes Geheimnis: Arrays sind intern in C nichts anderes als solche Pointer (*nur mit statischem Speicher*) daher ist es auch gültig einen Pointer als Array anzusprechen und wir können denselben Code von eben auch wie folgt schreiben:

```
D3DVERTEX *pVert = pvVertexliste;

for (int i=0; i<nAnz_Verts; i++) {
   pVert[i] = Lies_Vertex_aus_Datei();
   } // for
```

In diesem Fall können wir sogar `pVert` weglassen und direkt `pvVertexliste` verwenden. Oben haben wir `pVert` nur deshalb eingeführt, damit wir beim Verschieben des Pointers nicht die originale Speicheradresse `pvVertexliste` verlieren wo unsere Daten beginnen.

Yeah...cool. Aber es wird noch besser. Manche Programmierer sind faul, habe ich mal gehört. Das kann sogar so weit führen, dass unsere Leveldatei gar keine Zahl liefert wie viele Vertices sie enthält, sondern diese einfach gleich alle auflistet. In diesem Fall wissen wir erst nach dem Einlesen aller Vertices wie viele es wirklich waren. Toll, damit sind wir wieder bei demselben Problem wie vorhin gelandet, oder?
Naja, nicht direkt. Das tolle an Pointern ist die Tatsache, dass ihr Speicher **dynamisch** ist. Wir können den Speicher zur Laufzeit des Programms nicht nur allokieren (*bereitstellen*), sondern auch wieder freigeben (*dann wird der Pointer ungültig, dynamisch allokierten Speicher muss man aber immer am Ende eines Programms durch die* `free()` *Funktion freigeben um die allokierten Ressourcen des Programms freizumachen, sonst bleibt das Betriebssystem irgendwann ohne RAM*). Was aber für uns interessant ist, dass ist die Möglichkeit die Grösse des Speicherbereichs zu ändern während das Programm läuft. Super, sehen wir uns das mal genau an:

```
#define ANZ_VERTS 10000

int        g_Zähler = 0;
D3DVERTEX *pvVertexliste=NULL;

pvVertexliste = (D3DVERTEX*)malloc(ANZ_VERTS*sizeof(D3DVERTEX));
if (!pvVertexliste)
   fprintf(Protokoll,"Fehler: Nicht genug Speicher\n");
   return;
   }

while (Leveldatei != End_of_File) {
   pvVertexliste[g_Zähler] = Lies_Vertex_aus_Datei();
   g_Zähler++;

   if (g_Zähler % ANZ_VERTS == 0) {
      pvVertexliste = (D3DVERTEX*)realloc(pvVertexliste,
                                          (g_Zähler+ANZ_VERTS) *
                                          sizeof(D3DVERTEX));
      if (!pvVertexliste)
         fprintf(Protokoll,"Fehler: Nicht genug Speicher\n");
         return;
         }
      }
   } // while
```

Die Wunderfunktion `realloc()` sorgt dafür, dass der Speicherbereich an der angegebenen Adresse auf die angegebene Grösse erweitert oder verkleinert wird. Das Gute daran ist, dass alle Daten in dem Speicherbereich erhalten werden wenn der Bereich vergrössert wird. Bei einer Verkleinerung fallen nur die

Daten raus die nicht mehr passen, der Rest bleibt ebenfalls erhalten. Wie wir sehen initialisieren wir den Speicher auf 10'000 Einträge. Wenn wir diese Anzahl erreicht haben und der Speicher damit zuende ist, dann müssen wir mehr Speicher reallokieren. Wir benutzen die Modulofunktion `%` um festzustellen wann dieser Zeitpunkt erreicht ist, denn diese Funktion liefert immer dann 0 wenn `g_Zähler / 10'000` eine ganze Zahl ist (`g_Zähler` *ist damit ein VIelfaches von 10'000*). Dann erweitern wir den Speicherbereich um weitere 10'000 Einheiten.
Man muss aber bedenken, dass der Computer dann automatisch eine Stelle im Speicherbereich sucht, an der der gesamte Speicher des Pointers Platz findet. Es ist also sehr wahrscheinlich dass die Adresse des Pointers im RAM geändert wird und sollten wir noch andere Variablen haben die auf die ursprüngliche Adresse zeigen so sollte man immer annehmen dass diese damit ungültig geworden sind.

Super, damit haben wir dann nur noch das Problem, dass wir im Worst-Case Szenario `ANZ_VERTS-1` Einheiten Speicher zu viel allokieren. Haben wir beispielsweise 10'001 Vertices in unserer Leveldatei so allokieren wir durch obige Methode 20'000 Einheiten Speicher. Aber das Problem umgehen wir einfach, indem wir <u>nach</u> dem Einlesen aller Vertices den Speicher an genau die eingelesene Anzahl anpassen:

```
pvVertexliste = (D3DVERTEX*)realloc(pvVertexliste,
                                    g_Zähler * sizeof(D3DVERTEX));
```

Damit sitzt der Speicherbereich ganz exakt und wir müssen weder Vertices ablehnen noch allokieren wir zuviel Speicher den wir gar nicht benötigen. Daneben haben wir auch gleich noch gelernt, dass Pointer und Arrays im Grunde genommen das gleiche sind, sie unterscheiden sich nur hinsichtlich der Art des Speichers (*statisch oder dynamisch*). Und noch eine Definition aus ANSI C:

`&avVertexliste[0]` ist identisch mit `avVertexliste` man kann die Bezeichner `&` und `[0]` damit weglassen und greift aber dennoch auf die Speicheradresse zu an der das Array beginnt.

So, das war glaube ich das Wichtigste was ich zu Arrays und Pointer noch loswerden wollte. Jetzt machen wir uns an die 3D Goodies die ich versprochen habe.

Das Kreuzprodukt

Wie kennen ja bereits das Punktprodukt zweier Vektoren (`V1*V2`). Nun gibt es aber noch das sogenannte Kreuzprodukt zweier Vektoren (`V1xV2`) (*sprich V1 kreuz V2*). Das Punktprodukt liefert gewisse Informationen über den Winkel in dem beide Vektoren zueinander stehen in Form eines Skalars (*normale Zahl*) während das Kreuzprodukt etwas anderes feines tut. So lange die beiden Vektoren nicht parallel sind liefert das Kreuzprodukt einen Vektor der orthogonal (*rechtinklig*) zu den beiden anderen Vektoren ist. Wie wir später noch sehen werden ist es von elementarer Wichtigkeit in der 3D Welt solche rechtwinkligen Vektoren zu berechnen. Schauen wir uns erst mal die Formel an, wie man ein Kreuzprodukt aus dreidimensionalen Vektoren berechnet. Man muss nicht verstehen wir diese Rechnung zustande kommt, man muss einfach nur wissen wie es funktioniert:

```
inline D3DVECTOR xUtil_Kreuzprodukt(D3DVECTOR &v1,
                                    D3DVECTOR &v2) {
   D3DVECTOR vErgebnis;
   vErgebnis.x = v1.y * v2.z - v1.z * v2.y;
   vErgebnis.y = v1.z * v2.x - v1.x * v2.z;
   vErgebnis.z = v1.x * v2.y - v1.y * v2.x;
   return vErgebnis;
   }
```

Nun gut, das ist etwas komplizierter als das Punktprodukt aber auch nicht so viel. Kümmern wir uns nun darum was wir mit diesem Kreuzprodukt so alles machen können.

Normalenvektoren und Normalisierung

Einen Vektor der zu einer Fläche senkrecht steht bezeichnet man als **Normalenvektor**. Solche Vektoren haben in der 3D Grafik verschiedene spezielle Nutzen, insbesondere in der Berechnung der Beleuchtung der Fläche da man die Lichtintensität einer Fläche aus dem Winkel des Normalenvektors und des einfallenden Lichts berechnen kann.
Sehen wir uns einmal grafisch in der **Abbildung 1** an, wie man den Normalenvektor zu einer Fläche aus deren Vertices berechnen kann. Es überrascht uns jetzt auch nicht mehr, dass wir hier mit dem Kreuzprodukt arbeiten müssen.

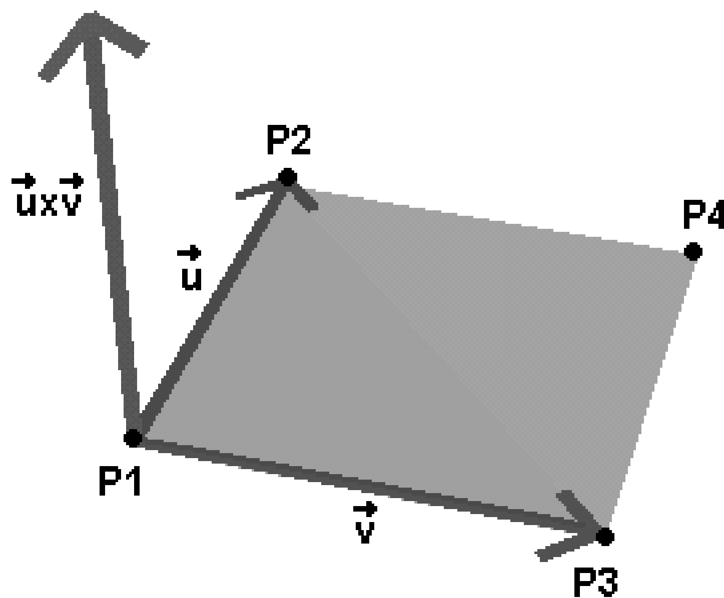


Abbildung 1: Der Normalenvektor einer Fläche

Wir konstruieren aus der Fläche einfach zwei Vektoren `U` und `V` die in der Fläche liegen. Das ist denkbar einfach, wir müssen dafür nur die Endkoordinaten der Vektoren (*P2, bzw P3*) von den Startkoordinaten abziehen (*beide male P1*). Dann bilden wir schnell das Kreuzprodukt `UxV` und erhalten als Ergebnis den Vektor der senkrecht zur Fläche steht. Da ist schon der ganze Trick. Es ist übrigens keine Voraussetzung dass `U` und `V` ebenfalls senkrecht zueinander stehen. In einem beliebigen Polygon können wir aus drei beliebigen Punkten die beiden Vektoren erstellen und erhalten dann den Normalenvektor des Polygons. Die beiden Vektoren `U` und `V` dürfen nur nicht parallel zueinander sein.

Und nun die Überraschung: Das **Normalisieren** von Vektoren hat nix mit dem Normalenvektor zu tun, auch wenn uns der Name Ähnlichkeiten nahelegen will. Wenn wir einen (*beliebigen*) Vektor normalisieren dann heisst das nichts anderes als dass der Vektor auf die Länge 1 gebracht wird und als normalisiert bezeichnet werden kann. Daraus ergeben sich zwei Fragen: (a) wie kann man die Länge eines Vektors berechnen und (b) wie kann man diese Länge auf 1 bringen und dabei die Ausrichtung des Vektors erhalten?

Widmen wir uns erst mal der ersten Frage. Wer kennt nicht Pythagoras berühmte Formel `a*a+b*b=c*c`!? Genau so kann man die Länge eines Vektors berechnen die man allgemeinhin auch **Betrag** des Vektors nennt:

```
fBetrag = sqrt(v1.x*v1.x + v1.y*v1.y + v1.z*v1.z);
```

Im 2D Raum kann man einen Vektor beispielsweise komponentenweise in ein Koordinatensystem eintragen. EInmal den Vektor selber und dann jeweils seine LÄnge auf der x und y Achse. Damit erhält man ein Dreieck mit rechtem Winkel und kann daher Pythagoras Formel anwenden. Im 3D Bereich funktioniert das natürlich analog. Ich sage nur: Pfannkuchen. Und nun zur zweiten Frage. Nachdem wir den Betrag eines Vektors haben bringen wir seine Länge auf 1 indem wir alle seine drei Komponenten durch die Länge teilen. Auch hier kann man wieder zu den eindimensionalen Vektoren (*aka Skalare aka normale Zahlen*) greifen um sich das zu veranschaulichen. Die Länge eines eindimensionalen Vektors ist natürlich identisch mit dem Betrag seiner einen Komponente. Der Vektor (5) ist beispielsweise 5 Einheiten lang. Wenn wir (5) auf die Länge 1 bringen wollen dann teilen wir die Zahl einfach durch sich selbst denn 5/5 = 1 oder?

```
inline D3DVECTOR xUtil_Normalisieren(D3DVECTOR *vVector) {
   float fBetrag = sqrtf((*vVector).x * (*vVector).x +
                         (*vVector).y * (*vVector).y +
                         (*vVector).z * (*vVector).z);
   (*vVector).x /= fBetrag;
   (*vVector).y /= fBetrag;
   (*vVector).z /= fBetrag;
   return (*vVector);
   }
```

Ist immer noch jemand der Meinung dass Vektorrechnung und 3D Grafik schwer ist? Wie man unschwer erkennen kann läuft in der 3D Mathematik und speziell in der Vektorrechnung alle auf simples Addieren und Subtrahieren hinaus, ist also Stoff aus der Grundschule **:-)**

Ebenen in 3D

Nun sind wir bei einem Thema gelandet das etwas abstrakt ist. Aber haltet durch denn 3D Ebenen sind unheimlich wichtig wenn wir das nächste Kapitel verstehen wollen. Das Problem ist hier, dass wir nun mit dem schicken Symbol ∞ zu tun bekommen. Das ist eine Acht die ohnmächtig geworden und umgekippt ist und sie symbolisiert in der Mathematik den Begriff **unendlich**. Und nun kommen wir dazu was eine Ebene ist:

Eine Ebene ist eine Fläche (*hier im 3D Raum*) die sich unendlich weit ausdehnt und unendlich dünn ist.

Hm...komisches Ding. Um uns das etwas plastischer vorzustellen nehmen wir mal ein Blatt Papier zur Hand. Dies erfüllt mehr oder weniger bereits das Kriterium unendlich dünn zu sein, obwohl das unter dem Elektronenmikroskop wohl anders aussähe. Na jedenfalls reicht uns die Approximation zu sagen dass das Papier dünn genug ist um als Anschauungsobjekt zu dienen. Aber nun zur zweiten Eigenschaft einer Ebene, sie ist unendlich gross. Dazu müssen wir uns vorstellen dass unser Blatt Papier unendlich gross ist, sich also bis an die Grenzen des Universums ausdehnt. Mann, was wir darauf alles zeichnen könnten **:-)**

Genug rumgekaspert, zurück zur harten Welt der Mathematik. Sehen wir uns einmal an wie man so eine Ebene mathematisch darstellen kann. Die entsprechende Formel lautet:

```
V*N + d = 0
```

Der Vektor `N` ist der Normalenvektor der Ebene, also ein Vektor der senkrecht zu der Ebene steht. Scrollt noch mal ein Stück hoch und seht Euch **Abbildung 1** an. Das dort gezeigte Polygon in Form eines Quadrats können wir uns als ein kleines Stück der Ebene vorstellen und der Normalenvektor ist dort auch bereits eingezeichnet. Der Skalar `d` ist einfach eine Fliesskommazahl die die Entfernung der Ebene zum Ursprung, also dem Nullpunkt (0,0,0), des Koordinatensystems angibt.
Aha, aber eine Ebene ist doch unendlich gross, an welchem Punkt der Ebene messen wir also die Entfernung zum Ursprung so dass die Ebenengleichung eindeutig bleibt? Die Antwort ist denkbar simpel, wir nehmen den Punkt an dem die Entfernung der Ebene zum Ursprung am kleinsten ist. Diesen Punkt haben wir genau dann wenn wir vom Ursprung den Normalenvektor der Ebene so oft aneinanderlegen bis wir die Ebene treffen. Dies ist die kürzeste Distanz die die Ebene zum Ursprung hat. Der Wert von `d` ist übrigens negativ wenn der Normalenvektor der Ebene vom Ursprung wegzeigt und positiv anderenfalls.

Moment, da war doch noch der Vektor `V` in der Formel? Dies ist jeder beliebige Punkt der in der Ebene liegt. Wir können uns also beispielsweise einfach einen Punkt ausdenken und ihn in die Formel einsetzen. Wenn das Ergebnis stimmt, also der Wert 0 herauskommt, dann ist unser Punkt rein zufällig ein Punkt der in der Ebene liegt, anderenfalls ist der Punkt nicht in der Ebene. Wir haben natürlich sofort gesehen, dass wir hier das Punktprodukt aus dem Vektor und dem Normalenvektor bilden. Das merken wir uns für später denn das werden wir weiter unten gleich noch brauchen. Hier aber erst noch mal ein Bild zum besseren Verständnis:

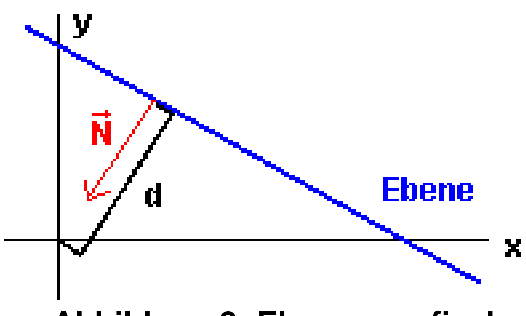


Abbildung 2: Ebenen grafisch

Wie das Bild zeigt kann man Ebenen nur schwer grafisch darstellen. Wir blicken hier genau auf die Seite einer Ebene die eigentlich unendlich dünn sein sollte (*blaue Linie*). Der roten Normalenvektor zeigt hier zum Ursprung hin und wir sehen auch wie wir die Distanz d messen.

Und hier ist unsere Datenstruktur für eine Ebene:

```
typedef struct XEBENE_TYP {
   float     fDistanz;   // Distanz zum Ursprung
   D3DVECTOR vNormal;    // Normalenvektor der Ebene
   } XEBENE;
```

Jetzt wissen wir ganz gut, was Ebenen sind und mit welchen Formeln das verbunden ist. Da das aber sehr abstrakt ist sollten wir noch mal einen Blick darauf riskieren wozu Ebenen in der 3D Spieleentwicklung unter anderem verwendet werden können (*müssen*).

Der View Frustrum

Wir haben uns ja schon mit diversen Berechnungen herumgequält, beispielsweise der Umrechnung von Welt- in Kamerakoordinaten und der Projektion von 3D auf 2D Koordinaten in den Viewport. Die **Abbildung 3** führt uns noch einmal vor Augen was dabei so herauskommt.

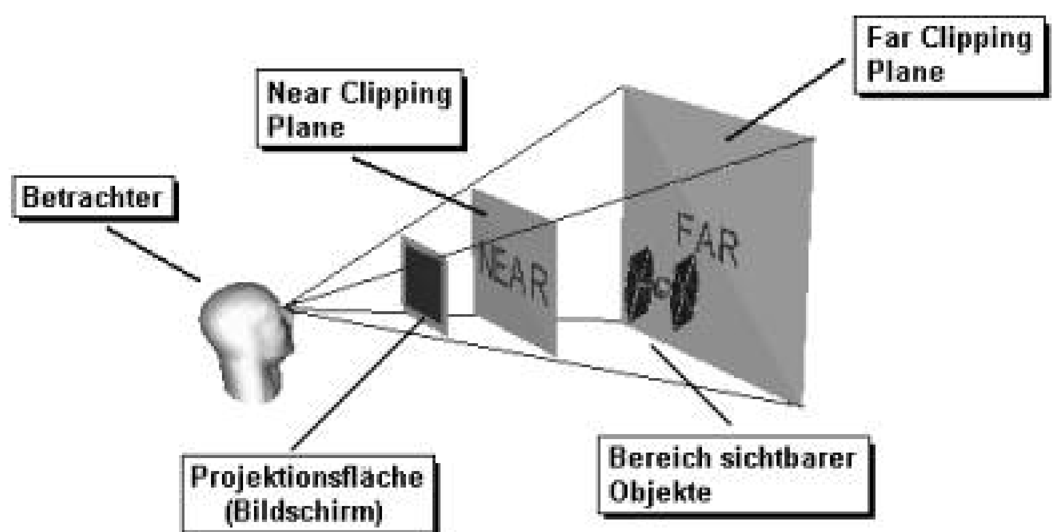


Abbildung 3: Der View Frustrum

Der Spieler blickt auf Fläche auf die die 3D Szene projiziert wird, in der Regel ist dies der Computermonitor. Wie man sehr gut sehen kann begrenzt der Monitor das Sichtfeld des Betrachters auf eine Art Pyramide. Je weiter die Entfernung in die der Spieler blickt desto grösser wird der Raum den er sehen kann. Sehr nah vor einem Fenster kann man beispielsweise nur wenige Meter nach rechts und links sehen, wenn man aber weit in die Ferne schaut dann kann man viele Kilometer des Horizontes sehen.

Es ist aber eine Verschwendung von Rechenzeit alle Objekte zu rendern die sehr weit entfernt sind. In der Regel sind diese Objekte so klein dass sie auf weniger als ein Pixel am Bildschirm projiziert würden. Wer hätte je eine Ameise auf eine Entfernung von einem Kilometer erkannt? Daher wird der Sichtbereich des Spielers neben den vier Pyramidenwänden, die durch die vier Seiten des Monitors entstehen, auch durch die sogenannte Near Clipping Plane und die Far Clipping Plane begrenzt. Die Near Clipping Plane sorgt dafür, dass wir Objekte zu nah an der Projektionsfläche nicht mehr malen werden um Berechnungsfehler auszuschliessen. Die Far Clipping Plane begrenzt dann die maximale Sichtweite des Spielers die wir in unserer virtuellen 3D Welt zulassen. Dadurch erhalten wir eine liegende Pyramide mit abgeschnittenem Kopf deren sechs Flächen (*Near, Far, Rechts, Links, Oben, Unten*) den Sichtbereich des Spielers festlegen. Alles was nicht wenigstens teilweise in diesem Bereich liegt muss nicht gerendert werden weil es eh physisch nicht sichtbar ist. Und diese geköpfte Pyramide nennt sich **View Frustrum**.

Die sechs Flächen des View Frustrums könnten man nun beispielsweise als Polygone oder gar als Vierecke darstellen. Das ist uns aber zu aufwendig und daher vereinfachen wir das Problem etwas und betrachten nicht die Flächen (*Vierecke*) des Frustrums, sondern die Ebenen in denen die Flächen liegen. Unseren View Frustrum in Direct3D können wir also durch sechs Ebenen beschreiben und der folgende Code dient dazu die sechs notwendigen Ebenengleichungen zu erstellen wobei angemerkt ist, dass die Normalenvektoren der Ebenen jeweils nach aussen zeigen und nicht in den View Frustrum hinein:

```
// ViewFrustrum Seiten:
#define VF_L 0 // Linke Clipping Plane
#define VF_R 1 // Rechte Clipping Plane
#define VF_O 2 // Obere Clipping Plane
#define VF_U 3 // Untere Clipping Plane
#define VF_N 4 // Ferne Clipping Plane
#define VF_F 5 // Nahe Clipping Plane

XEBENE g_VFrustrum[6];




/**
 * Extrahieren der ViewFrustrum Ebenen aus Direct3D.
 * (By Gary Simmons, Klaus Hartmann, Gil Gribb)
 */
void xUtil_ViewFrustrum_erstellen(XEBENE *VFrustrum) {
   D3DMATRIX matView, matProjektion, matViewProj;

   g_lpD3DDevice->GetTransform(D3DTS_PROJECTION,&matProjektion);
   g_lpD3DDevice->GetTransform(D3DTS_VIEW,&matView);

   xUtil_MatrixMult(&matViewProj,&matView,&matProjektion);

   // Linke Clipping Plane
   VFrustrum[VF_L].vNormal.x = -(matViewProj._14 + matViewProj._11);
   VFrustrum[VF_L].vNormal.y = -(matViewProj._24 + matViewProj._21);
   VFrustrum[VF_L].vNormal.z = -(matViewProj._34 + matViewProj._31);
   VFrustrum[VF_L].fDistanz  = -(matViewProj._44 + matViewProj._41);

   // Rechte Clipping Plane
   VFrustrum[VF_R].vNormal.x = -(matViewProj._14 - matViewProj._11);
   VFrustrum[VF_R].vNormal.y = -(matViewProj._24 - matViewProj._21);
   VFrustrum[VF_R].vNormal.z = -(matViewProj._34 - matViewProj._31);
   VFrustrum[VF_R].fDistanz  = -(matViewProj._44 - matViewProj._41);

   // Obere Clipping Plane
```

```
   VFrustrum[VF_O].vNormal.x = -(matViewProj._14-matViewProj._12);
   VFrustrum[VF_O].vNormal.y = -(matViewProj._24-matViewProj._22);
   VFrustrum[VF_O].vNormal.z = -(matViewProj._34-matViewProj._32);
   VFrustrum[VF_O].fDistanz  = -(matViewProj._44-matViewProj._42);

   // Untere Clipping Plane
   VFrustrum[VF_U].vNormal.x = -(matViewProj._14+matViewProj._12);
   VFrustrum[VF_U].vNormal.y = -(matViewProj._24+matViewProj._22);
   VFrustrum[VF_U].vNormal.z = -(matViewProj._34+matViewProj._32);
   VFrustrum[VF_U].fDistanz  = -(matViewProj._44+matViewProj._42);

   // Nahe Clipping Plane
   VFrustrum[VF_N].vNormal.x = -(matViewProj._14+matViewProj._13);
   VFrustrum[VF_N].vNormal.y = -(matViewProj._24+matViewProj._23);
   VFrustrum[VF_N].vNormal.z = -(matViewProj._34+matViewProj._33);
   VFrustrum[VF_N].fDistanz  = -(matViewProj._44+matViewProj._43);

   // Ferne Clipping Plane
   VFrustrum[VF_F].vNormal.x = -(matViewProj._14-matViewProj._13);
   VFrustrum[VF_F].vNormal.y = -(matViewProj._24-matViewProj._23);
   VFrustrum[VF_F].vNormal.z = -(matViewProj._34-matViewProj._33);
   VFrustrum[VF_F].fDistanz  = -(matViewProj._44-matViewProj._43);

   // Normalenvektoren der Ebenen normalisieren
   for (int i=0; i<6; i++) {
      float fBetrag = 1.0f /
                      sqrtf(xUtil_Punktprodukt(&VFrustrum[i].vNormal,
                                               &VFrustrum[i].vNormal));
      VFrustrum[i].vNormal.x *= fBetrag;
      VFrustrum[i].vNormal.y *= fBetrag;
      VFrustrum[i].vNormal.z *= fBetrag;
      VFrustrum[i].fDistanz  *= fBetrag;
      } // for
   } // xUtil_ViewFrustrum_erstellen
/*----------------------------------------------------------*
```

Puh, da kann man nur sagen gut dass es Leute gibt die so etwas erschaffen. Wir können diese Funktion nun nutzen und sie wird uns die sechs Ebenen des View Frustrums mit den korrekten Daten füllen. Es ist natürlich jedem hier klar, dass wir das tatsächlich in jedem Frame machen müssen da sich der VIew Frustrum natürlich ebenso wie die Kamera bewegt und die Ebenengleichungen dann immer anders aussehen.

Bounding Box Culling

Eine Überschrift, zwei Fragen. Was sind Bounding Boxen und was ist Culling? Klären wir erst mal die Frage nach den BBs (*Bounding Boxen*) und auch hier sagt ein Bild, wie beispielsweise **Abbildung 4**, mehr als tausend Worte.

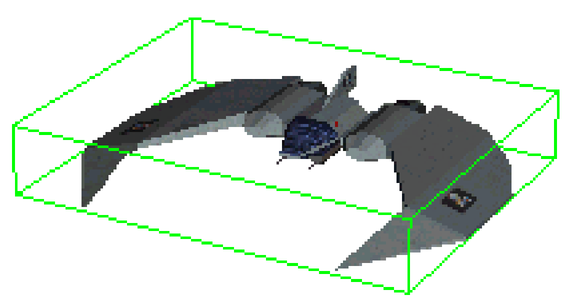


Abbildung 4: Eine Bounding Box

Eine Bounding Box ist eine Kiste die wir um alle Vertices eines 3D Objektes legen. Die Box ist also gerade so gross dass das Objekt wie massgeschneidert in sie hineinpasst. Nun gibt es aber verschiedene Arten von Bounding Boxen die unterschiedlich schwer zu handhaben sind. Ich werde mich hier auf die einfache Variante der Achsen-Ausgerichteten Bounding Boxe (AABB) konzentrieren. Das bedeutet, dass die Seiten der Box immer parallel zu dem Weltkoordinatensystem liegen. Damit nehmen wir zwar in Kauf, dass die Box eben nicht massgescheidert ist, sondern in der Regel etwas grösser ausfällt als nötig. Wenn das entsprechende 3D Modell beispielsweise nicht parallel zu den Weltachsen ist dann liegt es schief in der Box und die Box muss etwas grösser sein um das zu kompensieren.
AABB bieten uns aber den unschlagbaren Vorteil dass sie unheimlich einfach zu berechnen sind. Sehen wir uns aber erst mal die Datenstruktur an die wir für die Box benötigen werden:

```
typedef struct XBOX_TYP {
   D3DVECTOR vMin;       // Kleinster Eckpunkt der Box
   D3DVECTOR vMax;       // Grösster Eckpunkt der Box
   D3DVECTOR vMittelpkt; // Mittelpunkt (Zentrum) der Box
   } XBOX;
```

Für die Box braucht man nix anderes als die Angaben des minimalen und des maximalen Punktes. Da die Box parallel zu den Weltachsen ist handelt es sich hierbei um den Punkte der am weitesten unten, hinten und links liegt (`vMin`) bzw. am weitesten oben, vorne und rechts (`vMax`). Als kleinen Bonus berechnen wir dann noch den Mittelpunkt für die Box. Und hier sehen wir nun, wie wir eine solche Boy erstellen können, das ist nun wirklich reines Pfannkuchenessen. Wir geben der Funktion einfach eine Menge von Punkten an und sie erstellt daraus die AABB und gibt sie zurück.

```
XBOX Erstelle_BoundingBox(D3DVERTEX *Vertexliste,
                          LONG lAnz_Verts) {
   XBOX   xBox;
   XPOLY *pVertex = Vertexliste;
   float  fMax_x, fMax_y, fMax_z,
          fMin_x, fMin_y, fMin_z;

   // Startwerte beliebig wählen
   fMax_x = fMin_x = pVertex[0].x;
   fMax_y = fMin_y = pVertex[0].y;
   fMax_z = fMin_z = pVertex[0].z;

   for (LONG j=0; j<lAnz_Verts; j++) {
      // Nach neuem Maximalwert suchen
      fMax_x = MAX(fMax_x, pVertex[j].x);
      fMax_y = MAX(fMax_y, pVertex[j].y);
      fMax_z = MAX(fMax_z, pVertex[j].z);

      // Nach neuem Minimalwert suchen
      fMin_x = MIN(fMin_x, pVertex[j].x);
      fMin_y = MIN(fMin_y, pVertex[j].y);
      fMin_z = MIN(fMin_z, pVertex[j].z);
```

```
      } // for [Verts]

   // Ausdehnung der Box speichern
   xBox.vMax = xUtil_Vector(fMax_x, fMax_y, fMax_z);
   xBox.vMin = xUtil_Vector(fMin_x, fMin_y, fMin_z);

   // Mittelpunkt berechnen
   xBox.vMittelpkt = xUtil_Vector( (fMax_x+fMin_x)/2,
                                   (fMax_y+fMin_y)/2,
                                   (fMax_z+fMin_z)/2);
   return xBox;
   } // Erstelle_BoundingBox
/*----------------------------------------------------*/
```

Wenn das mal nicht einfach ist. Man beachte dass die beiden Punkte Min und Max der Box natürlich nicht mit einem Punkt aus der Liste übereinstimmen müssen, wie man auch in der **Abbildung 4** sehen kann. Daher müssen wir in der Funktion die x, y und z Koordinaten der Vertices jeweils getrennt behandeln. Immer wenn wir einen kleineren oder grösseren Wert als den bisherigen finden so speichern wir diesen. Am Ende haben wir dann jeweils den grössten und kleinsten x, y und z Wert aus der Vertexliste erwischt. Den Mittelpunkt der Box kann man dann durch einfaches Addieren und Dividieren der Min und Max Punkte erhalten.

Und nun zur Frage des Cullings. Bei der Erklärung des View Frustrums haben wir gesehen, dass wirklich nur ein sehr begrenztes Stück der gesamten virtuellen Welt am Bildschirm gesehen werden kann. Alle anderen Objekte der Welt sind nicht sichtbar und müssen daher nicht transformiert und nicht gerendert werden. Oder wie es die DirectX SDK Dokumentation sagt: "*Remember, the fastest polygons are the ones you don't draw*." Sicherlich sehen jetzt schon alle worauf das hinaus läuft. Wir müssen einfach nur alle Objekte berechnen und rendern die wenigstens teilweise im View Frustrum liegen. Alle anderen werden einfach ignoriert. Jedenfalls bei den Transformationen und dem Rendern, das Feld `vPos` usw. in der Datenstruktur muss man natürlich trotzdem setzen um beispielsweise die Bewegung des Objektes zu speichern. Okay, im Grunde genommen läuft es so, dass wir jedes Polygon nehmen und gegen die sechs Ebenen des View Frustrums testen. Kommt dabei heraus dass das Polygon wenigstens teilweise im View Frustrum liegt so wird es gerendert. Diesen Vorgang nennt man **Culling**.

Dann haben wir aber ein kleines Problem. Nehmen wir mal an dass unsere Objekte jeweils sehr viele Polygone haben. Dann landen wir dabei, dass wir in jedem Frame wenigstens Hunderte von Polygonen testen müssen was nicht gerade sehr schnell ist. Da wäre es unter Umständen schneller einfach alles blind zum Rendern zu schicken. Statt also alle Polygone eines Objektes zum Rendern zu schicken nehmen wir dafür einfach dessen Bounding Box. Denn wenn diese ausserhalb des View Frustrums ist so gilt das auch für das Objekt welches die Box umschliesst, oder?! Der Sinn der Bounding Boxen ist also einfach der, dass wir das Culling von Objekten damit schneller durchführen können.

```
// Culling Ergebnisse:
#define NWERT          int
#define BOX_INNERHALB  0
#define BOX_AUSSERHALB 1
#define BOX_GECLIPPED  2
```

Wir werden uns gleich eine Funktion schreiben die das Culling von Boxen gegen den View Frustrum ermöglicht. Hier sehen wir aber zuerst welche Ergebnisse eine solche Funktion liefern kann. Die Box liegt beispielsweise komplett innerhalb des View Frustrums, komplett ausserhalb oder teils-teils. Im ersten Fall können wir alles innerhalb der Box ohne weitere Culling Tests rendern. Im zweiten Fall können wir alles innerhalb der Box einfach komplett ignorieren. Im letzten Fall könnten wir den Inhalt der Box auch einfach ohne weitere Tests rendern, aber unter Umständen ist der Inhalt der Box so gross und unterteilt sich in weitere Objekte mit Bounding Boxen dass wir die Objekte innerhalb der Box wiederum durch den Culling Test schicken.

Und hier ist endlich die Funktion zum Cullen von Achsen-Ausgerichteten Bounding Boxen. Es sei angemerkt

dass diese Boxen natürlich nur so lange gültig sind, wie das Objekte in der Box seine Weltposition nicht verändert. Sobald sich das Objekt bewegen oder rotieren kann wird die Box ungültig. Man kann sie dann entweder neu berechnen oder mit bewegen und rotieren. Im letzten Fall wird die Box aber die Ausrichtung an den Weltachsen verlieren und man kann die folgende Funktion nicht mehr anwenden. Aber im zweiten Band meiner Bücherreihe werden wir uns das genauer ansehen und auch mit rotierten Boxen arbeiten.

```
NWERT xUtil_Cull_AABB(XBOX *pAABB) {
   D3DVECTOR bMax = pAABB->vMax;
   D3DVECTOR bMin = pAABB->vMin;
   D3DVECTOR NearPoint, FarPoint;
   BOOL      bClipped=FALSE;

   for (int i=0; i<6; i++) {
      // Initialisiere nahen Punkt unter der Annahme
      // dass alle Normalen Komponenten <= 0.0f sind
      NearPoint.x = bMax.x;
      NearPoint.y = bMax.y;
      NearPoint.z = bMax.z;

      FarPoint.x = bMin.x;
      FarPoint.y = bMin.y;
      FarPoint.z = bMin.z;

      // Überprüfe die Annahme...
      if (g_VFrustrum[i].vNormal.x > 0.0f) {
         NearPoint.x = bMin.x;
         FarPoint.x  = bMax.x;
         }

      if (g_VFrustrum[i].vNormal.y > 0.0f) {
         NearPoint.y = bMin.y;
         FarPoint.y   = bMax.y;
         }

      if (g_VFrustrum[i].vNormal.z > 0.0f) {
         NearPoint.z = bMin.z;
         FarPoint.z   = bMax.z;
         }

      // Checke ob der nahe Punkt ausserhalb liegt,
      // dann ist auch die ganze Box ausserhalb
      if (xUtil_Punktprodukt(&g_VFrustrum[i].vNormal, &NearPoint) +
          g_VFrustrum[i].fDistanz > 0)
         return BOX_AUSSERHALB;

      // Checke ob der ferne Punkt ausserhalb liegt
      if (xUtil_Punktprodukt(&g_VFrustrum[i].vNormal, &FarPoint) +
          g_VFrustrum[i].fDistanz > 0)
         bClipped = TRUE;
      } // for

   // Sind wir bis hierher gekommen dann ist die Box nicht ganz
   // ausserhalb. Lag der FarPoint aber ausserhalb dann ist die
   // Box nur teilweise im View Frustrum
   if (bClipped)
      return BOX_GECLIPPED;
```

```
   // Sonst komplett innerhalb
   return BOX_INNERHALB;
   } // xUtil_Cull_AABB
/*----------------------------------------------------------------*/
```

Die Variable `NearPoint` ist der Punkt auf der Box der dem View Frustrum am nächsten liegt. Je nach Wert der Komponenten der Normalenvektoren der View Frustrum Ebenen nehmen wir also die Werte von `vMin` oder `vMax` der Box. Danach prüfen wir ob die Box ausserhalb der Ebenen des View Frustrums liegt. Dazu benötigen wir doch einfach nur die Ebenengleichung `V*N+d=0`. Wir nehmen also das Punktprodukt zwischen dem Ebenennormalenvektor und dem NearPoint und addieren die Distanz der Ebene dazu. Ist das Ergebnis gleich 0 so liegt der Punkt genau in der Ebene. Ist das Ergebnis kleiner als 0 so heisst das, dass unser `NearPoint` näher am Ursprung liegt als die Ebene und da die Normalenvektoren des Frustrums vom Ursprung weg nach aussen zeigen liegt unser `NearPoint` (*und damit wenigstens ein Teil der Box*) damit innerhalb des Frustrums. Ist das Ergebnis grösser als 0 so gilt umgekehrt dass die Ebene dem Ursprung näher ist und der `NearPoint` (*und damit die gesamte Box*) ausserhalb des View Frustrums liegt.

Man kann übrigens auch feststellen ob die von Frustrum entfernteste Ecke, also der `FarPoint` der Box, innerhalb des Frustrums liegt. Wenn das zutrifft und der `NearPoint` ebenfalls innerhalb des Frustrums ist dann ist die gesamte Box komplett innerhalb.

Indizierte Primitive

Rufen wir uns mal ins Gedächtnis wie wir unsere grafischen Primitive bisher gerendert haben. Die Funktion `DrawPrimitiveUP()` hat diesen Job bisher erledigt. Diese Funktion hat allerdings noch einen Cousin der für diese Aufgabe meistens besser geeignet ist und das ist `DrawIndexedPrimitiveUP()`. Klären wir nun also den Unterschied zwischen diesen beiden. Im ersten Fall benötigen wir für jeden Punkt im 3D Modell einen Vertex. Wer die Übungsaufgabe am Anfang dieses Tutorials erledigt hat und ein rotierendes Rechteck erzeugt hat der weiss, dass wir hier 6 Vertices für ein Viereck benötigen, da wir zwei Dreiecke rendern.

Das klingt nicht nur so als ob das unschön wäre, sondern ist es auch. Ein Viereck hat vier Ecken, also sollte es auch mit vier Vertices auskommen. Denken wir mal an einen Würfel, dieser hat acht Eckpunkte aber um ihn wie bisher in Dreiecken zu rendern benötigen wir 6 (*Seiten*) * 6 (*Vertices für 2 Dreiecke je Seite*) = 36 Vertices in unserer Würfeldatenstruktur. Nun zu den inneren Abläufen von Direct3D. Immer wenn wir DrawPrimitiveUP() aufrufen dann nimmt Direct3D die übergebene Vertexliste und jagd alle Vertices durch die Transformationspipeline. Je weniger Vertices wir benutzen desto besser ist das. Und jetzt kommen die Indices ins Spiel.

Wir erzeugen nun beispielsweise wieder unser Viereck. Dazu erstellen wir auch wirklich nur die vier Vertices v0, v1, v2 und v3. Dann erzeugen wir zusätzlich noch eine Liste (*ein Array*) in dem wir die Indices angeben aus welchen der Vertices in der Liste sich das Dreieck zusammensetzt. Hier ein Beispiel:

```
D3DVERTEX aVertices[4] = { v0, v1, v2, v3 };

WORD aIndices[6] = { 0, 1, 2,   // erstes Dreieck
                     0, 2, 3 }; // zweites Dreieck
```

Jetzt können wir die Vertexliste und die Indicesliste mittels der Funktion `DrawIndexedPrimitive()` durch Direct3D rendern lassen. Direct3D muss jetzt nur noch 4 anstelle von ehemals 6 Vertices durch die aufwendige 3D Pipeline jagen und berechnen, das geht also schneller. Dann nimmt Direct3D die Indicesliste und rendert die beiden Dreiecke aus den Indices die angeben welche Vertices aus der Vertexliste ein Dreieck ergeben. In diesem Beispiel haben wir also zwei Dreiecke aus den Vertices v0, v1, v2 bzw. v0, v2, v3. Und nun zur Syntax der Renderfunktion:

```
HRESULT IDirect3DDevice8::DrawIndexedPrimitiveUP(
      D3DPRIMITIVETYPE PrimitiveType,
```

```
    UINT MinIndex,
    UINT NumVertices,
    UINT PrimitiveCount,
    CONST void* pIndexData,
    D3DFORMAT IndexDataFormat,
    CONST void* pVertexStreamZeroData,
    UINT VertexStreamZeroStride);
```

Der erste Parameter gibt wie gewohnt den Primitiv-Typ an, also beispielsweise `D3DPT_TRIANGLELIST`. Der zweite Index gibt an ab welcher Position in der Vertexliste wir rendern wollen und ist im Normalfall 0. Der dritte Parameter gibt dann an wie viele Vertices in der Liste stehen, gefolgt von dem vierten Parameter der sagt wie viele Primitive (*Dreiecke*) wir rendern wollen. Danach kommt ein Parameter der einen Zeiger auf den Start der Indicesliste enthält gefolgt von dem Parameter über das Datenformat der Indices. Diese können entweder `WORD`=16 (*Bezeichner* `D3DFMT_INDEX16`) oder `DWORD`=32 Bit (*Bezeichner* `D3DFMT_INDEX32`) lang sein. Als vorletzter Parameter folgt dann ein Zeiger auf den Start der Vertexliste und zu guter Letzt wieder die Grösse eines Vertex Objektes.

Nun können wir also unseren Würfel mit wesentlich weniger Vertices darstellen. Statt 36 Vertices zu definieren, und durch die 3D Pipeline berechnen zu lassen, würden wir hier mit nicht mehr als 8 Vertices auskommen und hätten damit die Rechengeschwindigkeit um ca. ein Vierfaches beschleunigt. Nun aber zu den Nachteilen dieser Methode. Wenn wir nur 8 Vertices für einen Würfel definieren so haben wir auch nur 8 Texturkoordinatenpaare und 8 Normalenvektoren für die Beleuchtung. Das resultiert darin dass man die Texturen auf dem Würfel kaum sinnvoll darstellen kann, da man die Texturen auf allen drei an einen Vertex angrenzenden Seitenflächen verändert wenn man die Texturkoordinaten modifiziert. Damit sieht die Textur auf einer Seitenfläche optimal aus, aber auf den beiden anderen Flächen wird sie höchstwahrscheinlich fehlerhaft verschoben oder verzerrt sein. Man muss also auch diese Aspekte immer bedenken.

Aber bei dem Würfel kann man auf alle Fälle jede Seitenfläche durch eine Indexliste darstellen, so dass man wenigstens je Seite statt 6 Vertices nur 4 Vertices definieren muss. Damit kommt man dann auf immerhin 6*4 = 24 statt 36 Vertices und spart immer noch viele Rechnungen ein, bleibt aber flexibel genug die Texturen korrekt anzupassen.
Ganz nebenbei haben wir jetzt auch gelernt warum man die Renderfunktionen so selten und daher mit so vielen Vertices auf einmal aufrufen sollte. Wenn wir eine riesige Vertexliste angeben aber daraus nur ein oder zwei Dreiecke rendern dann wird trotzdem die gesamte Vertexliste transformiert. Wenn wir auf unserem Würfel dann auch noch verschiedene Texturen auf allen Seitenflächen haben dann müssen wir dieselbe Vertexliste 6 mal zum Transformieren schicken was auch unschön ist. Dafür gibt es dann die sogenannten **Vertexbuffer** in Direct3D, aber das ist ein anderes Thema. Wir geben uns mit dem Speedzuwachs der Indizierten Primitive zufrieden und werden diese im nächsten Kapitel verwenden.

Vorberechnung von Lichteffekten

Immer noch alle da? Okay, wir sind jetzt sooooo nah dran unseren eigenen Quake Clone zu basteln, habt nur noch einen Absatz Geduld. Neben den Transformationen der Objekte (*die wir durch das Culling bereits reduziert haben*) ist die Beleuchtung der Objekte der nächste Speed-Killer unserer Engine. Wo immer das möglich ist sollten wir die Lichtengine von Direct3D ausschalten und vorberechnete Lichtwerte verwenden. Häh...wie geht das denn?

Um diese Frage zu klären müssen wir uns nur überlegen was genau das Licht macht und wie Direct3D das umsetzt. Nehmen wir mal an wir haben ein Polygon welches eine Wand in unserem Level darstellt. Auf dieser Wand liegt eine Textur die eine hellbraune Holzwand darstellt. Wenn viel weisses Licht auf diese Wand scheint dann in Direct3D etwas vereinfacht folgendes ablaufen.

Zuerst wird Direct3D die Farbe des Lichtes nehmen und in Werte zwischen 1.0 (*hellster Wert*) und 0.0 (*kein Licht*) für die drei RGB (*Rot, Grün, Blau*) zerlegen. Bei unserem hellen weissen Licht kommt Direct3D zu dem Schluss dass es mit einem (1.0, 1.0, 1.0) Licht zu tun hat. Dann nimmt es die vier Vertices der Wand und verpasst jedem Vertex diesen Farbwert. Nun nimmt es für die einzelnen Pixel am Bildschirm die von der Wand

belegt werden die korrespondierende Farbe aus der Textur und multipliziert dazu die interpolierten Farbwerte der Vertices der Wand. Da diese bei hellem weissen Licht 1.0 sind bleibt die Farbe der Textur genau so wie wir sie in unserem Malprogramm erstellt haben.

Gut dass ist einfach aber was will uns das sagen? Nun, probieren wir das ganze mal mit schwachen weissem Licht, welches die Farbwerte (0.2, 0.2, 0.2) hat. Sehen wir nun was passiert? Direct3D wird einfach die RGB Wert jedes Pixels der Textur mit diesen Werten multiplizieren. Das resultiert darin dass unsere hellbraune Holztextur insgesamt in dunklere Farbwerte verschoben wird und damit wie eine dunkelbraune Holztextur erscheint. So einfach ist das mit dem Licht im Grunde genommen. Dasselbe Funktioniert übrigens auch wenn man rotes, grünes oder was auch immer für Licht hat. Die Farbwerte auf der Textur werden dann entsprechend verschoben.

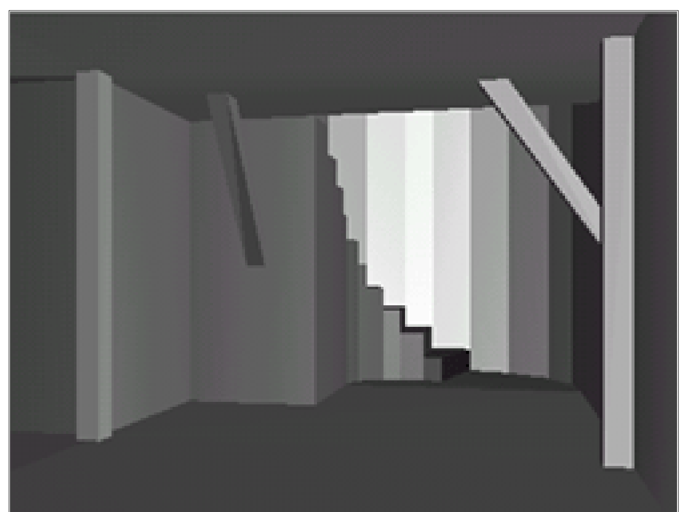
Das ist so einfach dass man es auch optimieren können muss, oder? Wenn wir schon so fragen dann ist das natürlich möglich. Wir können nämlich die folgende Datenstruktur definieren:

```
// Vertex Struktur (untransformiert, beleuchtet)
typedef struct D3DLVERTEX_TYPE {
   float x, y, z; // Position
   DWORD color;   // Diffuses Licht
   float tu, tv;  // Texturkoordinaten
   } D3DLVERTEX;

#define D3DFVF_LVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1)
```

Wenn wir diese mit der bisher von uns verwendeten Struktur D3DVERTEX vergleichen so stellen wir fest, dass der Normalenvektor in der Definition fehlt. Das liegt einfach daran, dass Direct3D diesen Vektor nur dazu gebraucht hat, die Beleuchtung zu berechnen. Hier ist er also überflüssig. Dafür haben wir aber das Feld `color` eingeführt mit dem entsprechenden Flexible Vertex Format Bezeichner `D3DFVF_DIFFUSE`. In diesem Feld speichern wir bei der Erstellung des Vertices gleich, welche Farbe das einfallende Licht an diesem Vertex haben wird. Dazu gibt es in Direct3D das folgende Makro:

```
   D3DCOLOR_XRGB(float fR, float fG, float fB)
```

Wow..aber, äh...woher wissen wir denn welche Farbe das Licht für diesen Vertex haben soll? Nun, ich gebe zu in der Regel weiss man das nicht, sondern wird es durch seine eigenen Formeln oder doch durch Direct3D berechnen lassen. Wenn wir aber über 3D Indoor Spiele sprechen dann gehen wir mal davon aus, dass wir unsere Level in einem Leveleditor erstellen. In diesem Leveleditor können wir dann auch Farben für die einzelnen Polygone festlegen, diese in der Leveldatei speichern und dann in unser Programm laden und in die entsprechenden Vertex Datenstrukturen speichern. In unserem Leveleditor werden wir also in der 3D Ansicht die Polygone so schattieren wie wir es gerne hätten. In unserem Spiel haben wir dann tatsächlich den Eindruck dass wir es mit Licht und Schatten zu tun haben. Einer abgelegenen Ecke kann man beispielsweise sehr dunkles Licht (0.1, 0.1, 0.1) zuweisen und die Ecke wird dann von Direct3D entsprechend dunkel dargestellt. Damit sparen wir uns die aufwendigen Berechnungen von Lichtquellen und Beleuchtung in Echtzeit und es sieht trotzdem alles gut aus. Eventuell sieht es sogar besser aus, da wir wirklich vorher sehen und festlegen können was welche Schattierung hat.
In der **Abbildung 5** unten sehen wir einen Screenshot des nächsten Kapitels. Die Textur der Wände ist beispielsweise immer dieselbe (*in derselben Helligkeit*), die Lichtengine von Direct3D ist deaktiviert, es finden also keine Lichtberechnungen statt. Aber die Farben der Vertices sind im Leveleditor entsprechend gesetzt so dass die Szene mehr oder weniger gut schattiert aussieht. Die linke Hälfte zeigt eine Ansicht aus dem Leveleditor wo man die verschiedenen Farbschattierungen von weiss bis schwarz erkennen kann. Die rechte Hälfte zeigt dann dieselbe Szene in der fertigen Engine mit Texturen deren Farbwerte mit den Farben der Vertices modifiziert worden. Quake und ähnliche Spiele machen das übrigens auch so. Dazu gibt es noch die Technik der Lightmaps aber das ist eine andere Geschichte.



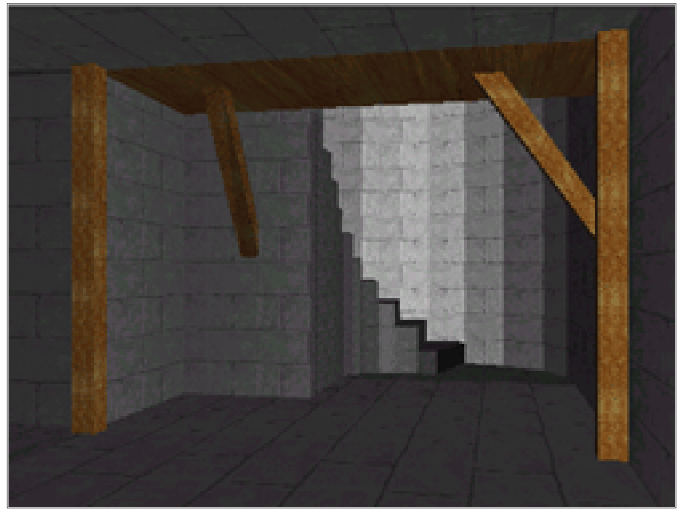


Abbildung 5: Prelit Leveldaten ohne vs mit Texturen

Nun müssen wir noch lernen wie wir die Lichtengine von Direct3D ausschalten, um unsere eigenen Farben verwenden zu können, anderenfalls wird die gesamte Szene falsch ausgeleuchtet. Im übrigen haben wir bisher auch nur ambientes Licht in Direct3D kennengelernt, welches auf jeden Vertex in der Szene mit gleicher Helligkeit strahlt, also gar keine Schattierungen produzieren würden. Es gibt in Direct3D aber noch andere Arten von Lichtquellen die bessere Effekte erzeugen aber natürlich rechenintensiver sind. In diesem Tutorial werden wir uns das wohl nicht mehr ansehen, aber das ist ebenfalls nur Pfannkuchenessen und die DirectX SDK Doku bietet entsprechende Beispiele und Erklärungen an. Das ambiente Licht ist übrigens auch der Grund, warum der Cat Jäger in unserem Code aus den letzten Kapiteln so komisch aussieht. Auch wenn man es nicht sofort auf den Punkt bringen konnte so hat das Auge dem Gehirn doch deutlich gemeltet dass da etwas faul ist. Dieses etwas ist die falsche Beleuchtung da es in der Natur einfach nicht sehr oft vorkommt dass ein Objekt keinerlei Schattierungen, sondern nur eine konstante Helligkeit auf allen Flächen hat.:

```
g_lpD3Device->SetRenderState(D3DRS_LIGHTING, FALSE);
```

So...das war glaube ich alles was wir an Vorbereitung auf das nächste Kapitel brauchen werden. Diesmal gibt es keinen Download, da wir ja nur dies und das besprochen haben ohne wirklich an einem Programm zu arbeiten. Aber das nächste Kapitel wird uns dafür entschädigen, versprochen!

Weiter geht's zum Kapitel 9...

von Stefan Zerbst

"We are doomed."
Star Wars IV - A New Hope, C3PO

BSP Bäume, 3D Indoor Level Rendering

Zu Beginn gleich die alles entscheidende Frage: Was zur Hölle ist ein BSP Baum? Die einfache Antwort: BSP Baum steht für **B**inary **S**pace **P**artitioning Baum und bedeutet, dass wir unsere simulierte 3D Welt damit in Partitionen unterteilen die in binären Bäumen gespeichert werden! Aha, dann die vielleicht wichtigere Frage: Wozu sollte uns das interessieren? Und die Antwort die alle zum Weiterlesen bringt: **Quake** (*alle Teile*) benutzt diese Technik beispielsweise um Indoor Level darzustellen!
Nachdem wir nun also genau wissen warum wir uns damit befassen wollen noch ein paar einleitende Worte. Natürlich gehört zu einer schnellen 3D Engine für das Rendering von Indoor Architektur mehr als ich hier darstellen kann. Wir haben in diesem Tutorial auch noch nicht so viele Techniken behandelt die uns bei der Erstellung einer schnellen Engine helfen müssten. Ich möchte trotzdem hier bereits ein BSP Demo in den Raum werfen da dies im Gegensatz zu allen anderen Themen im Internet schwer zu finden ist und auf Deutsch schon gar nicht. Wir beginnen unsere kleine Session hier mit einer Einführung in die Theorie die ich aber sehr anschaulich halten werde.

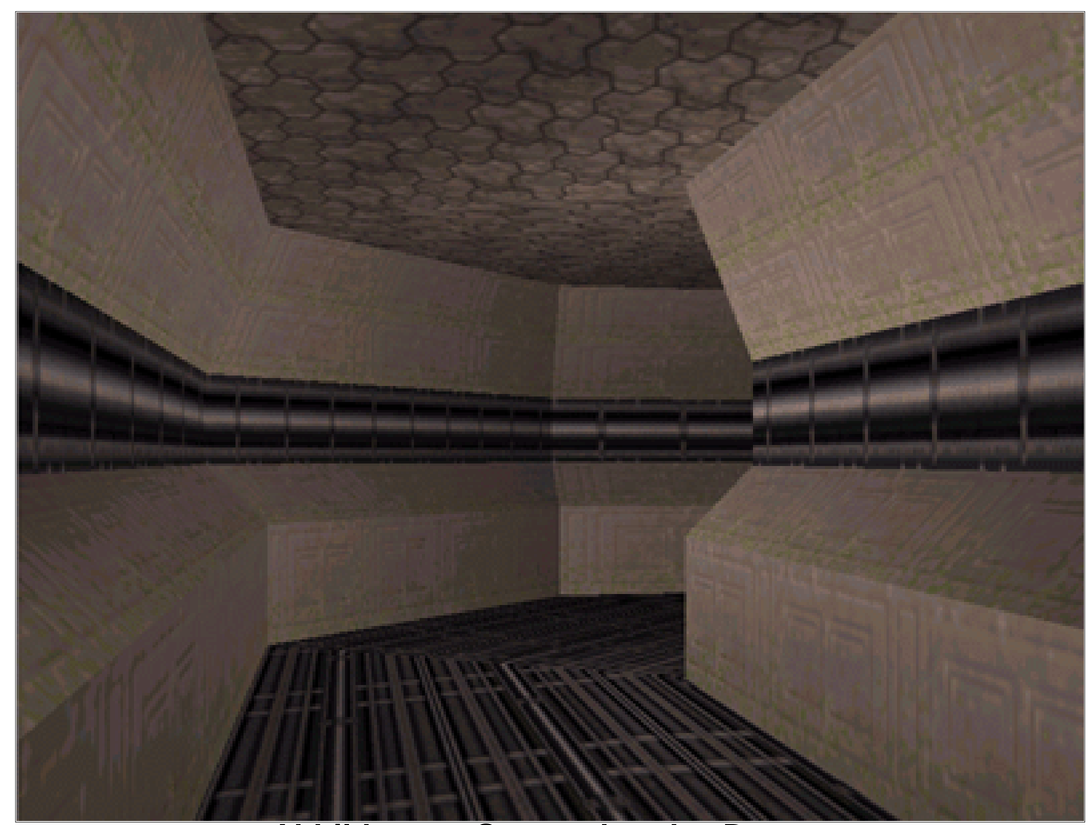


Abbildung a: Screenshot des Demos

Es ist von äusserster Wichtigkeit dass man diese Theorie mit offenen Ohren empfängt, denn das eigentliche Programmieren eines solchen BSP Biests ist gar nicht mal so kompliziert wie es eventuell klingt. Daher sollte man aber bei der Theorie trotzdem voll mitziehen, weil man sonst keine Chance hat den Code zu verstehen. Ich konnte mich allerdings nicht zurückhalten und habe den Code etwas aufgepeppt, will sagen unsere Level werden auch Texturen und Culling unterstützen. Dadurch wird der Code leider etwas umfangreicher und komplexer, als es bei einem reinen BSP Baum der Fall wäre. Ich halte es aber dennoch für interessanter eine Lösung anzubieten, die nicht nur einfarbige, langsam renderbare graue Wände zeigt, sondern schon eine gewisse Portion des Quake *Look and Feel* verstreut. Ein Bild gefällig?

Wem das bereits zu viel des guten ist, der findet unter http://nate.scuzzy.net ein OpenGL basiertes Demo eines Leaf-BSP Baumes ohne Culling oder Kollisionsabfragen von **Nate Miller**. In der Tat war mir auch der Code dieses Demo bei der Erstellung meines eigenen BSP Compilers sehr behilflich, gerade weil es sich um einen puren BSP ohne viel Schnick-Schnack handelt. Auch wenn die Verwendung von statischen Arrays den Code etwas verkompliziert **;-)** (*aber natürlich schneller macht*).

Referenzen

Informationen über BSP Bäume findet man heutzutage eigentlich wie Sand am Meer...aber es gibt solche und solche BSP Bäume, sprich Node-based und Leaf-based BSP Bäume. Erstere sind für die Spieleprogrammierung von wenig Interesse, werden aber meistens in Erklärungen im Internet verwendet. Letztere sind das, was wir wirklich wollen. Informationen über die findet man jedoch schon eher selten und schon gar nicht in guter Qualität. Ich habe aber im Laufe der Zeit neben der oben genannten noch zwei weitere Quellen entdeckt die hervorragend sind und mir alles notwendige beigebracht haben.
Das englische Tutorial von **Gerald Filimonov** hat die wunderbarste allgemeinverständliche Erklärung der Thematik die ich kenne, da kommt sogar das Tutorial von Mr. Gamemaker nicht ran (*sorry Gary*), da auch hier gilt: Es wird ohne den unnötigen Schnick-Schnack ein purer BSP beschrieben, leider ohne Demo. Aber insbesondere die grafische Einführung hat mir sehr gut gefallen und ich habe davon heftig geborgt um den BSP Baum Algorithmus ebenfalls grafisch anhand eines Beispiellevels vorzustellen.
Daneben gibt (*beziehungsweise gab*) es zwei Tutorials von **Gary Simmons** über BSP Bäume. Das erste über eben jene Node-based BSP Bäume und das zweite bereits mit Leaf-based BSP Baum inklusive PVS und allem drum und dran. Gary lieferte auch gleich läuffähigen Demo Code zu seinen Tutorials. Gary Simmons ist nun mit seinen Tutorials als BSP Instructor am Game Institute.

Wie auch immer, mein kleines BSP Demo siedelt also in der Mitte der vier genannten an und bietet ein Leaf-based BSP mit lauffähigem Demo allerdings ohne das recht aufwendige PVS System. Dieses Tutorial ist auch nur dazu gedacht, schnell in die grundlegenden Konzepte einzuführen. Aber wie bereits erwähnt gibt es in meinem Code beispielsweise auch Texturkoordinaten und Culling. Ich hoffe natürlich, dass mir die Gradwanderung zwischen einfacher, didaktischer Eleganz für die Erklärbarkeit einerseits und dem Hinzufügen von Komplexität zur wirklich sinnvollen Verwendbarkeit des Codes andererseits gelingt, aber urteilt selbst.

Ebenfalls lesenswert ist Michael Abrash's "*Inside Quake*" Artikelserie die es bei www.gamedev.net zu finden gibt. Für alle die den Mann nicht kennen: Michael Abrash ist einer der Programmierer von Quake 1, er weiss also wovon er spricht. Der Artikel liefert natürlich keinen Source Code, zeigt aber sehr gut diverse Techniken mit denen John Carmack herumgespielt hat bevor das BSP/PVS System der Quake Engine endlich dabei herauskam und man lernt auch, dass die Quake Basis-Engine nicht an einem Tag geschaffen wurde...das hat John Carmack ein ganzes Wochenende gekostet **:-)**

Warum ein BSP Baum?

Widmen wir uns der einleitenden Frage noch einmal etwas ausführlicher. Wir haben bereits gelernt, wie wir 3D Modelle aus X File Dateien laden können welche wir ganz einfach mit einem 3D Modellierungsprogramm wie AC3D erstellen können. Okay, warum nehmen wir dann nicht einfach unser tolles Modellierungsprogramm und erzeugen ein schönes Indoor Level von, sagen wir mal, unserer Wohnung (*um klein anzufangen*). Dann laden wir dieses Modell in unsere 3D Engine und schwups...schon haben wir unseren eigenen Quake Clone. Na gut, probiert es aus ich warte hier so lange...
...oh schön zurück?...Ah...ihr wartet immer noch darauf dass der erste Frame fertig gerendert wird **hehe**

In der Tat ist es wenig ratsam ein 3D Modell für Leveldaten zu verwenden. Stellen wir uns mal ein Quake Level vor welches im Durchschnitt so um die 10'000 Polygone haben wird die jeweils noch in Dreiecke zerlegt und als solche gerendert werden müssen. Gehen wir mal davon aus, dass jedes Polygon im Best-Case Szenario ein Viereck ist und damit nur aus zwei Dreiecken besteht. Selbst dann haben wir schon 20'000 Dreiecke die wir je Frame zum Rendern schicken. Selbst moderne 3D Beschleuniger machen da nicht mit, wenn wir noch Texturen auf den Polygonen haben und noch andere Berechnungen während eines Frames durchführen

müssen.
Wo ist das Problem? Der Bottleneck einer 3D Engine ist *immer* das Rendern der Grafik. Die heutigen Prozessoren sind schnell genug um die eine oder andere nachlässig aufwendige Berechnung zu tolerieren. Aber immer wenn wir ein Dreieck an das Device zum Rendern schicken dann geht unsere Anwendung in die Knie. Die Vertices des Dreiecks werden zuerst mit der Weltmatrix in Weltkoordinaten transformiert, dann mit der View Matrix in Kamerakoordinaten und dann mit der Projektionsmatrix von 3D auf 2D projiziert. Zusätzlich muss noch die Beleuchtung für die Vertices berechnet werden, was auch viel Zeit kostet je nach Art der Lichtquelle. Wenn das alles passiert ist dann wird das Device das Dreieck rasterisieren (*an die Pixel des Bildschirms anpassen*) und dann versuchen es zu rendern, Pixel für Pixel. Jeder Pixel der wirklich im sichtbaren Bereich des Bildschirms liegt wird dann mit dem Wert im Z Buffer verglichen, auch das kostet viel Zeit. Dann erst wird der Pixel gerendert, oder auch nicht wenn der Z Buffer dagegen ist. Liegt der Pixel aber ausserhalb des Bildschirms, so wird er einfach verworfen.
Nun haben wir zwei dicke Probleme. Das Device kontrolliert den Pixel auf Sichtbarkeit am Bildschirm erst nach allen Transformationen, Projektionen und Beleuchtungen für das Dreieck. Selbst wenn unser Dreieck zweihundert Kilometer hinter dem Spieler liegt wird es voll berechnet. Das zweite Problem ist, selbst wenn das Dreieck im Bereich des Bildschirms liegt, aber von anderen Dreiecken verdeckt wird, so wird das Device das Dreieck erst voll berechnen um es dann vom Z Buffer Test verwerfen zu lassen.

Lange rede kurzer Sinn. Stellen wir uns einen Wolkenkratzer in New York vor, den wir als 3D Indoor Level mit Liebe zum Detail als Leveldatei vorliegen haben. Jede einzelne Etage enthält Dutzende von Büros, Aufenthaltsräumen, Korridoren usw. Wenn wir den Level einfach als 3D Modell blind rendern, dann schicken wir selbstverständlich alle Dreiecke des Modells an das Device. Man sieht schnell ein dass die dann durchgeführten Berechnungen für 10'000sende von Dreiecken den Computer so zum Ächzen bringen, dass das Spiel unspielbar wird. Ausserdem zeugt dieser Ansatz davon, dass wir lieber einmal fett Zuschlagen anstatt einmal gescheit nachzudenken. Wir müssen also eine Möglichkeit finden, unsere 3D Level so in unsere Engine zu laden dass wir mit wenig Mühe herausfinden können welche von den 10'000 Polygonen am Bildschirm sichtbar sein könnten und welche eh neben oder gar hinter dem Spieler liegen und damit auf alle Fälle unsichtbar sind. Dann schicken wir einfach nur diejenigen Polygone unseres Levels zum Rendern, die vor dem Spieler liegen und damit potentiell sichtbar sind (*falls sie nicht von anderen Wänden verdeckt werden*). Der Z Buffer sorgt dann dafür, dass wir diese Polygone korrekt rendern können.
Nehmen wir einfach mal an, dass sich der Spieler genau in der Mitte des Wolkenkratzers befindet, also auch in der horizontalen Mitte auf einer Etage. Es ist leicht einzusehen, dass dann mindestens die Hälfte des Wolkenkratzers bereits hinter dem Rücken des Spielers liegt. Ohne die entsprechenden Dreiecke durch aufwendige Berechnungen zu schicken können wir sie einfach für diesen Frame ignorieren. Stellen wir uns weiter vor, dass der Spieler sich in einem Büroraum befindet. Wie viele Wände mag der haben? Vier, vielleicht auch fünf oder sechs. Selbst ein grosszügig gestalteter Raum wird kaum mehr als 30 Dreiecke haben und das ist alles was wir in diesem Frame *wirklich* rendern müssen. Das Problem ist aber, selbst wenn wir das Rendern nur auf diese eine Etage beschränken, so müssen wir wenigstens alle Räume an das Device schicken die *vor* dem Spieler liegen und damit potentiell sichtbar sind. Wir wissen ja nicht, ob man durch die Tür des Büros die dahinter liegenden Räume sehen kann, oder einen langen Flur mit vielen offenen Türen und den entsprechenden Räumen von denen man jeweils ein Stück sehen könnte.

Und hier kommt unser BSP Baum ins Spiel. Sinn des Baumes ist es nämlich, die Daten eines Levels so zu strukturieren, dass wir wesentlich schneller entscheiden können welche Teile des Levels im potentiellen Sichtbereich des Spielers liegen, also vor der Kamera. Alle anderen Bereiche (Dreiecke) des Levels können wir ohne kostspielige Berechnungen schnell für einen Frame verwerfen. Damit haben wir das erste Problem gelöst, nämlich wie wir die Teile über, unter, neben und hinter dem Spieler aussortieren. Bleibt immer noch das zweite Problem, mit den Bereichen die zwar vor dem Spieler liegen, aber von anderen Wänden verdeckt werden und daher unsichtbar sind.
Zwar verhindert der Z Buffer, dass wir grafische Fehlern beim Rendern haben. An der Tatsache dass wir einen sogenannten **Overdraw** haben ändert sich aber nix. Als Overdraw bezeichnet man die Tatsache, dass jeder Pixel des Bildschirms Vielfach angesprochen wird. Entweder übermalen wir weiter entfernte Wände ständig mit denen die näher am Spieler liegen oder der Z Buffer muss bemüht werden um zu verhindern dass wir eine bereits gemalte, nahe liegende Wand mit einer weiter entfernten übermalen, was grafische Fehler produzieren würde.

Das zweite Problem adressieren kommerzielle Anwendungen wie beispielsweise Quake durch die Verwendung eines PVS Systems, welches ich am Ende dieses Kapitels noch einmal allgemein erklären werde. Dadurch wird dann übrigens auch das erstgenannte Problem mit gelöst. Allerdings kann man das zweite Problem auch ohne PVS durch einen reinen BSP Baum lösen, wenn man bei seiner Geometrie ein paar Zugeständnisse macht. Etwas detaillierter bedeutet dass für uns, dass wir keine allzugrossen Level verwenden sollten, aber was noch wichtiger ist: Wir müssen die Sichtweite des Spielers in einem massvollen Rahmen halten. Wenn der Spieler eben nicht durch einen langen Flur viele andere Räume sehen kann, dann können wir ein wenig tricksen. So lange der Spieler nur Geometrie sehen kann, die ein paar Dutzend Meter entfernt ist und nicht ein paar Hundert Meter sollten wir auch bei grösseren Level eine gute Framerate erreichen. Und genau das werden wir jetzt endlich versuchen, nachdem wir nun eingesehen haben, dass wir mit dem Hau-Drauf Ansatz eigentlich wenig erreichen, so wie im realen Leben auch.

Fassen wir das ganze Blabla noch einmal zusammen. Mit einem BSP Baum können wir also

(a) ...die einzelnen Polygone so gruppieren dass man sie auf Sichtbarkeit hin prüfen kann und ganze Gruppen von Polygonen mit wenigen Tests vor dem Rendern aussortieren kann, und dass man...

(b) ...die einzelnen Polygone so ordnen kann dass sie in korrekter Reihenfolge gerendert werden und sich nicht gegenseitig überdecken. Das ist heutzutage aufgrund des schnellen Z Buffers nicht mehr so wichtig, aber... ...was sehr wohl wichtig ist, das ist das Rendern der Polygone in korrekter Reihenfolge von hinten nach vorne aus der Sicht der Kamera wenn man mit **Transparenzeffekten** arbeitet! Das haben wir hier zwar noch nicht besprochen und werden das auch nicht tun, aber dennoch ein Satz: Transparenz wie beispielsweise Glasscheiben oder andere lichtdurchlässige Objekte im Level können nur korrekt dargestellt werden wenn *zuvor alle dahinter* liegenden Objekte gerendert wurden.

Ganz nebenbei fallen bei einem BSP Baum noch andere Goodies für uns ab, mit denen wir uns aber später beschäftigen werden. Jetzt sollten wir uns erst mal in die anschauliche Theorieeinführung stürzen die ich versprochen habe. Oh...eines habe ich noch vergessen. Wenn wir jetzt von Leveldaten sprechen dann bezieht sich das wirklich nur auf die Objekte die tatsächlich zu der *Architektur* des Levels gehören, und nicht zur *Einrichtung*. Unser Wolkenkratzer besteht also wirklich nur aus Wänden, Decken und Böden. Hochdetallierte statische Objekte wie beispielsweise Tische, Lampen, Stühle, Schränke und so weiter werden dem Level erst *nach* dem Erstellen des BSP Baumes zugefügt. Sie würden den Baum an sich unnötig kompliziert und zu gross machen. Dasselbe gilt natürlich insbesondere für nicht-statische (*also animierte, bewegliche*) Objekte wie beispielsweise Gegner, Monster usw. Ein BSP Baum eigent sich auch nur für wirklich statische Geometrie!!! Special Effects wie beispielsweise eine einstürzende Wand kann ein BSP nicht verarbeiten, dazu muss man dann spezielle Tricks anwenden.

Theorie und Algorithmus der BSP Bäume

Das Ziel eines BSP Baumes ist es die virtuelle 3D Welt in **konvexe** Häppchen aufzuteilen. Damit stellt sich die Frage was konvex ist und warum das das Ziel der BSP Bäume ist. Betrachten wir die rechts stehende **Abbildung 1** um den Unterschied zwischen konvex und nicht-konvex (*konkav oder komplex*) zu erkennen:
Als konvex bezeichnet man Formen bei denen es keine *Dellen* gibt (*jetzt würde mich ein Mathe Prof zum ersten Mal im Verlaufe dieses Tutorials schlagen*). Stellen wir uns die beiden rechts stehenden Formen als Grundriss eines Zimmers vor. Bei dem konvexen linken Zimmer spielt es keine Rolle wo in dem Zimmer wir uns befinden. Wenn wir an einer beliebigen Stelle des Zimmers stehen und uns drehen, dann können wir jede Wand des Zimmers sehen. Keine Wand wird durch eine andere verdeckt. (*Aha, der aufmerksame Leser wittert hier bereits die Lösung für das gegenseitige Verdecken von Dreiecken in unserem Level was wir in der Einleitung besprochen haben*)

konvex nicht-konvex

Abbildung 1: Konvexität

Anders bei dem nicht-konvexen Objekt. Hier ist es eben *nicht* egal, wo in dem Raum wir stehen. Es gibt Stellen im Raum an denen wird eine Wand wenigstens teilweise durch eine andere verdeckt. In einer solchen Situation müssten wir die Polygone unseres 3D Levels umständlich sortieren um zu wissen welche Wand welche anderen Wände verdeckt. Wir können ohne Z Buffer nicht einfach die Wände des Levels blind durcheinander rendern und dann erwarten dass wir per Zufall die richtige Reihenfolge erwischt haben. Es ist sehr wahrscheinlich dass wir dann erst eine Wand malen die eine andere eigentlich verdecken sollte und danach jene besagte Wand die dann fälschlicherweise über die zuerst gerenderte Wand gemalt wird obwohl es umgekehrt sein sollte.

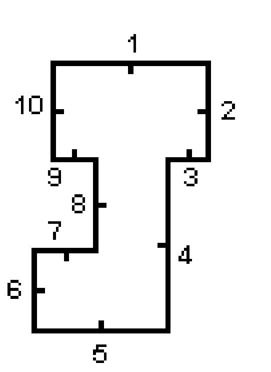


Abbildung 2: Ein Testlevel

Heutzutage verwendet man natürlich einen Z Buffer um genau solche Fehler zu verhindern. Ein aktivierter Z Buffer kostet aber Zeit bei dem Vergleich der einzelnen Pixel beim Rendern und ein 3D Programm ist schneller ohne einen Z Buffer. Daher wäre es natürlich schön, wenn wir unsere Polygone in entsprechend konvexen Gruppen vorliegen hätten die bereits sortiert sind. Zudem bietet diese Gruppierung noch andere Vorteile wie eine einfachere Kollisionsabfrage. Damit ist es also erklärtes Ziel unsere BSP Baumes eine Liste von Polygonen (*also unser Level*) in konvexe Häppchen aufzuteilen. Sehen wir uns an einem Beispiel an wie das funktioniert.
Die links stehende **Abbildung 2** zeigt uns ein kleines Beispiellevel welches wir in einen BSP Baum zerlegen wollen. Wir sehen, dass bereits ein so kleines Level nicht mehr konvex ist. Das Level ist entsprechend einfach gehalten und besteht nur aus 10 Polygonen die beliebig durchnummeriert sind. Da wir ja nun aus den vorhergehenden Kapiteln wissen, dass 3D Flächen immer nur eine Vorderseite haben müssen wir auch wissen welche das ist. Der kleine Strich an jeder Wand zeigt welche Seite des Polygons die sichtbare Vorderseite ist.

Okay, mögen die Spiele beginnen. Unser Job, oder vielmehr der des BSP Baumes, ist es nun diese Menge von Polygonen in konvexe Häppchen zu unterteilen. Unser Algorithmus zur Erstellung des BSP Baumes beginnt also damit, dass wir einem sogenannten **BSP Compiler** (*einem Programm welches den BSP Baum erstellt*) die Menge von Polygonen einfüttern aus denen unser Level besteht. Aus dieser vollkommen ungeordneten Suppe von Polygonen muss der BSP Compiler (*also wir*) jetzt eine sinnvolle Struktur erzeugen. Dazu werden wir jetzt eines der Polygone auswählen und unsere Menge an Polygonen durch die Ebene des ausgewählten Polygons unterteilen...Moment erst ein Bild:

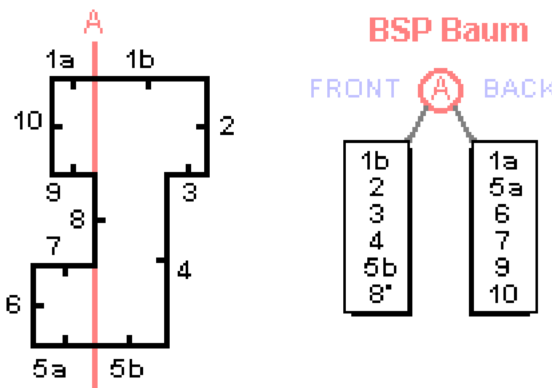


Abbildung 3: Unterteilung des Levels

Die **Abbildung 3** zeigt, dass wir das Polygon Nummer 8 verwendet haben, um die Menge der Polygone zu teilen. Wir werden später noch lernen wie wir ein Polygon geschickt für diesen Job auswählen. Jetzt begnügen wir uns erst mal mit der Erklärung, dass dieses Polygon die Menge der Polygone möglichst in zwei gleich grosse Hälften spalten sollte.
Auf der rechten Seite der Abbildung sehen wir bereits den BSP Baum in der Entstehung. Diese Struktur nennt man übrigens Baum, weil sie grafisch aufgemalt wie ein umgedrehter Baum aussieht der sich in immer mehr Äste verzweigt. Binary heisst der BSP Baum weil jedes Baumelement immer genau zwei (=*binär*) neue Äste hat. Dazu sehen wir auch, warum der BSP Baum Space Partitioning heisst, denn er dazu dient Raum (=*space*)

aufzuteilen (=*partitionieren*). Aber zurück zur Abbildung.
Wir nehmen einfach die Ebene in der das Teilungspolygon (*hier Nummer 8*) liegt. Diese Ebene ist in der Abbildung rot markiert und als A bezeichnet. Eine Ebene ist übrigens so etwas wie eine Fläche ohne Ränder die sich unendlich weit erstreckt. Stellen wir und das Ding also wie eine Art riesige Glasscheibe vor die kein Ende hat (*hier holt der Mathe Prof zum zweiten Schlag aus*). Nun prüfen wir alle Polygone der Polygonmenge auf welcher Seite der Teilungsebene sie liegen. Die Vorderseite (*Front*) der Ebene zeigt natürlich in die gleiche Richtung wie das Teilungspolygon (*hier immer noch Nummer 8*). Wir starten also unseren BSP Baum mit der Ebene A als sogenanntem Wurzelelement und geben diesem Baumelement A zwei Äste. Ein Ast wird als Front und der andere Ast wird als Back (*Rückseite*) bezeichnet. In diese beiden Äste füllen wir nun entsprechend die Polygone die vor beziehungsweise hinter der Teilungsebene liegen.
Dabei gibt es (*natürlich!*) ein paar Sonderfälle. Was wäre die Welt ohne sie? Was sehr häufig vorkommen wird ist, dass einige Polygone auf beiden Seiten der Ebene liegen, will sagen sie werden durch die Ebene in zwei Teile geschnitten. Das passiert hier im Fall der Polygone Nummer 1 und Nummer 5. In so einem Fall müssen wir die Polygone teilen (*splitten*). Aus Polygon 1 und 5 werden jeweils zwei vollkommen neue Polygone, das ursprüngliche Polygon hört auf zu existieren. Die neuen Polygone erhalten jeweils die kleinen Buchstaben a und b um sie kenntlich zu machen. Der zweite Spezialfall ist der, dass ein Polygon genau in der Teilungsebene liegt. Das ist mindestens für das eine Polygon der Fall welches das Teilungspolygon war (*jaja...immer noch Nummer 8*). Für alle Polygone die in der Ebene liegen kontrollieren wir, in welche Richtung sie zeigen. Alle Polygone die in dieselbe Richtung wie die Ebene (*also wie das Teilungspolygon*) zeigen kommen in die Frontliste des BSP Baumes, so auch das Teilungspolygon selbst. Alle Polygone die in die andere Richtung kucken kommen in die Backliste des Baumes.

Easy oder? Also machen wir weiter. Nun nehmen wir jeweils eine der beiden Listen und wiederholen das Verfahren einfach. Dabei wird die Menge der Polygone nun auf diejenigen Polygone beschränkt die sich tatsächlich in der Liste befinden, und <u>*keine*</u> anderen. Ein weiteres Kriterium welches wir beachten müssen ist natürlich, dass wir jedes Polygon nur <u>*einmal*</u> als Teilungspolygon verwenden dürfen, daher hat Polygon Nummer 8 in der Liste eine entsprechende Markierung erhalten. Es hätte doch wenig Sinn, wenn wir jetzt noch einmal das Polygon Nummer 8 verwenden würden, da alle Polygone in der Frontliste sowieso schon auf einer Seite der Teilungsebene liegen. Die **Abbildung 4** zeigt wie der BSP Baum das Level im nächsten Schritt unterteilt wenn wir zuerst die Backliste bearbeiten. Damit besteht die Menge aller Polygone in diesem Schritt also <u>*nur*</u> aus den Polygonen die hinter der Teilungsebene A (*also in der entsprechenden Backliste*) liegen.

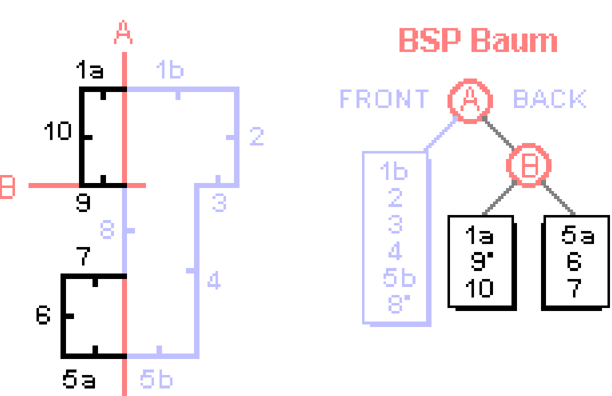


Abbildung 4: Weitere Teilung des Levels

Hier wählen wir das Polygon Nummer 9 als Teilungspolygon und erstellen die Teilungsebene B. Im BSP Baum kommen dann wie gehabt alle Polygone die vor der Ebene liegen (*oder auf ihr und in dieselbe Richtung blicken*) in die Frontliste im linken Ast und die anderen Polygone in den rechten Ast zur Backliste. Sehen wir uns diese beiden Listen nun gut an. Die beiden entsprechenden Teilmengen aus den Polygonen [1a,9,10] beziehungsweise [5a,6,7] bilden nun konvexe Formen.
Weder die Frontliste noch die Backliste ist weiter unterteilbar. Egal welches der Polygone wir jeweils aus den Mengen auswählen und als Teilungspolygon verwenden, wir können die Menge einfach nicht weiter unterteilen. Die Backliste würde in jedem Fall leer bleiben. Damit haben wir auch schon unser Kriterium gefunden, ab wann wir es mit einem konvexen Raum zu tun haben und die weitere Unterteilung stoppen können.

Damit gehen wir zurück zu dem letzten unvollständig bearbeiteten Ast in unserem BSP Baum. Das ist hier die

Frontliste der Teilungsebene A. Hier setzen wir also an und machen mit unserem Verfahren weiter. Dabei halten wir immer im Hinterkopf dass das Polygon Nummer 8 in dieser Liste bereits als Teilungspolygon verwendet wurde und die nicht noch einmal passieren darf. Daher ist die 8 auch mit einem zusätzlich Sternchen in der Liste versehen worden, wir wissen ja wie vergesslich wir manchmal sind **;-)**

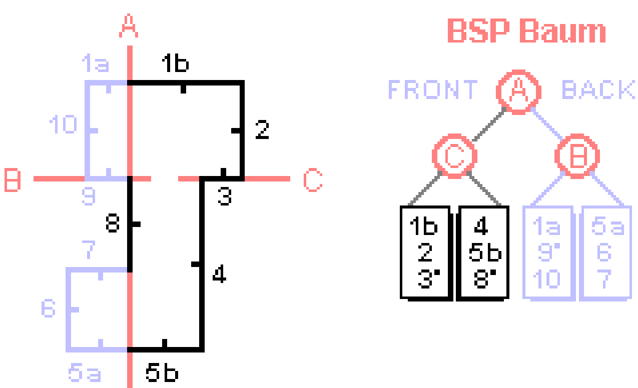


Abbildung 5: Weitere Teilung des Levels

Na, sehen wir alle ein was hier passiert ist? Wir haben das Polygon Nummer 3 als Teilungspolygon gewählt und dann einfach die anderen Polygone als Front oder Back klassifiziert. Aber an dieser Stelle gibt es noch zwei interessante Anmerkungen zu machen. Zum einen ist die neue Teilungsebene C identisch mit der alten Ebene B und wie wir sehen macht das dem Algorithmus gar nichts aus. Zum anderen hätten wir hier auch ein anderes Teilungspolygon wählen können, beispielsweise das Polygon Nummer 4. Dann hätten wir aber das Polygon 1b noch einmal splitten müssen und zwar in 1bx und 1by. Der Algorithmus hätte auch dann keine Probleme bekomme aber das wäre trotzdem unschön geworden. Dazu gleich mehr.
Hier sehen wir aber, dass unser BSP Baum schon fertig ist. Die neue Front- und Backliste ist ebenfalls jeweils von konvexer Form und kann nicht weiter geteilt werden. Wir haben jetzt einen BSP Baum mit vier sogenannten Leafs (*Blättern*) in denen sich jeweils drei Polygone einer konvexen Form befinden. Wir werden später einsehen dass uns die Struktur des BSP Baumes einige Kniffe erlaubt mit deren Hilfe wir das Rendern enorm beschleunigen können.

Höhere Geschwindigkeit durch Bounding Boxen...

Für das obige Beispiel mag das noch nicht so sichtbar sein. Stellen wir uns aber vor, dass die Elemente A, B und C ebenso wie die vier Leafs mit den eigentlichen Polygonen jeweils eine sogenannte Bounding Box gespeichert haben. Das ist einfach eine grosse Kiste die alle Polygone einschliesst die in allen unteren Ästen unter einem Element stehen. Die Bounding Box des Elementes A umfasst also alle Polygone der gesamten Welt (*Nummer 1a, 1b, 2, 3, 4, 5a, 5b, 6, 7, 8, 9 und 10*), die Box des Elementes B ist schon etwas kleiner und schliesst nur noch die Backlist Elemente von A ein (*Nummer 1a, 5a, 6, 7, 9 und 10*) und die Box des Backleafs von C ist dann beispielsweise nur noch so gross und so in der Welt positioniert, dass sie nur noch die Polygone der Backliste von C einschliesst (Nummer 4, 5b und 8).
Der Sinn der Sache ist folgender. Während des Spiels kennen wir natürlich die Position des Spielers in der 3D Welt. Wir kontrollieren also ausgehend von der Wurzel A des Baumes ob die entsprechende Bounding Box im Sichtbereich des Spielers liegt. Ist dies der Fall so setzen wir den Test jeweils für die beiden Kinderäste fort und immer so weiter. Finden wir irgendwann eine Box die nicht im Sichtbereich des Spielers liegt, so können wir sofort sagen, dass ebenfalls keines der Elemente (*ob Bounding Box, Node oder Leaf mit den eigentlichen Polygonen*) in dem unterliegenden Ast im Sichtbereich des Spielers liegend wird...ohne dass wir jedes Element explizit prüfen müssten. Stellen wir in obigem Beispiel also fest, dass die Bounding Box von Element B ausserhalb des Sichtbereichs liegt so haben wir mit einem einzigen Test bereits die Hälfte aller Polygone in unserer Welt aussortiert. *Wow, Klasse!*
Und das gilt nicht nur für für unser Minibeispiel. Stellen wir uns den BSP Baum eines Levels in Quake Grösse vor. Wenn wir bereits im ersten oder zweiten Schritt unter der Wurzel eine Bounding Box finden die nicht im Sichtbereich liegt, so haben wir nach nur ein, zwei kurzen Tests bereits ca. 5'000 von 10'000 Polygonen *eliminiert*. In dem anderen Ast werden wir auch noch viele Elemente samt ihrer Kinderäste aussortieren können

so dass wir wieder in die *schwarzen* Polygon Zahlen kommen, also eine Anzahl an Dreiecken die in Echtzeit am Bildschirm renderbar sind ohne dass der Prozessor schlapp macht. Das Entfernen von nicht sichtbaren Polygonen aus der Rendering Pipeline nennt man übrigens Culling und es ist die *wichtigste* Technik, um grosse Polygonmengen in Echtzeit behandelbar zu machen.
Natürlich unter der Voraussetzung dass unser View Frustrum nicht allzu gross ist. Je kleiner der View Frustrum desto mehr Bounding Boxen liegen ausserhalb...logisch. Den Frustrum können wir aber nur durch eine Variable kleiner machen, nämlich durch die Far Clipping Plane. Alle anderen 5 Flächen des Frustrums sind durch den FOV festgelegt worden. Der Nachteil dabei ist natürlich, dass wir in einem solchen System keine grossen, offenen Gebiete zulassen können. Wenn wir einen sehr langen Korridor haben, so wird dieser dann eventuell durch die Far Clipping Plane einfach nach der Hälfte abgeschnitten und nicht mehr gerendert. Wir müssen also unsere Level so konstruieren, dass der Spieler von keinem Punkt und in keiner Ausrichtung im Level weiter sehen können sollte, als wir es durch die Far Clipping Plane bestimmt haben. Das ist zwar bei weitem nicht so gut wie ein PVS (*siehe unten*) aber es versetzt uns dazu in die Lage, sehr komplexe und insgesamt grosse Level trotzdem halbwegs schnell zu rendern.

...und sinnvolle Auswahl der Teilungsebene

Zurück zu den Teilungsebenen. Unser Ziel sollte es nicht nur sein, den BSP Baum möglichst gleichmässig zu halten, also mit möglichst gleich vielen Polygonen in jedem Ast des Baumes (*das haben wir in obigem Beispiel unrealistischerweise zufällig genau getroffen*).Ein zweites wichtiges Kriterium bei der Auswahl des besten Teilungspolygons ist nämlich, wie viele Polygone durch ein Teilungspolygon gesplittet werden müssen. Das splitten heisst hier wirklich splitten, denn wir werden die Polygone wirklich physisch teilen und aus einem Polygon zwei erzeugen. Wenn wir ein Level aus 10'000 Polygonen haben und jedes Polygon davon splitten müssten so enthält unser BSP Baum später wirklich 20'000 Polygone. Wenn wir also nachher das beste Teilungspolygon aus einer Menge von Polygonen auswählen so müssen wir beachten, dass die entsprechende Teilungsebene die Menge der Polygone (a) möglichst gleichmässig in zwei Teile teilt und (b) dabei möglichst wenige Splits durchführt. Diese beiden Ziele können durchaus gegensätzlich sein, wir müssen dann einen geeigneten Kompromiss finden.

Kurze Pause!

Okay, wie geht es jetzt weiter? Im folgenden werde ich kurz und knapp den Pseudocode angeben, der zur Erstellung eines BSP Baumes nötig ist. Dann hat man später die nötige Übersicht, um den BSP Code in den Life-Fire Quelltexten schnell zu identifizieren, da die Einbindung von Bounding Boxen, Texturkoordinaten und allen Datenstrukturen dann noch etwas Overhead zufügt der dem Verständnis nicht gerade auf die Sprünge hilft.

Im Anschluss an die Darstellung des Algorithmus im Pseudocode folgt dann die reale Implementierung für das lauffähige Demo. Dort werden wir alles entwickeln was wir brauchen um unsere Indoorlevel auf den Schirm zu bringen. Im letzten Teil dieses Kapitels werde ich dann darauf eingehen was unserem Level noch fehlt und wie wir das alles einbinden können. Für diesen Teil gibt es aber nur noch Denkanreize und keinen Code mehr, der BSP Baum ist schon lang genug und selber coden macht ja bekanntlich auch schlau. Aber keine Panik, Ihr werdet dann genug wissen um selber weiter zu machen.

Erstellung des BSP Baumes in Pseudocode

Wie immer beginnt unsere Arbeit damit, eine Datenstruktur zu entwickeln. Sowohl für die Wurzel des Baumes als auch wie die Nodes und die Leafs verwenden wir allerdings dieselbe Struktur. Dabei ist aber zu beachten, dass die Leafs andere Daten speichern müssen. Wenn der BSP Baum erst mal fertig ist, dann werden die Nodes (*auch die Wurzel ist ein normaler Node*) jeweils ihre Teilungsebene speichern, eine Bounding Box um alle ihre Kinderäste und zwei Zeiger auf die beiden Kinderäste. Bei den Leafs sind diese beiden Zeiger dann entsprechend leer. Dafür speichern die Leafs jeweils eine Liste ihrer Polygone ebenso wie die Anzahl der Polygone in der Liste:

```
typedef struct XNODE_TYP {
   XEBENE     xEbene;     // NODE: Teilungsebene
   UCHAR      blnLeaf;    // NODE oder LEAF
   LONG       lAnz_Polys; // Anzahl der Polygone
   XPOLY     *pPolys;     // LEAF: Liste der Polygone
   XNODE_TYP *pFront;     // NODE: Frontliste
   XNODE_TYP *pBack;      // NODE: Backliste
   XBOX       xBox;       // Bounding Box
   } XNODE;
```

Um die entsprechenden Datentypen wie `XEBENE` oder `XPOLY` machen wir uns hier erst mal keine Gedanken. Das sind wiederum Datenstrukturen für die entsprechenden Elemente, die wir aber hier als existent und gegeben voraussetzen werden. Okay, dann können wir die Konstruktion unseres Baumes beginnen:

```
XNODE BSP_Baum;

BSP_Baum.pPolys     = Lies_Polygonliste_aus_Leveldatei();
BSP_Baum.lAnz_Polys = Lies_Anzahl_Polygone_aus_Leveldatei();

Erstelle_BSP_Baum(&BSP_Baum);
```

Wir benötigen also nur die Anzahl der Polygone des Levels und die Polygone in Form einer Liste. Beiden Informationen sind in der Leveldatei gespeichert und wir lesen sie einfach aus. Der simple Funktionsaufruf von `Erstelle_BSP_Baum()` erstellt dann den gesamten Baum. Betrachten wir uns also diese Funktion, in der korrekten Annahme dass sich dahinter etwas mehr verbirgt:

```
void Erstelle_BSP_Baum(XNODE *pNode) {
   // Suche beste Teilungsebene aus der Polygonliste
   pNode->xEbene = Finde_besten_Splitter(pNode->pPolys,
                                         pNode->lAnz_Polys);
   // Keine Teilungsebene gefunden => Muss ein konvexes Leaf sein
   if (!pNode->xEbene) {
      pNode->blnLeaf = TRUE;
      // Erzeuge die Bounding Box für die Polygone
      pNode->xBox = Erstelle_BoundigBox(pNode->pPolys, pNode->lAnz_Polys);
      pNode->pFront = NULL;
      pNode->pBack  = NULL;
      return;
      }

   XNODE xFrontnode, xBacknode;
   int   nKlasse;
   // Durchlaufe alle Polygone und sortiere sie
   for (LONG l=0; l<pNode->lAnz_Polys; l++) {
      // Klassifiziere Polygon auf welcher Seite der Ebene es liegt
      nKlasse = Klassifiziere_Polygon(pNode->xEbene, pNode->pPolys[l]);

      if (nKlasse == BSP_FRONT) {
         Füge_Polygon_ein(xFrontnode.pPolys, pNode->pPolys[l]);
         xFrontnode.lAnz_Polys++;
         }
      else if (nKlasse == BSP_BACK) {
         Füge_Polygon_ein(xBacknode.pPolys, pNode->pPolys[l]);
         xBacknode.lAnz_Polys++;
         }
      else if (nKlasse == BSP_SPAN) {  // Teile das Poly in a und b
```

```
         XPOLY xPolyA, xPolyB;
         Splitte_Polygon(pNode->xEbene, pNode->pPolys[l], &xPolyA, &xPolyB);
         Füge_Polygon_ein(xFrontnode.pPolys, pPolyA);
         xFrontnode.lAnz_Polys++;
         Füge_Polygon_ein(xBacknode.pPolys, pPolyB);
         xBacknode.lAnz_Polys++;
         }
      else if (nKlasse == BSP_PLANAR) {
         if (Poly_selbe_Ausrichtung_wie_Ebene(pNode->pPolys[l], pNode->xEbene)) {
            Füge_Polygon_ein(xFrontnode.pPolys, pPolyA);
            xFrontnode.lAnz_Polys++;
            }
         else {
            Füge_Polygon_ein(xBacknode.pPolys, pPolyA);
            xBacknode.lAnz_Polys++;
            }
         }
      } // for
   pNode->pFront = &xFrontnode;
   pNode->pBack  = &xBacknode;

   // Erzeuge die Bounding Box für die Polygone
   pNode->xBox = Erstelle_BoundigBox(pNode->pPolys, pNode->lAnz_Polys);

   // Die Polygonliste eines Nodes brauchen wir nicht mehr
   free(pNode->pPolys);
   pNode->pPolys = NULL;

 // Rufe Funktion rekursiv für Kinder auf
 Erstelle_BSP_Baum(pNode->pFront);
 Erstelle_BSP_Baum(pNode->pBack);
 } // Erstelle_BSP_Baum
```

Diese mächtige Funktion übernimmt eigentlich nur zwei Aufgaben. Zuerst muss sie aus der ungeordneten Suppe der Polygonliste ein Polygon auswählen, welches sich am besten als Teilungspolygon eignet. Dieses nennen wir ab jetzt aber nicht mehr Teilungspolygon, sondern Splitter - schliesslich handelt es sich dabei jetzt um eine Ebene und kein Polygon mehr. Die zweite Aufgabe der Funktion ist es dann, alle Polygone aus der Liste dieses Nodes in eine Front- und in eine Backliste (*also die neuen beiden Kinder*) zu teilen. Dies ist natürlich nur nötig, wenn wir einen Splitter gefunden haben. Ist das nicht der Fall, dann bildet die Liste der Polygone eine konvexe Menge und wir definieren diesen Node als Leaf.
Die Klassifizierungsfunktion entscheidet dann, auf welcher Seite des Splitters ein Polygon aus der Liste liegt. Im Falle dass es davor oder dahinter liegt sortieren wir es in die entsprechende Liste des neuen Astes ein. Liegt das Polygon auf beiden Seiten (=*spanning*) so müssen wir das Polygon in zwei neue Polygone teilen, eins vor und eins hinter der Ebene, die dann jeweils in die entsprechende Liste kommen. Im Falle eines planaren Polygons welches also in derselben Ebene wie der Splitter liegt, prüfen wir wie oben besprochen in welche Richtung das Polygon blickt und sortieren es in die entsprechende Liste ein.
Danach berechnen wir noch schnell eine Bounding Box um alle Elemente die in diesem Node vertreten waren. Nun können wir die Liste der Polygone für den aktuellen Node löschen, denn die brauchen wir nie wieder. Die eigentlichen Polygone sind an die beiden Kinderäste weitergegeben worden und eine Bounding Box um die Polygone haben wir auch schon gespeichert.

Und jetzt der Clou: Die Erstellung der beiden Kinderäste und deren Kinderästen und deren Kinderästen und so weiter läuft doch genau so ab. Daher können wir die Funktion selbst wieder aufrufen, jeweils für die beiden Kinderäste. Dort wird dann ebenfalls ein Splitter ausgewählt, die Polygone auf zwei Liste sortiert usw. Wenn eine Funktion sich selbst aufruft dann nennt man das übrigens **Rekursion**. Erst wenn jeder Unterast des BSP Baumes in einem Leaf geendet hat dann wird ein Zweig des rekursiven Aufrufes beendet. Sind alle Äste mit

Leafs abgeschlossen, dann ist unser gesamter BSP Baum fertig erstellt.

STOP!
Die obige Funktion ist der Kern des gesamten BSP Baumes und jeder sollte sie an dieser Stelle verstanden haben. Es ist im Moment noch unwichtig zu wissen, wie wir eigentlich den besten Splitter finden oder Polygone klassifizieren, das ist Detailarbeit. Aber den eigentlichen Ablauf der Funktion sollte man schon verstanden haben. Wer damit jetzt noch Probleme haben sollte, der referiert am besten noch einmal zurück zu dem kleinen Demolevel welches wir weiter oben Schritt für Schritt zerlegt haben. Dort haben wir genau die Schritte ausgeführt die diese Funktion hier jetzt erledigt...nix anderes!

Und bevor man jetzt noch schlimmeres hinter den anderen Funktionen vermutet kann ich Euch beruhigen: Lediglich die Funktion zum Splitten eines Polygons ist nicht ganz trivial. Aber auch dazu kommen wir noch!

Auswahl des besten Splitters in Pseudocode

Machen wir uns an die Auswahl des besten Splitters. Diese ist wirklich mehr als trivial und wir haben eigentlich schon alles besprochen. Wir haben eine Suppe von Polygonen und wollen aus diesen ein Polygon auswählen dessen Ebene wir als Splitter verwenden werden. Wenn wir einfach irgendein beliebiges Polygon auswählen so haben wir ein paar Probleme am Hals. Vor allem wird unser Baum sehr unausgewogen. Der Splitter wird ja später dazu benutzt, die Suppe aus Polygonen in zwei Teile zu teilen die dann zwei neue Kinderäste im BSP Baum bilden. Wählen wir unsere Splitter so, dass viele Polygone auf einer Seite (*Front oder Back*) liegen und nur wenig Polys auf der anderen Seite, so wird unser Baum sehr unausgewogen. Das ist unschön und ungewünscht weil das Durchlaufen des Baumes, beispielsweise beim Rendern, dann länger dauert als nötig weil der Baum tiefer ist als idealerweise erforderlich.
Ebenso ist unser zweites Kriterium für einen guten Splitter zu berücksichtigen. Der Splitter soll wie eben besprochen bestenfalls in zwei gleich grossen Listen von Polygonen resultieren. Aber wir wollen auch so wenig wie möglich Polygone zerlegen. Der Splitter muss also auch so gewählt werden, dass möglichst wenig Polygone durch die Ebene des Splitters in zwei Teile zerlegt werden müssen (*was in diesem Zusammenhang als Splitten von Polygonen bezeichnet wird*).

Eventuell sind diese beiden Kriterien sogar gegensätzlich. Eine Teilungspolygonebene könnte unsere Polygonliste beispielsweise genau in zwei gleich grosse Hälften teilen, aber sehr viele Polygonsplits verursachen. Ein anderes Polygon dagegen könnte 0 Polygonsplits verursachen, aber sehr ungleiche Listen erstellen wo eine Liste nur halb so viele Polygone enthält wie die andere. Beide potentielle Splitter wären nicht sehr gut geeignet. In der Regel verwendet man eine Formel in folgender Form:

```
Punkte = abs(Anz_Front - Anz_Back) + (Anz_Splits * 3);
```

Man prüft <u>*jedes*</u> Polygon der Liste als Splitter. Dabei zählt man jeweils wie viele Polygone ein Splitter in die beiden isten (Front und Back) einsortieren würde und wie viele Polygonsplits er verursachen würde. Aus diesen Werten erstellt man einen Punktewert für jeden möglichen Splitter und am Ende vergleicht man die Punktewerte aller Splitter. Derjenige mit dem kleinsten Wert ist der beste Splitter.
Betrachten wir die obige Formel. Das erste Kriterium wird durch die Subtraktion berücksichtigt. Immer wenn die Anzahl der Polygone in der Front- und in der Backliste gleich sind, dann wird das erste Kriterium 0 und ist gut für einen Splitter. Je mehr die Anzahl in beiden Listen abweicht, um so mehr Punkte erhält ein Splitter (*was schlecht für ihn ist*). Das zweite Kriterium berücksichtigen wir, indem wir die Anzahl der Polygonsplits zählen und zu dem bisherigen Punktewert addieren. Je mehr Splits um so schlechter für den Splitter.
Wir hatten aber auch bereits geklärt, dass das Rendern der Polygone der Bottleneck jeder 3D Anwendung ist. Es ist daher schlimmer, wenn ein Splitter viele Polygone splittet als wenn er die Polygone ungleichmässig in Front- und Backliste einsortiert. Daher multiplizieren wir die Anzahl der Splits noch mit einem Faktor (*hier 3*) um jeden Polygonsplit eines Splitters strenger zu bestrafen als eine ungleiche Sortierung. Dieser Faktor könnte auch weniger oder mehr sein, je nach Gewichtung die man setzen möchte.

Nun zu den Implementierungsdetails: Wir nehmen einfach die Liste von Polygonen und lassen eine Schleife über sie laufen. In jedem Schleifendurchlauf wählen wir ein neues Polygon aus der Liste und markieren es als aktuellen Splitter. Dann starten wir wieder eine Schleife über alle Polygone der Liste in der Schleife. Nun

klassifizieren wir alle Polygone für den aktuellen Splitter und erstellen seinen Punktewert.
Am Ende der beiden Schleifen haben wir dann alle Polygone der Liste als Splitter getestet und dasjenige gefunden, welches die wenigsten Punkte kassiert hat und damit der beste Splitter ist. *Einfacher als Pfannkuchen essen, oder?*

```
XEBENE Finde_besten_Splitter(XPOLY *pPolys, LONG lAnz_Polys) {
   XEBENE xBestSplitter, Akt_Splitter;

   for (LONG i=0; i<lAnz_Polys; i++) {
      Akt_Splitter = pPolys[i].xEbene;

      for (LONG j=0; j<lAnz_Polys; j++) {
         // Klassifiziere Polygon auf welcher Seite der Ebene es liegt
         nKlasse = Klassifiziere_Polygon(Akt_Splitter, pPolys[j]);

         // Erhöhe die entsprechenden Zähler und setze xBestSplitter
         // auf Akt_Splitter falls dessen Punktewert kleiner

         } // for [AnzPolys]
      } // for [AnzPolys]

   return xBestSplitter;
   } // Finde_besten_Splitter
```

Okay, den ganzen Zählerkrams habe ich hier ignoriert, das sehen wir nachher im echten Quellcode. Aber jetzt haben wir eine sehr gute Vorstellung davon, was diese Funktion tut. Eigentlich ist das nicht viel. Wir nehmen einfach jedes Polygon der Liste und prüfen dann alle Polygone damit durch um zu sehen ob es ein guter Splitter wäre oder nicht. Keine komplexen Berechnungen mit wilden mathematischen Formeln, sondern der gute alte *Brute-Force* Ansatz.

Splitten eines Polygons in Pseudocode

Das Splitten eines Polygons...irgendwo muss ja ein Haken sein da bis hier alles so einfach war. Eigentlich ist das Splitten an sich gar nicht so schwer, aber die Implementierung aller feinen Details wird schon sehr haarig so dass diese Funktion die komplexeste unseres gesamten Demos wird. Das hängt einfach damit zusammen, dass wir einerseits berücksichtigen müssen dass wir die Polygondaten irgendwie in Dreiecken halten müssen um Direct3D zufrieden zu stellen und dass wir auch Texturkoordinaten neu berechnen müssen und alle solche Lapalien halt. Das leppert sich ganz schön was zusammen und das hben wir uns für später auf. Hier also nur ein paar Pseudoworte was man im Grunde genommen machen muss:

```
void Splitte_Polygon(XEBENE xEbene, XPOLY xPoly, XPOLY *xPolyA, XPOLY *xPolyB) {

   for (int i=0; i<xPoly.AnzVerts; i++) {
      // Klassifiziere aktuellen Vertex mit xEbene
      // Falls in Front füge Vertex i zu neuem Poly A
      // Sonst Vertex i zu neuem Poly B

      // Falls Linie von Vertex i-1 zu Vertex i die Ebene
      // schneidet erstelle einen neuen Vertex direkt auf
      // der Ebene und füge ihn in Poly A und B ein.
      } // for
   } // Splitte_Polygon
```

Wir man sieht ist auch das nix dramatischen an sich. Alle Vertices eines Polys werden durchlaufen und sortiert, je nachdem ob sie vor der hinter dem Splitter liegen. Das ist im Prinzip genau dasselbe wie das Teilen der Polygonliste in Front und Back. An den Stellen wo das Polygon aber die Ebene schneidet müssen wir einen

neuen Punkt genau auf der Ebene erstellen und diesen Punkt den beiden neuen Polys A und B zuordnen. Damit haben wir dann zwei neue Polygone auf je einer Seite der Ebene.

Und jetzt geht es los

Bereits Sun Tzu hat es gewusst und seinem Buch *The Art of War* zur Grundlage gemacht: Ohne eine gute Strategie werden wir keine Schlacht gewinnen, am wenigsten die Schlacht um den BSP Baum oder die Schlacht um das Verständnis des Lesers. Es gibt sicherlich viele Strategien wie man den folgenden Code erklären kann. Man kann entweder Bottom-Up vorgehen und die kleinsten Teile wie das Splitten von Polygonen zuerst entwickeln. Danach werden dann alle kleinen Teile zum grossen Ganzen zusammengesetzt.
Ich habe mich hier aber für den gegenteiligen Ansatz entschieden und werden das BSP Demo Top-Down entwickeln. Dass bedeutet ich werde mit der Initialisierung es Baumes beginnen und dabei die Existenz der ganzen kleinen Hilfsfunktionen als gegeben voraussetzen. Die Funktion `Split_Poly()` wird beispielsweise dazu da sein ein Polygon an einer Ebene zu teilen, ob wir diese Funktion vorher bereits implementiert haben oder nicht halte ich nicht für so entscheidend. Jeder kann schnell einsehen wozu die Funktion da ist und wir werden sie erst mal als Black Box verwenden. Erst später gehen wir dann in die ganzen dreckigen Details und schreiben diese Funktionen dann. Diese Strategie halte ich für besser was die Verständlichkeit betrifft. Also sammeln wir unsere Truppen und beginnen die Schlacht.

Inspektion der Truppen vor der Schlacht

Was brauchen wir als erstes für unser kleines BSP Projekt? Falsch, diesmal sind es nicht die Datenstrukturen, sondern ein paar Definitionen die ich hier gleich mal in den Raum schmeisse damit wir sie nachher gleich griffbereit haben:

```
// Maximale Baumgrösse
#define BSP_NODES_MAX 10000

// Polygon Klassifizierungen
#define BSP_FRONT 0
#define BSP_BACK 1
#define BSP_PLANAR 2
#define BSP_SPANNING 3

// Rechengenauigkeit:
#define BSP_DELTA 0.00001

// Für die Polygone:
#define MAX_VERTICES 20
#define MAX_INDICES 40
#define MAX_TEXTUREN 256
```

Ich habe mich hier dazu hinreissen lassen die maximale Grösse des Baumes doch als Konstante Zahl für ein statisches Array zu wählen. Natürlich sollten wir auch hier in einem echten Life-Fire Programm mit dynamischem `realloc()` arbeiten (*siehe vorheriges Kapitel*), das macht den Code aber nicht gerade überschaubarer. Okay, die Baumgrösse gibt also an, wie viele Elemente (*Wurzel, Nodes und Leafs*) unser Baum insgesamt haben darf.

Die Polygonklassifizierungen drücken aus auf welcher Seite einer Ebene ein Polygon oder ein Punkt liegen. Vor oder hinter der Ebene (*Front, Back*), oder genau in der Ebene (*Planar, hier würde mein Mathelehrer vermutlich wieder zum Schlag ausholen und lieber ein Coplanar hören wollen*) oder ein Polygon kann die Ebene schneiden und damit auf beiden Seiten der Ebene existieren (Spanning). Letzteres geht logischerweise nicht für Punkte, es sei denn es handelt sich um tunnelnde Elementarteilchen oder eine extreme Form der

Heisenbergschen Unschärferelation. Aber die Extremphysik lassen wir hier mal aussen vor.

Nun zur Rechengenauigkeit. Wenn wir bestimmen wollen ob ein Punkt oder ein Polygon auf dieser oder jenen Seite einer Ebene liegt so müssen wir dafür (*wer hätte das gedacht*) mathematische Berechnungen anstellen. Hierbei können wir durchaus mit sehr kleinen Kommazahlen zu tun bekommen und selbst der Datentyp `float` mit seinen acht Kommastellen reich hier nicht aus, um Rechenfehler durch Rundungen zu vermeiden. Wir definieren also einen Wert ab dem wir uns zufriedengeben. Sagen wir maleine Rechnung muss 0 ergeben wenn ein Punkt genau in einer Ebene liegt, kleiner als 0 für hinter der Ebene und grösser 0 für vor der Ebene. Mit der von uns gewählten Genauigkeit definieren wir nun, dass der Punkt hinter der Ebene liegt wenn das Ergebnis kleiner als -`BSP_DELTA` ist, vor der Ebene wenn das Ergbenis grösser als +`BSP_DELTA` ist und in der Ebene wenn das Ergebnis zwischen -`BSP_DELTA` und +`BSP_DELTA` liegt.

Zu guter Letzt haben wir noch definiert wie viele Vertices und Indices ein Polygon bei uns haben darf, das sehen wir dann später in der Polygondatenstruktur. Dazu noch die Anzahl von maximal möglichen Texturen. Und weil wir gerade dabei sind sehen wir uns mal zwei wichtige globale Variablen an. Ein Array für die Elemente des Baumes und einen Zähler wie viele Elemente sich in diesem Array gerade aufhalten. Immer wenn wir dem Baum einen neuen Node hinzufügen so erhöhen wir den Zähler und nehmen dessen Wert als Indes in das Array an dem sich der Neue Node befindet.

```
XNODE g_aNode_Pool[BSP_NODES_MAX];
LONG  g_lPool_Index=0;
```

Keine Panik das wird später schnell klar. Erst mal sehen wir uns aber die wohl wichtigste Datenstruktur für den BSP Baum an, nämlich die eines Polygons, also jener Vielecke aus denen sich die Levelgeometrie zusammensetzt.

```
/**
 * Struktur für ein Polygon
 */
typedef struct XPOLY_TYP {
   D3DLVERTEX vVerts[MAX_VERTICES];  // Vertices
   LONG       lAnz_Verts;            // Anzahl der Vertices
   WORD       wIndices[MAX_INDICES]; // Indices auf Vertices
   LONG       lAnz_Indices;          // Anzahl der Indices
   XEBENE     xEbene;                // Ebene des Polys
   BOOL       blnWar_schon_Splitter; // Schon mal Splitter gewesen?
   UCHAR      nTextur_Index;         // Textur des Polys
   } XPOLY;
/*--------------------------------------------------------*/
```

Die meisten Felder der Struktur sind selbsterläuternd. Es gibt eine Liste von Vertices und eine Liste von Indices aus welchen Dreiecken sich das Polygon zusammensetzt. Dazu berechnen wir zu Beginn für jedes Poly die Ebene in der es liegt da wir diese ja für den BSP Algorithmus benötigen. Dazu brauchen wir ein Feld in dem wir uns merken ob dieses Polygon (*bzw. seine Eben*) schon einmal als Splitter verwendet wurde, das darf ja nur maximal einmal der Fall sein. Zu guter Letzt hat ein Polygon auch immer eine Textur und wir speichern hier den Index den Textur in dem Texturarray der BSP Baum Struktur, die wir nachher noch sehen werden.

Widmen wir uns erst mal der nächstwichtigeren Struktur, nämlich derjenigen die wir für Elemente des Baumes verwenden werden.

```
/**
 * Struktur für Root, Nodes und Leafs des BSP Baumes
 */
typedef struct XNODE_TYP {
   XEBENE     xEbene;    // NODE: Teilungsebene
   UCHAR      blnLeaf;   // NODE oder LEAF
   LONG       lAnz_Polys;// Anzahl der Polygone
```

```
   XPOLY      *pPolys;     // LEAF: Liste der Polygone
   XNODE_TYP *pFront;      // NODE: Frontliste
   XNODE_TYP *pBack;       // NODE: Backliste
   XBOX       xBox;        // Bounding Box
   } XNODE;
/*-------------------------------------------------------*/
```

Wie wir sehen speichern wir an einem Node erst mal eine Ebene, das ist die wichtigste Information für den Node und sagt aus an welcher Ebene dieser spezielle Node das Level teilt. Als nächstes folgt eine Variable die aussagt ob es sich bei dem Baumelement um ein Leaf handelt. Dann haben wir einen Zähler wie viele Polygone sich in dem Leaf befinden. Handelt es sich um ein inneres Baumelement so sagt dieser Zähler aus wie viele Polygone in allen Leafs unter ihm stehen. Das brauchen wir zwar nicht unbedingt, es ergibt sich aber bei der Baumkonstruktion sowieso.
Als nächstes haben wir dann eine Liste aus Polygonen. Nach der Baumerstellung ist dieses Element nur für Leafs gültig. Nur für einen inneren Node gelten hingegen die beiden vorletzten Elemente die die Zeiger auf die beiden Kindern des Nodes sind. Zu guter Letzt haben wir die Bounding Box des Nodes die die gesamte Geometrie in den Leafs unter einem Node umschliesst. Das letzte was wir jetzt noch brauchen ist eine Struktur für die Texturen:

```
/**
 * Struktur für die Texturen
 */
typedef struct XTEXTUR_TYP {
   LPDIRECT3DTEXTURE8 lpTextur;
   char               achName[128];
   } XTEXTUR;
/*--------------------------------*/
```

Diese Struktur benötigen wir nur damit wir Texturen nicht mehrfach laden. Wir speichern alle verwendeten Bitmapgrafiken als Direct3D Texturobjekte in dieser Struktur zusammen mit dem Namen der Grafikdatei. Wie das genau funktioniert sehen wir dann später beim Laden der Texturen.

Erkunden des Schlachtfeldes

Der aufmerksame Leser wird festgestellt haben, dass uns bisher noch eine Struktur für den BSP Baum fehlt. Oder brauchen wir gar keine Struktur dafür weil ich alles irgendwie in C Funktionen quetschen werde? Nein, nein. Diesmal gibt es eine echte Klasse für den BSP Baum in der alles drinsteckt was nötig ist. Sehen wir uns diese Klassenfunktion jetzt mal an. Ich habe mich bemüht alles so zu kommentieren dass man gleich sehen kann was welchen Sinn erfüllt. Die einzelnen Funktionen werden wir uns dann Stück für Stück vornehmen.

Als erstes wird uns auffallen, dass diese Klasse (*C++ Klassen sind für uns Pragmatiker nichts anderes als C Strukturen in die wir auch Funktionen stopfen können*) drei öffentliche Funktionen hat die unser Programm beliebig aufrufen darf. Wie es der Zufall so will gibt es je eine Funktion für jede Phase des Spielablaufes. Ein Funktionsaufruf erstellt den Baum, der zweite Aufruf rendert den Baum und der dritte gibt den Speicher frei. Hier erst mal die vollständige Klassendefinition:

```
/**
 * Die Klasse des BSP Baumes:
 */
class XBSP {
   public:
      // Member Funktionen:
      // ==================
      BOOL Init(char *achDateiname);// PHASE 1: starten
      void Render(void);            // PHASE 2: Schleife
      void Kill(void);              // PHASE 3: beenden
```

```
   protected:
      // Member Variablen:
      // =================
      XNODE    xWurzel;
      UCHAR    Anz_Texturen;
      XTEXTUR aTextur[MAX_TEXTUREN];

      // Member Funktionen:
      // ==================
      // Laden der Leveldaten aus Datei
      BOOL  Lade_Leveldaten(char *achDateiname);
      UCHAR Textur_Manager(char *achDateiname);

      // Rekursive Erstellung des Baumes:
      void Erstelle_BSP_Baum(XNODE *pNode);
      XBOX Erstelle_BoundingBox(XPOLY *Polyliste, LONG lAnz_Polys);
      BOOL Finde_besten_Splitter(XPOLY *Polyliste,
                                 LONG lAnz_Polys,
                                 XEBENE *Splitter);

      // Rendern des fertigen Baumes
      void Render_rekursiv(XNODE *pNode, D3DVECTOR vPosition);
      void Render_Leaf(XNODE *pNode);

      // Hilfsfunktionen für 3D Mathemathik
      int Klassifiziere_Punkt(XEBENE xEbene, D3DVECTOR vPunkt);
      int Klassifiziere_Poly(XEBENE xEbene, XPOLY *pPoly);
      XEBENE Ebene_von_Poly(XPOLY *pPoly);
      void   Split_Poly(XPOLY *Poly, XEBENE *pEbene,
                        XPOLY *FrontSplit, XPOLY *BackSplit);

      // BSP Baum Speicher freigeben
      void Kill_rekursiv(XNODE *pNode);
   };
/*-----------------------------------------------------*/
```

Okay, was die privaten Memberfunktionen (*die nur von Funktionen der Klasse selbst aufgerufen werden dürfen*) tun kann man eigentlich am Namen her erkennen. Jedenfalls hoffe ich mal dass Ihr ein nach der Lektüre des BSP Algorithmus sehen könnt was wozu da ist. Wir gehen das alles gleich im Detail durch, wichtig sind aber zunächst die drei Variablen der Klasse. Das `xWurzel` Feld ist der Startpunkt des gesamten Baumes. Dazu gibt es nur noch einen Zähler wie viele Texturen das Level verwendet und dann ein Array in dem sich die Texturen befinden. Na dann wollen wir unseren Level mal laden, oder?

Aufmunitionieren

Unsere BSP Klasse ist so designed, dass man später mit wenig Aufwand den Baum erstellen kann und das sieht dann im Programm wie folgt aus:

```
XBSP BSP_Baum;
BOOL blnErg;

blnErg = BSP_Baum.Init("testlevel.zfx");
if (blnErg == FALSE) {
   fprintf(Protokoll,"SpielInit: BSP Baum failed \n");
```

```
   return FALSE;
   }
```

Wow, cool. Das riecht aber ganz verdächtig danach, dass die Initialisierungsfunktion mehr macht als man auf den ersten Blick vermuten würden. Sehen wir sie uns also genau an um unsere Vermutung zu bestätigen:

```
BOOL XBSP::Init(char *achDateiname) {
   BOOL blnErgb;

   fprintf(Protokoll, "\nStarte BSP Baum Erstellung { \n");

   // Noch keine Texturen
   this->Anz_Texturen = 0;

   // Lade die Leveldaten und erstelle Wurzel-Polyliste
   blnErgb = this->Lade_Leveldaten(achDateiname);
   if (!blnErgb) {
      fprintf(Protokoll, "Fehler: BSP Lade_Level() failed\n");
      return FALSE;
      }

   // Erstelle den Baum von der Wurzel ausgehend
   this->Erstelle_BSP_Baum(&this->xWurzel);

   fprintf(Protokoll, " BSP Baum erstellt: \n");

   return TRUE;
   } // Init
/*-------------------------------------------------------------*/
```

Wie man sieht passiert auch hier noch nix dramatisches. Wir verzweigen die Arbeit einfach auf zwei weitere Funktionen, nämlich einmal das Laden des Levels und zum anderen das Erstellen des BSP Baumes aus den geladenen Leveldaten. In diesem Abschnitt setzen wir uns nun damit auseinander wie wir die Leveldaten laden. Dazu müssen wir uns natürlich erst mal auf ein Format einigen in denen die Leveldaten vorliegen sollen und das definiere ich wie folgt (*in Form einer ASCII Textdatei*):

```
int
float float float float float
[...]
int
int int int hex
[...]
char
```

Die Textdatei listet einfach alle Polygone des Levels nacheinander auf und die Polygoninformationen sind wie folgt aufgebaut. Als erstes kommt eine Zahl die angibt wie viele Vertices dieses Polygon hat. Danach folgen diese Vertices mit ihren drei Koordinaten x, y und z sowie zwei weiteren Werten für die Texturkoordinaten u und v. Nachdem dann alle Vertices definiert wurden folgt eine Zahl die angibt aus wie vielen Dreiecken das Polygon besteht, Direct3D kann schliesslich nur Dreiecke rendern. Danach folgen dann jeweils die Dreiecke in Form von je drei Indices auf die oben definierten Vertices und zusätzlich ein Hexadezimalwert für die Farbe des Dreiecks. Abschliessend steht dann der Name der verwendeten Grafikdatei für die Textur des Polygons. Nun wiederholt sich das ganze Spielchen für jedes Polygon der Datei.

Easy, aber woher nehmen wir diese Dateien mit Leveldaten? Nun, ich habe AC3D zusammen mit einem selbstdefinierten Export Plugin verwendet um meine Testdaten zu erstellen. Das Plugin ist im Download enthalten, wer also über AC3D verfügt der kann bequem seine eigenen Level bauen. Und nun zur Funktion die uns die Leveldaten in unsere Datenstrukturen einliest.

Diese Funktion tut im Grunde genommen nichts anderes als einen Pointer vom Typ `XPOLY` zu definieren. Diesem wird dynamischer Speicher zugewiesen um ihn zu einem dynamischen Array zu machen so wie im letzten Kapitel besprochen. Wir öffnen also unsere Leveldatei, die ja nix anderes ist als eine grosse Liste von Polygondaten, und lesen dann die einzelnen Daten in die korrespondierenden Felder der Polygonliste. Zu guter Letzt speichern wir die fertige Polygonliste dann an der Wurzel des BSP Baumes:

```
BOOL  XBSP::Lade_Leveldaten(char *achDateiname) {
   DWORD  dwFarbe;
   FILE  *pDatei;
   XPOLY *vPolygon;
   int    n=0, nAnz_Dreiecke, nIndex;
   char   buffer[8], chFarbe[8], chTextur[50];

   // Öffne die Leveldatei
   pDatei = fopen(achDateiname, "r");
   if (!pDatei) {
      fprintf(Protokoll, "Fehler: BSP Level nicht gefunden\n");
      return FALSE;
      }

   // Stelle Speicher für die Verts bereit
   vPolygon = (XPOLY*)malloc(sizeof(XPOLY)*100);
   if (!vPolygon) {
      fprintf(Protokoll, "Fehler: malloc() Wurzel Verts\n");
      return FALSE;
      }

   // Hole alle Polygone aus der Leveldatei
   while (!feof(pDatei)) {
      // Anzahl der Vertices einlesen
      fscanf(pDatei, "%d", &vPolygon[n].lAnz_Verts);

      // Koords und Tex-Koords der Vertices einlesen
      for (LONG l=0; l<vPolygon[n].lAnz_Verts; l++) {
        fscanf(pDatei, "%f %f %f %f %f\n",
           &vPolygon[n].vVerts[l].x,  // Koordinaten
           &vPolygon[n].vVerts[l].y,
           &vPolygon[n].vVerts[l].z,
           &vPolygon[n].vVerts[l].tu, // Text-Koords
           &vPolygon[n].vVerts[l].tv);
           } // for [Verts im Poly]

      // Anzahl der Dreiecke im Polygon bestimmen
      fscanf(pDatei, "%d", &nAnz_Dreiecke);
      vPolygon[n].lAnz_Indices = 0;

      // Indices und Farbe der Dreiecke einlesen
      for (LONG m=0; m<nAnz_Dreiecke; m++) {
         fscanf(pDatei, "%d %d %d %s\n",
            &vPolygon[n].wIndices[vPolygon[n].lAnz_Indices+0],
            &vPolygon[n].wIndices[vPolygon[n].lAnz_Indices+1],
            &vPolygon[n].wIndices[vPolygon[n].lAnz_Indices+2],
            &chFarbe);

         sprintf(buffer, "0x%s", chFarbe);
         sscanf(buffer, "%i", &dwFarbe);
```

```
        // Speichere die Farbe für die drei Vertices des Dreiecks
        nIndex = vPolygon[n].lAnz_Indices;
        vPolygon[n].vVerts[vPolygon[n].wIndices[nIndex+0]].color =
           dwFarbe;
        vPolygon[n].vVerts[vPolygon[n].wIndices[nIndex+1]].color =
           dwFarbe;
        vPolygon[n].vVerts[vPolygon[n].wIndices[nIndex+2]].color =
           dwFarbe;

        // Drei Indices zugefügt
        vPolygon[n].lAnz_Indices += 3;
        } // for [Indices im Poly]

     // Textur laden und Index im Poly speichern
     fscanf(pDatei, "%s\n", &chTextur);
     vPolygon[n].nTextur_Index = this->Textur_Manager(chTextur);

     // Als unbenutzt markieren
     vPolygon[n].blnWar_schon_Splitter = FALSE;

     // Ebene des Dreiecks berechnen
     vPolygon[n].xEbene = this->Ebene_von_Poly(&vPolygon[n]);

     n++; // nächstes Dreieck

     // Falls Platz im Array nicht reicht:
     if ( (n%100) == 0) {
        vPolygon = (XPOLY*)realloc(vPolygon,
                               sizeof(XPOLY)*(n+100));
        if (!vPolygon) {
           fprintf(Protokoll, "Fehler: realloc() \n");
           return FALSE;
           }
        }
     } // while

  // endgültige Arraygrösse festlegen
  vPolygon = (XPOLY*)realloc(vPolygon, sizeof(XPOLY)*n);

  // Anzahl der Polygone und die Polyliste speichern
  this->xWurzel.lAnz_Polys = n;
  this->xWurzel.pPolys     = vPolygon;

  return TRUE;
  } // Lade_Leveldaten
/*--------------------------------------------------------*/
```

Die Funktion `fscanf()` ist eigentlich selbsterklärend, oder? Sie funktioniert sozusagen wir `fprintf()` nur umgekehrt und lädt Daten aus einer Textdatei in Variablen des Programms. Ich habe hier aber noch Gebrauch von zwei weiteren unserer Funktionen machen müssen. Zum einen benötigen wir einen Texturmanager, den wir uns gleich ansehen, und zum anderen benötigen wir die Funktion die die Ebene eines Polygons berechnet. Wir glauben hier einfach mal ganz spontan dass das mit rechten Dingen zugeht, denn die Funktion ist erst weiter unten an der Reihe.

Jetzt zum Sinn und Zweck des Texturmanagers. Selbst wenn unser Level Tausende von Polygonen verwendet

so ist es doch recht unwahrscheinlich, dass wir auch Tausende von verschiedenen Texturen haben. In der Regel werden wir mit zwanzig oder dreissig verschiedenen Texturen für die Levelarchitektur auskommen. Es ist daher Blödsinn und Speicherverschwendung diese wenigen Texturen Tausende Male zu laden. Jede Textur einmal zu laden reicht vollkommen aus. Die Texturen unseres Levels werden also jeweils nur einmal in das entsprechende Array der BSP Klasse geladen und jedes Polygon erhält statt des Texturobjektes nur einen Index in dieses Array an dem die passende Textur steckt. Lange Rede kurze Funktion:

```
UCHAR XBSP::Textur_Manager(char *achDateiname) {
   HRESULT hr;
   UCHAR   i;

   // Prüfe ob die Textur schon einmal geladen wurde
   for (i=0; i<this->Anz_Texturen; i++) {
      // Ja => also den Index auf Texturarray speichern
      if (lstrcmpi(&this->aTextur[i].achName[0], achDateiname) == 0)
         return i;
      } // for

   // Lade neue Textur in BSP Texturarray
   hr = D3DXCreateTextureFromFileA(g_lpD3DDevice, // D3D Device
                                   achDateiname,  // Grafikdatei
                                   &this->aTextur[i].lpTextur);
   if (FAILED(hr)) {
      fprintf(Protokoll, "Fehler: Textur %s laden\n", achDateiname);
      return 0;
      }

   // Kopiere den BMP Namen in das Texturarray
   lstrcpyn(&this->aTextur[i].achName[0], achDateiname, 128);

   // Zähle Anzahl Texturen mit
   this->Anz_Texturen++;

   // Gib Index der Textur zurück;
   return i;
   } // Textur_Manager
/*-------------------------------------------------------*/
```

Die Funktion durchläuft einfach alle bisher in den BSP Baum geladenen Texturen und vergleicht deren Namen mit dem Namen der Textur die geladen werden möchte. Die Funktion `lstrcmpi()` liefert den Wert 0 zurück falls die beiden Strings übereinstimmen. Entdecken wir die Textur bereits unter den zuvor geladenen so geben wir den Index in das Array zurück an dem wir die Textur finden können. Diesen Wert merkt sich das Polygon dann ganz einfach.
Finden wir die Grafikdatei noch nicht unter den Texturen so laden wir sie als neues Element in das Array des BSP Baumes, speichern auch ihren Namen dort, erhöhen den Zähler der geladenen Texturen und geben den entsprechenden Index an das Polygon zurück.

So zurück zum Laden der Leveldaten. Da hatten wir ja noch eine Hilfsfunktion verwendet die wir bisher noch nicht gesehen haben. Zu jedem Polygon welches wir aus der Leveldatei geladen haben und das wir als `XPOLY` speichern müssen wir auch dessen Ebene berechnen und uns merken. Im vorherigen Kapitel haben wir ja die Ebenenformel kennengelernt und diese müssen wir jetzt einfach für jedes Polygon erzeugen:

```
XEBENE XBSP::Ebene_von_Poly(XPOLY *pPoly) {
   XEBENE xEbene;

   D3DVECTOR v1, v2;
```

```
   v1.x = pPoly->vVerts[1].x-pPoly->vVerts[0].x;
   v1.y = pPoly->vVerts[1].y-pPoly->vVerts[0].y;
   v1.z = pPoly->vVerts[1].z-pPoly->vVerts[0].z;

   v2.x = pPoly->vVerts[2].x-pPoly->vVerts[0].x;
   v2.y = pPoly->vVerts[2].y-pPoly->vVerts[0].y;
   v2.z = pPoly->vVerts[2].z-pPoly->vVerts[0].z;

   xEbene.vNormal = xUtil_Kreuzprodukt(v1, v2);

   xEbene.vNormal = xUtil_Normalisieren(&xEbene.vNormal);

   xEbene.fDistanz = - (
         pPoly->vVerts[2].x*xEbene.vNormal.x +
         pPoly->vVerts[2].y*xEbene.vNormal.y +
         pPoly->vVerts[2].z*xEbene.vNormal.z);

   return xEbene;
   } // Ebene_von_Poly
/*----------------------------------------*/
```

Keine Überraschungen hier. Wir nehmen einfach die ersten drei Vertices des Polygons und erzeugen uns daraus zwei Vektoren (*Endpunkt minus Startpunkt*) die damit logischerweise in der Ebene des Polygons liegen. Diese *kreuzen* wir dann um einen zu ihnen rechtwinklig gelegenen Vektor zu erhalten. Das ist damit logischerweise der Normalenvektor der Ebene nachdem er Normalisiert wurde. Zu guter Letzt benötigen wir noch einen Wert für -d der die Entfernung der Ebene von Nullpunkt angibt.
Hm...warum nehmen wir diese komische Berechnung da? Die Ebenenformel ist doch `n*x+d=0` oder etwas umgeformt `n*x=-d`. Damit ist die negative Entfernung der Ebene zum Ursprung das Ergebnis einer Multiplikation (*Punktprodukt*) aus einem beliebigen Punkt in der Ebene mit deren Normalenvektor. Statt v2 hätten wir also auch jeden anderen Punkt aus dem Polygon verwenden können.

Die grosse Schlacht

Und jetzt der Moment den wir alle gefürchtet haben. Nachdem wir das Laden der Leveldaten und das Initialisieren der Polygonliste eben abgehakt haben sind wir nun beim letzten Schritt der Funktion `XBSP::Init` angekommen. Und dieser ist das Arbeitspferd und der Kern der BSP Klasse. Jetzt implementieren wir die Funktion `XBSP::Erstelle_BSP_Baum`. Es gilt zu beachten dass die eben besprochene Funktion zum Laden der Leveldaten bereits das Wurzelelement des BSP Objektes initialisiert hat, allem voran mit der Polygonliste. Mit diesem Wurzelelement rufen wir nun das Arbeitspferd auf. Okay, natürlich ist diese Funktion wiederum ein einziges Aufrufen von anderen Hilfsfunktionen aber da müssen wir jetzt durch. So just hang on, ich habe immer kommentiert was die entsprechende Funktion macht und wir nehmen diese hier erst mal als Zerbie-gegeben hin. Erst mal behandeln wir diese Funktion in ihrer Gänze und danach kommen die anderen alle noch dran, versprochen.

```
// Array für alle Elemente des Baumes unter der Wurzel
XNODE g_aNode_Pool[BSP_NODES_MAX];
// Zähler für die erzeugten Elemente
LONG g_lPool_Index=0;

void XBSP::Erstelle_BSP_Baum(XNODE *pNode) {
   XNODE *pFrontnode, *pBacknode;
   LONG   lAnz_F=0, lAnz_B=0;
   BOOL   blnErgb;
   int    nKlasse;

   // Berechne die Bounding Box um alle Polygone des Nodes
```

```
   pNode->xBox = this->Erstelle_BoundingBox(pNode->pPolys,
                                           pNode->lAnz_Polys);
   // Finde die beste Teilungsebene dieses Nodes
   blnErgb = this->Finde_besten_Splitter(pNode->pPolys,
                                        pNode->lAnz_Polys,
                                        &pNode->xEbene);
   // Falls keine Ebene gefunden ist dieser Node ein Leaf
   if (!blnErgb) {
      pNode->blnLeaf = TRUE;
      pNode->pBack   = NULL;
      pNode->pFront  = NULL;
      return;
      }

   // Zu wenig Speicher im Pool für kompletten Baum
   if (g_lPool_Index >= (BSP_NODES_MAX-2)) {
      fprintf(Protokoll, " Fehler: MAX_NODES erreicht \n");
      return;
      }

   pFrontnode = &g_aNode_Pool[g_lPool_Index++];
   pBacknode  = &g_aNode_Pool[g_lPool_Index++];

   pFrontnode->lAnz_Polys = 0;
   pBacknode->lAnz_Polys  = 0;

   // Speicher in ausreichender Menge allokieren
   pFrontnode->pPolys = (XPOLY*)malloc(sizeof(XPOLY)*pNode->lAnz_Polys);
   pBacknode->pPolys  = (XPOLY*)malloc(sizeof(XPOLY)*pNode->lAnz_Polys);
   if (!pFrontnode->pPolys || !pBacknode->pPolys) {
      fprintf(Protokoll, "malloc() Front-/Back failed\n");
      return;
      }
```
wird fortgesetzt...

Zuerst zu den wichtigen deklarierten Variablen. Wir haben je ein Objekt für einen Front- und einen Backnode des aktuellen Nodes und entsprechende Zähler für die Polygone in den Listen. Als erstes berechnen wir dann die Bounding Box für das übergebene `XNODE` Objekt. Zu Beginn ist das natürlich die Wurzel des BSP Baumes. Danach suchen wir aus der Menge der Polygone des Baumelementes den besten Splitter heraus. Finden wir diesen nicht, dann handelt es sich bei der Menge der Polygone in diesem Baumelement um eine konvexe Menge und wir definieren dieses Baumelement als Leaf und beenden die Funktion.

Gut, wenn wir aber einen besten Splitter finden dann war die Polygonmenge eben nicht konvex, sondern noch teilbar und wir verwenden dann die globale Variable `g_lPool_Index` die mitzählt wieviele Elemente unser Baum schon hat und dafür sorgt dass wir nicht mehr als `BSP_NODES_MAX` Elemente erzeugen. Dazu haben wir das globale Array `g_aNode_Pool` in dem wir dann die zwei nächsten freien Positionen jeweils der Front- und der Backliste zuweisen. Die beiden Pointer auf den Front- und den Backnode sind damit nichts anderes als Adressen in das statische, globale Array aus `XNODE` Objekten. Ist auf den ersten Blick vielleicht etwas verwirrend und auf den zweiten Blick etwas unschön, erleichtert hier aber die Arbeit etwas weil wir nicht überall dynamischen Speicher verwenden müssen. Also weiter in der Funktion. Wenn wir kein Leaf haben dann müssen wir logischerweise noch einen ganzen Batzen Arbeit erledigen und dazu kommen wir nun.
Oben ist noch zu sehen dass wir die Polygonzähler der beiden Kinder erst mal auf 0 setzen und dann für die Polygonlisten Speicher allokieren. Dabei erhält jedes Kind des aktuellen Baumelementes genug Platz für so viele Polygone wie im aktuellen Baumelement enthalten sind. Das ist natürlich zu viel Platz, denn im Durchschnitt solte jedes Kind ja nur halb so viel Polygone erhalten. Da wir das aber auf keinen Fall vorher genau bestimmen können (*siehe Theorie zur Auswahl des besten Splitters oben*) gehen wir erst mal auf Nummer Sicher und passen den Speicher erst später richtig an. Okay, dann machen wir mal weiter mit der

Funktion, denn jetzt kommt der spannende Teil **:-)**

```
Erstelle_BSP_Baum() Fortsetzung:
   // Durchlaufe alle Polygone und sortiere sie
   for (LONG l=0; l<pNode->lAnz_Polys; l++) {
      // Klassifiziere Polygon auf welcher Seite der Ebene es liegt
      nKlasse = this->Klassifiziere_Poly(pNode->xEbene,
                                         &pNode->pPolys[l]);

      if (nKlasse == BSP_FRONT) {
         pFrontnode->pPolys[pFrontnode->lAnz_Polys] = pNode->pPolys[l];
         pFrontnode->lAnz_Polys++;
         }
      else if (nKlasse == BSP_BACK) {
         pBacknode->pPolys[pBacknode->lAnz_Polys] = pNode->pPolys[l];
         pBacknode->lAnz_Polys++;
         }
      else if (nKlasse == BSP_SPANNING) {
         XPOLY xPolyFRONT, xPolyBACK;
         // Teile das Poly in a und b
         this->Split_Poly(&pNode->pPolys[l], &pNode->xEbene,
                          &xPolyFRONT, &xPolyBACK);
         // Frontliste
         pFrontnode->pPolys[pFrontnode->lAnz_Polys] = xPolyFRONT;
         pFrontnode->lAnz_Polys++;

         // Backliste
         pBacknode->pPolys[pBacknode->lAnz_Polys] = xPolyBACK;
         pBacknode->lAnz_Polys++;
         }
      else if (nKlasse == BSP_PLANAR) {
         // Gleiche Richtung wie die Ebene oder nicht?
         float Pkt_Produkt;
         Pkt_Produkt = pNode->pPolys[l].xEbene.vNormal.x *
                       pNode->xEbene.vNormal.x +
                       pNode->pPolys[l].xEbene.vNormal.y *
                       pNode->xEbene.vNormal.y +
                       pNode->pPolys[l].xEbene.vNormal.y *
                       pNode->xEbene.vNormal.y;
         // Wohin mit dem Polygon?
         if (Pkt_Produkt >= 0) {
            // Frontliste
            pFrontnode->pPolys[pFrontnode->lAnz_Polys] = pNode->pPolys[l];
            pFrontnode->lAnz_Polys++;
            }
         else {
            // Backliste
            pBacknode->pPolys[pBacknode->lAnz_Polys] = pNode->pPolys[l];
            pBacknode->lAnz_Polys++;
            }
         } // BSP_PLANAR
      } // for
wird fortgesetzt...
```

Wie sagen wir Panzergrenadiere? ***Dran, drauf, drüber!*** Dieser Teil der Funktion sollte uns schwer bekannt vorkommen. Wir durchlaufen alle Polygone des Baumelementes und hetzen unsere magische Klassifizierungsfunktion darauf. Diese spuckt dann einen der vier blau markierten Rückgabewerten aus der

besagt in welcher Relation zur Teilungsebene des besten Splitters sich das jeweilige Polygon befindet. Am einfachsten sind die Fälle **BSP_FRONT** und **BSP_BACK**, hier können wir das Polygon problemlos in die Liste des entsprechenden Kindnodes einsortieren. Im Falle von **BSP_SPANNING** schneidet das Polygon aber die Ebene und muss fachgerecht in zwei Teile zerlegt werden. Zum einen den Teil der vor und zum anderen den Teil der hinter der Splitterebene liegt. Die beiden entsprechenden Teile die durch unsere magische Teilungsfunktion erzeugt werden, werden dann in die entsprechende Liste einsortiert.
Der letzte mögliche Fall ist **BSP_PLANAR** bei dem das Polygon direkt in der Teilungsebene liegt. In diesem Fall müssen wir anhand des Normalenvektors entscheiden ob das Polygon in dieselbe Richtung wie die Ebene blickt oder nicht und es dann wie oben in der Theorie besprochen einsortieren. Das Punktprodukt zwischen den beiden Normalenvektoren gibt unter anderem acuh Informationen über den Winkel der beiden Vektoren zueinander preis...ein Hoch auf das Punktprodukt. Ist das Ergebnis grösser als 0 so ist der Winkel zwischen den beiden Vektoren kleiner als 90 Grad. Damit blickt das Polygon in dieselbe Richtung wie die Ebene. Umgekehrt gilt natürlich das Gegenteil. Und nun die gute Nachricht: Das war eigentlich schon die gesamte Funktion zru Erstellung des Baumes. Fehlt nur noch der Abschluss der Funktion und der rekursive Aufruf:

Erstelle_BSP_Baum() Fortsetzung:

```
   // Speicher genau anpassen:
   pFrontnode->pPolys = (XPOLY*)realloc(pFrontnode->pPolys,
                         sizeof(XPOLY)*pFrontnode->lAnz_Polys);
   pBacknode->pPolys  = (XPOLY*)realloc(pBacknode->pPolys,
                         sizeof(XPOLY)*pBacknode->lAnz_Polys);

   pNode->pFront = pFrontnode;
   pNode->pBack  = pBacknode;

   // Polygonliste des Nodes brauchen wir nicht mehr
   free(pNode->pPolys);
   pNode->pPolys = NULL;

   // Rufe Funktion rekursiv für Kinder auf
   this->Erstelle_BSP_Baum(pNode->pFront);
   this->Erstelle_BSP_Baum(pNode->pBack);

   return;
   } // Erstelle_BSP_Baum
/*----------------------------------------------*/
```

Nach dem Lauf der Schleife wissen wir dann auch ganz genau wie viele Polygone in der Front- und wie viele in der Backliste gelandet sind und passen den Speicher entsprechend an. Nun geben wir die Adressen der jeweiligen `XNODE` Objekte (in dem globalen Array) dem aktuellen Baumelement bekannt, so dass dieses dann Zugriff auf seine Kinder hat. Die Polygonliste eines Nodes der kein Leaf ist wird nie wieder gebraucht, schliesslich ist die Boundig Box alles was wir brauchen. Also blockieren wir keinen überflüssigen Speicher und geben diesen wieder frei. Zu guter Letzt der eigentliche Trick. Für die beiden Kinder des Baumelementes rufen wir dieselbe Funktion auf um die Kinder wiederum in Front und Back Elemente zu teilen, so wie es der BSP Algorithmus halt verlangt. Fertig!

Nebengefechte

So...nun zu den kleinen Nebengefechten. Hier besprechen wir den Pfannkuchenkram, also die Erstellung der Bounding Boxen für eine Menge von Polygonen und die Klassifizierung von Polygonen in Relation zu einer gegebenen Ebene. Fangen wir mit den Boxen an:

```
#define MAX(a,b) ((a<b) ? (b) : (a))
#define MIN(a,b) ((a<b) ? (a) : (b))

XBOX XBSP::Erstelle_BoundingBox(XPOLY *Polyliste,
```

```
                                LONG lAnz_Polys) {
   XBOX xBox;
   XPOLY *pPoly = Polyliste;
   float fMax_x, fMax_y, fMax_z,
         fMin_x, fMin_y, fMin_z;

   // Startwerte beliebig wählen
   fMax_x = fMin_x = pPoly->vVerts[0].x;
   fMax_y = fMin_y = pPoly->vVerts[0].y;
   fMax_z = fMin_z = pPoly->vVerts[0].z;

   for (LONG i=1; i<lAnz_Polys; i++, pPoly++) {
      for (LONG j=0; j<pPoly->lAnz_Verts; j++) {
        // Nach neuem Maximalwert suchen
        fMax_x = MAX(fMax_x, pPoly->vVerts[j].x);
        fMax_y = MAX(fMax_y, pPoly->vVerts[j].y);
        fMax_z = MAX(fMax_z, pPoly->vVerts[j].z);
        // Nach neuem Minimalwert suchen
        fMin_x = MIN(fMin_x, pPoly->vVerts[j].x);
        fMin_y = MIN(fMin_y, pPoly->vVerts[j].y);
        fMin_z = MIN(fMin_z, pPoly->vVerts[j].z);
        } // for [Verts]
     } // for [Polys]

   // Ausdehnung der Box speichern
   xBox.vMax = xUtil_Vector(fMax_x, fMax_y, fMax_z);
   xBox.vMin = xUtil_Vector(fMin_x, fMin_y, fMin_z);

   // Mittelpunkt berechnen
   xBox.vMittelpkt = xUtil_Vector( (fMax_x+fMin_x)/2,
                         (fMax_y+fMin_y)/2,
                         (fMax_z+fMin_z)/2);
   return xBox;
   } // Erstelle_BoundingBox
/*-------------------------------------------------*/
```

Lächerlich dass man uns mit so etwas überhaupt belästigt. Im letzten Kapitel dieses Tutorials haben wir ja schon eine Funktion für Bounding Boxen gesehen. Diese arbeitete allerdings mit Vertexlisten. Hier haben wir aber eine Liste von Polygonen eines Nodes. Also brauchen wir einfach noch eine Schleife über alle Polygone der Liste und darin die Schleife über alle Vertices eines Polygons. Der Rest der Funktion ist identisch. Also kümmern wir uns nun um das Klassifizieren.

Unser Job ist es eigentlich eine Funktion zu schreiben die ein Polygon in Relation zu einer Ebene klassifiziert. Dieses Problem, ebenso wie einige andere die später noch folgen, lassen sich auf das Problem zur Klassifizierung eines Punktes in Relation zu der Ebene zurückführen, also schreiben wir uns zunächst so eine Funktion:

```
int XBSP::Klassifiziere_Punkt(XEBENE xEbene,
                              D3DVECTOR vPunkt) {
   float fErgbn;
   D3DVECTOR vAuf_Ebene, vRichtung;

   // Wir brauchen einen Punkt auf der Ebene. Nimm den normalisierten
   // Normalenvektor mal die Entfernung der Ebene zum Ursprung:
   vAuf_Ebene.x = xEbene.vNormal.x * xEbene.fDistanz;
   vAuf_Ebene.y = xEbene.vNormal.y * xEbene.fDistanz;
   vAuf_Ebene.z = xEbene.vNormal.z * xEbene.fDistanz;
```

```
   // Endpunkt-Startpunkt=Richtungsvektor zw. beiden
   vRichtung.x = vAuf_Ebene.x - vPunkt.x;
   vRichtung.y = vAuf_Ebene.y - vPunkt.y;
   vRichtung.z = vAuf_Ebene.z - vPunkt.z;

   // Punktprodukt
   fErgbn = vRichtung.x * xEbene.vNormal.x
          + vRichtung.y * xEbene.vNormal.y
          + vRichtung.z * xEbene.vNormal.z;

   if (fErgbn < -BSP_DELTA)
      return BSP_FRONT;
   if (fErgbn > BSP_DELTA)
      return BSP_BACK;
   return BSP_PLANAR;
   } // Klassifiziere_Punkt
/*---------------------------------------*/
```

Piece of cake, ist alles eine Frage der Kopfarbeit denn der grosse Vorteil bei Vektorrechnung ist, dass man sich alles im Kopf aufmalen kann. Zuerst brauchen wir einen Punkt der auf der Ebene liegt. Einen solchen finden wir ganz einfach. Wir nehmen den Normalenvektor der Ebene und verlängern seine Länge auf die Entfernung die die Ebene vom Ursprung hat. Damit haben wir einen Vektor der vom Ursprung genau bis auf die Ebene (*und damit auf einen Punkt auf dieser*) zeigt.
Als nächstes berechnen wir einen Richtungsvektor von dem Punkt dessen Relation zur Ebene wir prüfen wollen zu dem Punkt der auf der Ebene liegt. Wir haben also einen Vektor zwischen diesen beiden Punkten. Die gedankliche Höchstleistung kommt aber nun. Wir setzen diesen Vektor in die Ebenegleichung ein und multiplizieren ihn mit dem Normalenvektor der Ebene, daraus erhalten wir den Wert von -d also die Entfernung. Dadurch dass wir aber nicht den Vektor vom Urpsrung zu dem Testpunkt verwenden, sondern den Vektor von der Ebene zum Testpunkt erhalten wir als Ergebnis den Wert 0 falls der Testpunkt in der Ebene liegt.
Ist das Ergebnis kleiner als 0 so liegt der Punkt vor der Ebene und ist das Ergbenis grösser als 0 so liegt der Testpunkt hinter der Ebene. Das klingt zwar erst mal etwas verwirrend, ist aber korrekt. Wir können hier nicht den Vektor vom Ursprung zum Testpunkt verwenden weil wir dann die Ausrichtung des Normalenvektors der Ebene auch noch berücksichtigen müssten, da die Ebene zum Ursprung hin oder von ihm weg blicken kann. Durch unsere Methode erhalten wir gleich das richtige Ergebnis. Aufgrund von Rundungsfehlern können wir aber nie genau auf 0.000000 rechnen, daher verwenden wir in obiger Formel die Konstante `BSP_DELTA` um ein wenig Ungenauigkeit zuzulassen.

Bewaffnet mit dieser Möglichkeit einen beliebigen Punkt im Raum zu einer gegebenen Ebene zu klassifizieren können wir ganz fix eine Funktion erschaffen die ein Polygon zu einer Ebene klassifiziert. Ein Polygon ist in diesem Sinne schliesslich nicht mehr als eine Ansammlung von Punkten:

```
int XBSP::Klassifiziere_Poly(XEBENE xEbene, XPOLY *pPoly) {
   D3DVECTOR vPunkt;
   int nAnz_Front=0, nAnz_Back=0, nAnz_Auf=0;
   int nKlasse;

   // Durchlaufe Punkte des Polys und klassifiziere sie
   for (int i=0; i < pPoly->lAnz_Verts; i++) {
      // Mache einen Vektor aus dem Punkt
      vPunkt.x = pPoly->vVerts[i].x;
      vPunkt.y = pPoly->vVerts[i].y;
      vPunkt.z = pPoly->vVerts[i].z;
      // Klassifizieren den Vektor zum Punkt
      nKlasse = this->Klassifiziere_Punkt(xEbene, vPunkt);
```

```
      // Vor, hinter oder auf der Ebene
      if (nKlasse == BSP_FRONT)
         nAnz_Front++;
      else if (nKlasse == BSP_BACK)
         nAnz_Back++;
      else {
         nAnz_Front++;
         nAnz_Back++;
         nAnz_Auf++;
         }
      } // for [alle Verts]

   // Alle Verts des Polys sind planar
   if (nAnz_Auf == pPoly->lAnz_Verts)
      return BSP_PLANAR;
   // Alle Verts des Polys liegen vor der Ebene
   else if (nAnz_Front == pPoly->lAnz_Verts)
      return BSP_FRONT;
   // Alle Verts des Polys liegen hinter der Ebene
   else if (nAnz_Back == pPoly->lAnz_Verts)
      return BSP_BACK;
   // Sowohl als auch => Poly spannt über die Ebene
   else
      return BSP_SPANNING;
   } // Klassifiziere_Poly
/*----------------------------------------*/
```

Hier sehen wir mal wieder eines ganz klar. Auch 3D Programmierer benutzen Wasser zum Kochen. Wir durchlaufen einfach alle Punkte des Polygons und klassifizieren sie. Dabei zählen wir mit wie viele Punkte als Front, Back und Planar klassifiziert wurden. Nach der Schleife müssen wir dann nur noch prüfen ob **alle** Punkte Front, oder **alle** Punkte Back oder **alle** Punkte planar waren. Trifft keiner der drei Fälle zu dann haben wir ein Poygon welches sich nicht eindeutig auf eine Seite oder in die Ebene zwängen lässt und dieses schneidet dann logischerweise die Ebene. Pfannkuchen.

Wahl der Waffen

Glaubt es oder nicht, unser BSP Baum ist so gut wie fertig. Ein Blick zurück auf die Funktion `XBSP::Erstelle_BSP_Baum` sagt uns, dass uns nur noch zwei weitere Hilfsfunktionen fehlen die wir dort benötig haben. Hier werden wir eine davon besprechen, nämlich die Auswahl des besten Splitters mit dessen Hilfe wir die Menge der Polygone eines Baumnodes teilen werden.

Glücklicherweise haben wir unse Theoriestunden ja bereits hinter uns. Dort hatten wir gesehen dass wir hauptsächlich zwei Schleifen brauchen. Eine äussere Schleife läuft über alle Polygone in der Liyte des zu prüfenden Nodes und wir wählen je Schleifendurchlauf ein anderes Polygon aus der Liste welches wir als Splitter testen. Für diesen Test brauchen wir die innere Schleife die wiederum über alle Polygone des Nodes läuft. Für jeden potentiellen Splitter klassifizieren wir alle Polygone der Liste und zählen wieder mit wie viele Polygone auf der Front- oder Backseite des Splitters liegen und wie viele Polygonsplits dieser potentielle Splitter verursachen würde. Abschliessend erstellen wir den Punktewert für jeden potentiellen Splitter nach folgender Formel:

```
   ulScore = abs(lFront - lBack) + (lSplits * 3);
```

Das haben wir ja oben auch schon besprochen. Die Anzahl der Frontpolygone minus die Anzahl der Backpolygone sorgt für einen ausgewogenen Baum. Jeden Polygonsplit bestrafen wir hier durch den Faktor 3. Je kleiner der Punktewert desto geeigneter ist ein potentieller Splitter, der Idealwert ist 0 denn das bedeutet 0 Splits und die gleiche Anzahl Polygone in Front und Back. Wir speichern dann immer den Splitter mit dem

kleinsten Punktwert als bisher besten Splitter und wenn wir mit der Funktion durch sind müssen wir nur noch eines prüfen: Haben wir überhaupt einen Splitter gefunden?
Das ist auch ganz einfach, denn ein potentieller Splitter ist auch wirklich nur ein Splitter wenn die Menge der Polygone nicht konvex ist. Das bekommen wir anhand folgender zwei Kriterien heraus. Nach einem Split muss entweder die Front- und die Backliste des Splitters mehr als 0 Elemente enthalten oder der Splitter muss mindestens einen Polygonsplit verursachen der dann die Front- und Backliste entsprechend füllen wird. So, dann schreiben wir schnell die Funktion:

```
BOOL XBSP::Finde_besten_Splitter(XPOLY *Polyliste, LONG lAnz_Polys,
                                 XEBENE *BestSplitter) {
   XPOLY *pBest_Splitter = NULL;
   XPOLY *pSplitter = NULL;
   XPOLY *pAkt_Poly = NULL;
   LONG   lFront  = 0, // Wie viele Polys liegen
          lBack   = 0, // bei jedem potentiellen
          lPlanar = 0, // Splitter in Front, Back
          lSplits = 0; // Planar oder spannen?
   int    nKlasse;
   ULONG  ulScore,
          ulBest_Score = 1000000;
   BOOL   blnSplitter_gefunden = FALSE; // mind. einer?

   // Durchlaufe alle Poly in der Liste
   for (int i=0; i<lAnz_Polys; i++) {
      // Nächstes Polygon als Splitter testen
      pSplitter = &Polyliste[i];

      // Zähler zurücksetzen
      lFront = lBack = lPlanar = lSplits = 0;

      // Jedes Poly max einmal als Splitter zulassen
      if (pSplitter->blnWar_schon_Splitter)
         continue;

      // Zurück auf den Start der Polygonliste
      pAkt_Poly = Polyliste;
      // Durchlaufe wieder alle Polys für diesen Splitter
      for (int j=0; j<lAnz_Polys; j++, pAkt_Poly++) {
         // Den Splitter nicht mit sich selbst testen
         if (i==j)
            continue;
         // Klassif. akt. Poly j für akt. Splitter i
         nKlasse = this->Klassifiziere_Poly(pSplitter->xEbene,
                                             pAkt_Poly);
         // Zähle wie viele Polys für Splitter i in die
         // jeweiligen Listen fallen
         if (     nKlasse == BSP_FRONT)
            lFront++;
         else if (nKlasse == BSP_BACK)
            lBack++;
         else if (nKlasse == BSP_PLANAR)
            lPlanar++;
         else
            lSplits++;
         } // for [innen]
```

```
      ulScore = abs(lFront-lBack) + (lSplits*3);

      // Haben wir einen neuen besten Score?
      if (ulScore < ulBest_Score) {
         // Ist die Polyliste nicht konvex?
         if ( ((lFront > 0) && (lBack > 0)) ||
              (lSplits > 0) ) {
            ulBest_Score   = ulScore;
            pBest_Splitter = pSplitter;
            // Okay wir haben mind. einen!
            blnSplitter_gefunden = TRUE;
            }
         } // if [ulScore]
      } // for [aussen]

   // Falls KEIN Splitter mehr gefunden abbrechen
   if (!blnSplitter_gefunden)
      return FALSE;

   // Splitter Polygon als benutzt markieren
   pBest_Splitter->blnWar_schon_Splitter = TRUE;

   // Splitter Ebene speichern
   (*BestSplitter) = pBest_Splitter->xEbene;

   return TRUE;
   } // Finde_besten_Splitter
/*-------------------------------------------*/
```

Wir dürfen nur nicht vergessen auch das entsprechende Feld `blnWar_schon_Splitter` eines Polygones zu setzen wenn dessen Ebene als Splitter verwendet wurde. Weiter oben hatten wir ja besprochen dass es keinen Sinn macht ein Polygon mehr als einmal als Teilungspolygon heranzuziehen.

Einsatz der Special Forces

Nun ist es so weit, die schweren Geschützen haben ihren Job getan und es ist an der Zeit die Spezialeinheiten aufzutauen. Jetzt geht es um die einzige Funktion bei der ganzen BSP Baum Geschichte von der selbst ich zugeben muss dass sie nicht ganz trivial ist. Es geht hier um das Splitten eines Polygons das eine Ebene schneidet in zwei Teile die dann jeweils komplett und ausschliesslich auf je einer Seite der Ebene liegen.

Für diesen Job brauchen wir eine kleine Hilfsfunktion die dazu dient den Schnittpunkt zwischen einer Ebene und einer Linie zu finden falls dieser existiert. Wofür brauchen wir das? Nun, eigentlich ist das Teilen eines Polygons gar nicht so schlimm. Wir nehmen das Polygon und selektieren immer zwei aufeinanderfolgende Punkte von diesem. Damit haben wir eine Linie zwischen zwei Vertices welche auch gleichzeitig eine Kante des Polygons ist. Dann prüfen wir ob dieses Linienstück die Ebene schneidet. Falls neien dann liegt diese Kante des Polygons ausschliesslich auf einer Seite des Polygons und beide Vertices kommen in das entsprechende der beiden neue Teilpolygon (*je eines vor bzw. hinter der Ebene*) die aus dem Schnitt resultieren. Schauen wir einmal auf die **Abbildung 7**.

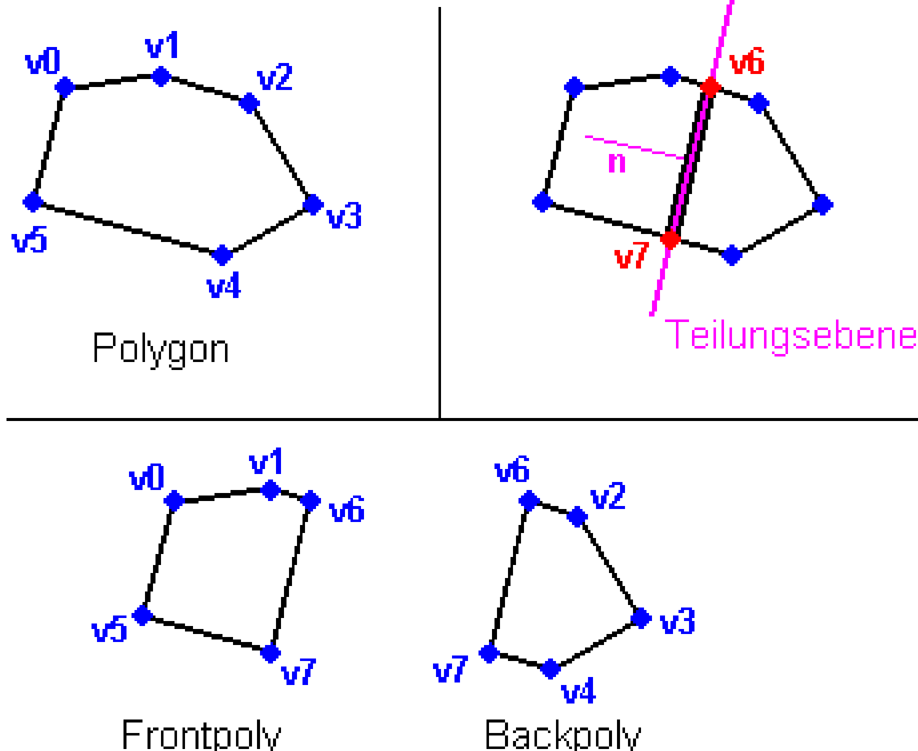


Abbildung 7: Teilung eines Polygons

Ganz oben links sehen wir das zu teilende Polygon aus sechs Vertices. Rechts daneben sehen wir auch schon was passieren muss. Wir haben die Teilungsebene und deren Normalenvektor eingezeichnet. Nun nehmen wir die Vertices v0 und v1 und prüfen diese Kante mit der Ebene: Es gibt keinen Schnittpunkt und beide Vertices liegen auf der Frontseite der Ebene. Dann gehen wir weiter zu v1 und v2 und prüfen diese Kante. Hier gibt es einen Schnitt denn v1 liegt auf der Frontseite und v2 liegt auf der Backseite der Ebene. Die beiden Vertices werden also dann den entsprechenden Polygonteilen zugeordnet, aber nun fehlt unseren beiden neuen Polygonen etwas. Wir müssen den Schnittpunkt der Kante mit der Ebene genau berechnen und diesen als neuen Vertex (*hier als v6 rot markiert*) in beide Teilpolygone Front und Back einsortieren. Dasselbe passiert dann für die Kante v4v5 in der der neue Vertex v7 entsteht.
Im unteren Teil der Abbildung sehen wir dann die beiden neuen Polygone die nun keinen Schnittpunkt mit der Teilungsebene mehr aufweisen. Das ursprüngliche Polygon ist damit vollkommen verschwunden und wir fügen an dessen Stelle diese beiden neuen Polygone in unsere Polygonlisten ein.

Okay, bevor wir also ein Polygon splitten können brauchen wir die Hilfsfunktion die uns den Schnittpunkt eines Linienstücks zwischen zwei Vertices mit einer Ebene liefert. Und damit beginnen wir nun. Dazu tauchen wir nun erstmalig etwas tiefer in die Gefilde der 3D Mathematik ab, den folgenden Abschnitt könnte man auch mit "*Mein Freund das Punktprodukt*" betiteln, oder besser "*Warum ich das Punktprodukt hassen und lieben gelernt habe*."

Was will uns das Punktprodukt V1*V2 zwischen zwei Vektoren sagen? Dazu haben wir ja bereits ein paar Dinge gehört, aber werfen wir mal einen Blick auf **Abbildung 8** um uns das noch einmal vor Augen zu führen. Und ab jetzt heisst es erst mal *schlucken*, *schlucken*, *schlucken*...was wir damit anfangen können das sehen wir schon noch früh genug.

$$\cos(\alpha) = \frac{V1 \cdot V2}{|V1| \cdot |V2|} \Rightarrow$$

Wenn V1,V2 Einheitsvektoren dann |V1| und |V2| = 1

$$\Rightarrow \cos(\alpha) = V1 \cdot V2$$

V1

V2

Abbildung 8: Das Punktprodukt

Ganz allgemein beschreibt das Punktprodukt zwischen V1 und V2 den Cosinus des Winkels Alpha zwischen den beiden Multipliziert mit den Beträgen der Vektoren. Der Betrag eines Vektors ist nichts anderes als die Länge des Vektors und man schreibt dies in der Mathematik mit den vertikalen Balken rechts und links vom Vektor. |V1| bedeutet also den Betrag/die Länge von V1. In der obigen Abbildung sieht man die Formel etwas umgeform, nämlich durch die beiden Beträge der Vektoren geteilt.
Die Formel für das Punktprodukt wird jedoch wesentlich einfacher wenn wir davon ausgehen, dass die beiden Vektoren V1 und V2 Einheitsvektoren sind. Wieder so ein Fremdwort. Das bedeutet einfach, dass die beiden Vektoren normalisiert sind und damit die Länge 1 haben. Durch 1*1 brauchen wir aber nicht zu teilen und daher fällt dieser Term aus der Formel weg. Das Punktprodukt zwischen zwei Einheitsvektoren liefert also direkt den Cosinus des Winkels zwischen den beiden.

Schlucken, schlucken, schlucken...und weiter gehts. Die **Abbildung 9** bringt uns wieder auf Kurs Richtung Schnittpunkt einer Linie mit einer Ebene. Hier sehen wir wie wir die Formel des Punktproduktes geometrisch interpretieren können wenn **einer** der beiden Vektoren kein Einheitsvektor ist.

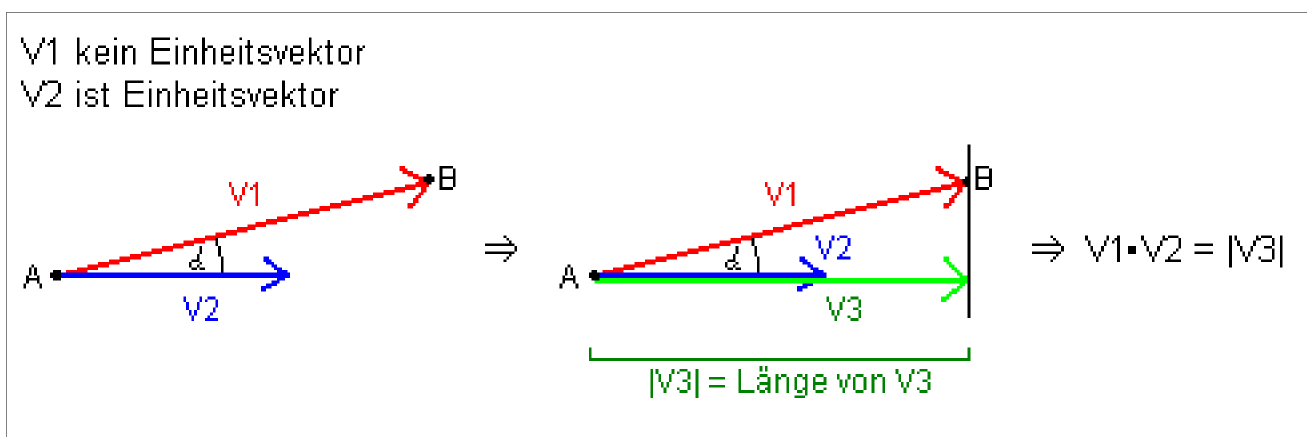


Abbildung 9: Das Punktprodukt II

V1 ist hier kein Einheitsvektor während V2 sehr wohl ein Einheitsvektor ist. In einem solchen Fall können wir das Ergebnis des Punktproduktes zwischen V1 und V2 so interpretieren, dass das Ergbenis des Punktproduktes |V3| (*der Länge des Vektors V3*) entspricht. Dabei ist V3 eine Projektion des Vektors V1 auf den Einheitsvektor V2. *Schlucken, schlucken, schlucken*...
Nehmen wir einmal an der Punkt A sei die Position der Kamera und Punkt B sei ein Punkt in einer Ebene. Damit wäre der Vektor V1 ein Vektor von der Kameraposition zu einem Punkt in einer Ebene. Cool. Stellen wir uns weiter vor dass der Einheitsvektor V2 der Normalenvektor der Ebene ist...was lernen wir daraus? Wir haben nun eine Möglichkeit gefunden die Entfernung der Kamera (*oder eines beliebigen Punktes*) zu einer Ebene zu berechnen. Bisher konnten wir immer nur die Entfernung der Ebene vom Ursprung berechnen. Hui...noch cooler. Aber jetzt wird es brutal, also Mund...äh Augen auf.

Jetzt können wir uns daran machen eine Methode zu entwickeln die uns den Schnittpunkt einer Polygonkante mit einer Ebene ermittelt. Definieren wir also folgendes. Punkt A ist der Startpunkt der Polygonkante (*z.B. V0 in der Polygonliste*), Punkt B ist ein Punkt auf der Ebene an der das Polygon geteilt werden soll und Punkt C ist der Endpunkt der Polygonkante (*z.B. V1 in der Polygonliste*). Mit Papier und Bleistift kommen wir schnell zu

folgender Skizze in der **Abbildung 10** die unser Teilungsproblem löst.

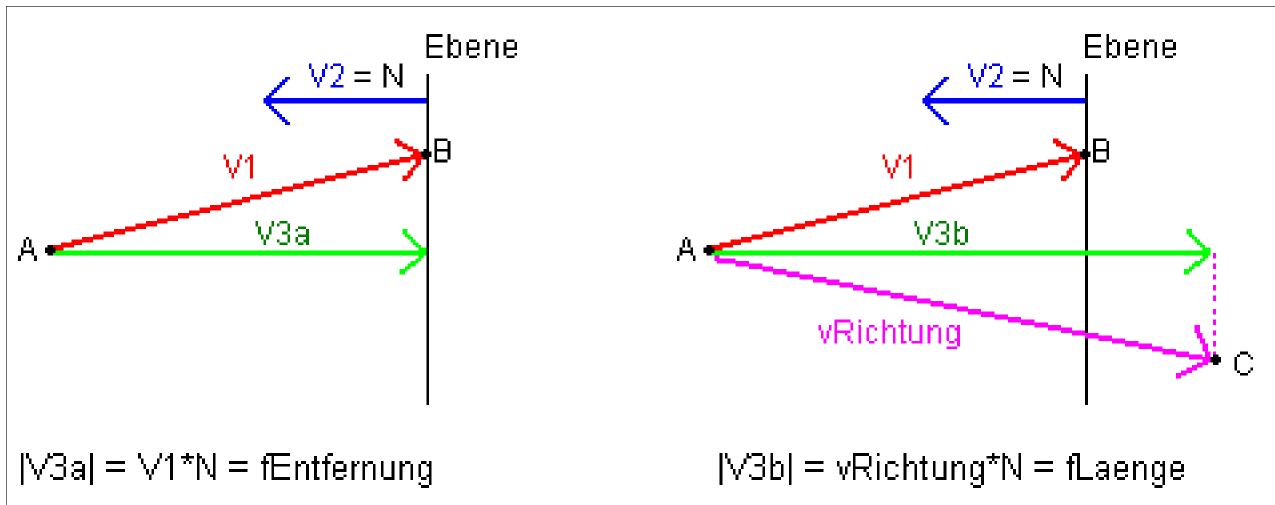


Abbildung 10: Punktprodukte

Nun denn meine jungen Jedis. Der linke Teil der Abbildung wiederholt nur noch mal ein wenig. Wir haben eine Ebene mit dem Nromalenvektor N. Erstellen wir das Punktprodukt aus diesem Einheitsvektor N und dem Vektor V1 vom Punkt A zu einem Punkt B auf der Ebene dann erhalten wir als Ergebnis die Entfernung des Punktes A von der Ebene, also mathematisch Formuliert |V3a|.
Auf dem rechten Teil der Abbildung kommen wir dann zur Sache. Wir ziehen nun einen Vektor vRichtung von Punkt A zu Punkt C und nehmen das Punktprodukt von diesem Vektor mit dem Normalenvektor N der Ebene. Das Ergebnis dieser Berechnung ist sozusagen die Entfernung des Punktes A von der Ebene falls diese Ebene parallel verschoben wäre so dass Punkt C in der Ebene liegen würde. In Ermangelung eines besseren Namens haben ich diesem Wert fLaenge genannt. *Schlucken, schlucken, schlucken*...

Und nun bitte Nachdenken. Wir suchen den Schnittpunkt des Vektors vRichtung zwischen A und C mit der Ebene. Aus der Zeichnung können wir aber eines Ersehen. Den Schnittpunkt des Strahls V3 mit der Ebene könnten wir schon berechnen, aber wir haben hier noch einen interessanten Zusammenhang. Der Prozentsatz bei dem der Vektor V3b von der Ebene geschnitten wird entspricht nämlich genau dem Prozentsatz bei dem auch der Vektor vRichtung von von der Ebene geschnitten wird. In anderen Worten, wenn V3b nach drei Vierteln auf die Ebene trifft, dann trifft auch vRichtung nach drei Vierteln auf die Ebene.
Wir berechnen also den Prozentsatz an dem der Schnitt auftritt mit `|V3a|/|V3b| = fEntfernung/fLaenge`. Nun müssen wir nur noch den Vektor vRichtung mit diesem Prozentwert multiplizieren und zu dem Vektor zum Punkt A addieren, dann haben wir den Schnittpunkt von vRichtung mit der Ebene genau bestimmt. Und den Prozentwert an dem dieser Schnitt auftritt, denn können wir später auch noch mal verwenden. Wir entwickeln daher jetzt eine Funktion die die Punkte A und C als Start- bzw. Endpunkt einer Linie aus zwei aufeinanderfolgenden Vertices eines Polygons aufnimmt, ebenso wie den Normalenvektor einer Ebene und einen Punkt auf dieser Ebene. Daraus berechnet die Funktion den Schnittpunkt und den Prozentsatz und gibt diese Werte als Call-by-Referenz zurück.

```
BOOL xUtil_Schnittpunkt(D3DVECTOR *vLinienstart,
                        D3DVECTOR *vLinienende,
                        D3DVECTOR *vAuf_Ebene,
                        D3DVECTOR *vNormal,
                        D3DVECTOR *vSchnittpunkt,
                        float *fProzent) {
   D3DVECTOR vRichtung,// Vektor von vLinstart zu vLinEnde
             V1;       // Vektor von vLinstart zu vAufEbene
   float     fLaenge,
             fEntfernung;

   vRichtung.x = vLinienende->x - vLinienstart->x;
   vRichtung.y = vLinienende->y - vLinienstart->y;
```

```
   vRichtung.z = vLinienende->z - vLinienstart->z;

   // Punktprodukt: Wenn v1 NICHT normalisiert und v2 IST
   // normalisiert dann erhalten wir so die Entfernung des
   // Punktes v1 zu der Ebene mit Normalenvektor v2
   // HIER: Entfernung von vLinienstart zur Ebene wenn Ebene
   // so verschoben dass Linienende auf der Ebene liegt.
   fLaenge = vRichtung.x * (*vNormal).x
           + vRichtung.y * (*vNormal).y
           + vRichtung.z * (*vNormal).z;

   if (fabsf(fLaenge)<0.0001) {
      return FALSE;
      }

   V1.x = vAuf_Ebene->x - vLinienstart->x;
   V1.y = vAuf_Ebene->y - vLinienstart->y;
   V1.z = vAuf_Ebene->z - vLinienstart->z;

    //  Punktprodukt:  Wenn  v1  NICHT  normalisiert  und  v2  IST
    //  normalisiert  dann  erhalten  wir  so  die  Entfernung  des
    //  Punktes  v1  zu  der  Ebene  mit  Normalenvektor v2
   // HIER: Entfernung von vAuf_Ebene zu Linienstart
   fEntfernung = V1.x * (*vNormal).x
               + V1.y * (*vNormal).y
               + V1.z * (*vNormal).z;

   // Entfernung durch Laenge gibt an bei wieviel Prozent
   // der Linie der Schnittpunkt mit der Ebene liegt (0-1)
   *fProzent = fEntfernung / fLaenge;

   if (*fProzent<0.0f)
      return FALSE;
   else if (*fProzent>1.0f)
      return FALSE;
   else {
      vSchnittpunkt->x = vLinienstart->x + vRichtung.x*(*fProzent);
      vSchnittpunkt->y = vLinienstart->y + vRichtung.y*(*fProzent);
      vSchnittpunkt->z = vLinienstart->z + vRichtung.z*(*fProzent);
      return TRUE;
      }
   } // xUtil_Schnittpunkt
/*-------------------------------------------------------*/
```

You still with me? Wer bis hierhin durchgehalten hat der hat es eigentlich schon geschafft. Die eben entwickelte Funktion ist der schlimmste Brocken Mathematik in dem BSP Code und den haben wir jetzt in einer handlichen Funktion versteckt. Machen wir uns also daran den Code zu schreiben mit dessen Hilfe wir ein Polygon an einer Ebene splitten in zwei neue Polygone erzeugen können. Mit der nun geleisteten Vorarbeit ist das im Prinzip nur noch ein wenig Bastelarbeit. Der Code ist zwar extrem lang da es viel zu tun gibt, aber nicht sonderlich kompliziert. Los geht's, fangen wir mit den lokalen Variablen an (*diese Funktion basiert auf dem BSP II Tutorial von Gary Simmons, in Perez; Royer "Advanced 3D Game Programming..." [oder wie auch immer es jetzt heisst] findet sich ebenfalls entsprechender Code mit Erläuterung*):

```
void XBSP::Split_Poly(XPOLY *Poly, XEBENE *pEbene,
                      XPOLY *FrontSplit, XPOLY *BackSplit) {
   D3DLVERTEX vFrontList[20],  // Neues Polygon vor Ebene
              vBackList[20];   // Neues Polygon hinter der Ebene
```

```
   D3DVECTOR  vSchnittpunkt,   // Schnittpunkt mit der Ebene
              vAuf_Ebene,      // Beliebiger Punkt auf der Ebene
              vPunktA, vPunktB;// Zwei Folgepunkte im Polygon
   WORD       wAnz_Front=0,    // Anzahl Verts im Frontteil
              wAnz_Back=0,     // Anzahl Verts im Backteil
              wLoop=0,         // Schleifenzähler
              wAkt_Vertex=0;   // Aktueller Vertex
   float      fProzent;        // Prozent bis Schnittpkt
wird fortgesetzt...
```

Eigentlich dürfte hier alles klar sein. Wir brauchen zwei Listen mit Vertices, je eine für eines der beiden neuen Polygone auf der Front- und der Backseite der Ebene. Dann benötigen wir einen Vektor zum Schnittpunkt einer Linie zwischen zwei Vertices des Polys mit der Ebene (*siehe allgemeine Splitting-Erklärung bei **Abbildung 7** oben*). Dazu haben wir zwei Vektoren auf zwei aufeinanderfolgende Vertices im Polygon zwischen denen wir eine Linie aufmachen und einen Vektor zu einem Punkt auf der Ebene, letzteren brauchen wir ja für die eigentliche Schnittpunktfunktion welche wir bereits entwickelt haben. Dann haben wir noch ein paar Zähler und den Prozentwert wo auf der Linie zwischen zwei Vertices des Polys der Schnittpunkt zur Ebene liegt. Diesen Wert gibt uns die Schnittpunktfunktion zurück und wir brauchen ihn zur Berechnung der Texturkoordinaten an neu erzeugten Vertices bei den Schnittpunktkoordinaten.

Der erste Schritt dieser Funktion ist es nun, einen Vektor zu einem Punkt auf der Ebene zu finden. Das klingt vielleicht ungeheuer kompliziert, ist es aber nicht. Schliesslich gibt es unendlich viele davon, das Problem ist nur einen zu finden. Das ist aber lächerlich einfach. Wir nehmen einfach den Normalenvektor der Ebene und laufen diese Richtung vom Ursprung aus so lange ab bis wir die Entfernung der Ebene zum Ursprung gegangen sind. Et voilá, wir sitzen genau auf der Ebene! Diesen Vektor merken wir uns für später und beginnen dann bereits damit, die beiden neuen Polygone zu konstruieren. Wir nehmen den ersten Vertex des Polygons und klassifizieren ihn gegen die Ebene. Entsprechend dieser Klassifizierung sortieren wir den ersten Vertex richtig ein, Front, Back oder in beide Polygone wenn der Vertex planar ist, denn dann muss ein Teil beider neuen Polygone sein weil er genau auf der Ebene liegt.

```
Split_Poly() Fortsetzung:
   // Wir brauchen einen Punkt auf der Ebene. Nimm normalisierten
   // Normalenvektor mal Entfernung der Ebene zum Ursprung:
   vAuf_Ebene.x = pEbene->vNormal.x * pEbene->fDistanz;
   vAuf_Ebene.y = pEbene->vNormal.y * pEbene->fDistanz;
   vAuf_Ebene.z = pEbene->vNormal.z * pEbene->fDistanz;

   // Klassifiziere den ersten Vertex des Polygons und
   // sortiere ihn in die entsprechende Liste
   D3DVECTOR v1_Vector;
   v1_Vector.x = Poly->vVerts[0].x;
   v1_Vector.y = Poly->vVerts[0].y;
   v1_Vector.z = Poly->vVerts[0].z;

   switch (this->Klassifiziere_Punkt(*pEbene, v1_Vector))
      {
      case BSP_FRONT:
         vFrontList[wAnz_Front++] = Poly->vVerts[0];
         break;
      case BSP_BACK:
         vBackList[wAnz_Back++] = Poly->vVerts[0];
         break;
      case BSP_PLANAR:
         vBackList[wAnz_Back++]   = Poly->vVerts[0];
         vFrontList[wAnz_Front++] = Poly->vVerts[0];
         break;
      default:
         PostQuitMessage(0);
      } // switch
wird fortgesetzt...
```

Und nun wird es mal wieder haarig denn jetzt kommt ein grosser Code Happen der nur schwer zu trennen ist und eventuell etwas unüberschaubar wirkt. Eigentlich ist das aber gar nicht so schwer. Überlegen wir mal allgemein was wir jetzt machen müssen.

Wir benötigen nun eine Schleife über alle Vertices des Polys beginnend ab dem zweiten Vertex da wir den ersten ja bereits abgehandelt haben. Den jeweils aktuellen Vertex klassifizieren wir dann gegen die Ebene. Ist dieser Vertex planar so kommt er in beide neuen Polygone, so wie im entsprechenden Fall des ersten Vertex. Dann ist die Schleife schon zu Ende. Ist dieser Vertex jedoch nicht planar so wird es komplizierter. Dann nehmen wir den aktuellen Vertex den wir als B bezeichnen und den vorhergehenden Vertex im Polygon den wir als A bezeichnen. Damit haben wir eine Polygonkante zwischen den Punkten A und B und was wir damit machen sehen wir uns gleich an, zuerst aber das neue Code Häppchen. Es sei angemerkt dass der `else` Fall hier wegen der Übersichtlichkeit noch nicht implementiert ist. Genau dort gehört die Behandlung der Linie AB hin.

Split_Poly() Fortsetzung:

```
   // Schleife über alle Vertices im Poly
   for (wLoop=1; wLoop < Poly->lAnz_Verts+1; wLoop++) {
      if (wLoop == Poly->lAnz_Verts)
         wAkt_Vertex = 0;
      else
         wAkt_Vertex = wLoop;

      // Nimm zwei aufeinanderfolgende Vertices des Polys
      vPunktA.x = Poly->vVerts[wLoop-1].x;
      vPunktA.y = Poly->vVerts[wLoop-1].y;
      vPunktA.z = Poly->vVerts[wLoop-1].z;

      vPunktB.x = Poly->vVerts[wAkt_Vertex].x;
      vPunktB.y = Poly->vVerts[wAkt_Vertex].y;
      vPunktB.z = Poly->vVerts[wAkt_Vertex].z;

      // Klassifiziere den Punkt in Bezug zur Ebene
      int nErgbn = this->Klassifiziere_Punkt(*pEbene, vPunktB);

      // Falls Vertex direkt in der Ebene dann kommt eine
      // Kopie in beide neuen Polygone
      if (nErgbn == BSP_PLANAR) {
         vBackList[wAnz_Back++]   = Poly->vVerts[wAkt_Vertex];
         vFrontList[wAnz_Front++] = Poly->vVerts[wAkt_Vertex];
         }
      // Sonst kontrollieren wir ob die beiden zuletzt betrachteten
      // Punkte eine Linie bilden die die Ebene schneidet. Dann
      // müssen wir einen neuen Punkt erzeugen
      else {

         } // else [!BSP_PLANAR]
      } // for [AnzVerts]
```
wird fortgesetzt...

Wow, immer noch easy...und es wird auch nicht komplizierter. Denn was müssen wir im `else` Fall machen? Wir müssen lediglich prüfen ob es einen Schnittpunkt der Line AB mit der Ebene gibt. Ist dies **nicht** der Fall, so liegen die beiden Punkte A und B auf derselben Seite der Ebene und wir können Punkt B ganz normal klassifizieren und in das richtige Polygon einsortieren.
Im Horrorszenario dass es einen Schnittpunkt gibt haben wir die Situation, dass A und B auf verschiedenen Seiten der Ebene liegen. Wir oben allgemein besprochen müssen wir dann die genauen Koordinaten des Schnittpunktes berechnen und diesen Schnittpunkt in beide neuen Polygone als zusätzlichen Punkt einsortieren. Doch dafür haben wir bereits unsere Schnittpunktfunktion, ist also doch gar nicht so kompliziert. Hier also der Code für den eben ausgelassenen `else` Fall:

else Fall Fortsetzung:

```
         // Gibt es einen Schnittpunkt?
```

```
if (xUtil_Schnittpunkt(&vPunktA, // Linienstart
                       &vPunktB, // Linienende
                       &vAuf_Ebene,
                       &pEbene->vNormal,
                       &vSchnittpunkt,
                       &fProzent) == TRUE)
   {
   // Erstelle neuen Vertex mit Texturkoordinaten
   float fDeltaX, fDeltaY, fTexX, fTexY;

   fDeltaX = Poly->vVerts[wAkt_Vertex].tu -
             Poly->vVerts[wLoop-1].tu;
   fDeltaY = Poly->vVerts[wAkt_Vertex].tv -
             Poly->vVerts[wLoop-1].tv;
   fTexX = Poly->vVerts[wLoop-1].tu + (fDeltaX * fProzent);
   fTexY = Poly->vVerts[wLoop-1].tv + (fDeltaY * fProzent);

   // Erstelle einen Vertex aus dem Vektor
   D3DLVERTEX vKopie;
   vKopie.x     = vSchnittpunkt.x;
   vKopie.y     = vSchnittpunkt.y;
   vKopie.z     = vSchnittpunkt.z;
   vKopie.color = Poly->vVerts[0].color;
   vKopie.tu    = fTexX;
   vKopie.tv    = fTexY;

   // Neuen Punkt in Front- und Backliste
   vBackList[wAnz_Back++]   = vKopie;
   vFrontList[wAnz_Front++] = vKopie;

   // Aktuellen Punkt auch noch einsortieren
   if (nErgbn == BSP_FRONT) {
      if (wAkt_Vertex != 0) {
         vFrontList[wAnz_Front++] =
             Poly->vVerts[wAkt_Vertex];
         }
      } // if [FRONT]
   else if (nErgbn == BSP_BACK) {
      if (wAkt_Vertex != 0) {
         vBackList[wAnz_Back++] =
             Poly->vVerts[wAkt_Vertex];
         }
      } // else [BACK]
   } // if [Schnittpunkt]

// Falls kein Schnittpunkt können wir den Punkt
// gefahrlos in die passende Liste einsortieren
else {
   if (nErgbn == BSP_FRONT) {
      if (wAkt_Vertex!=0) {
         vFrontList[wAnz_Front++] =
             Poly->vVerts[wAkt_Vertex];
         }
      }
   else if (nErgbn == BSP_BACK) {
      if (wAkt_Vertex!=0) {
```

```
                  vBackList[wAnz_Back++] =
                      Poly->vVerts[wAkt_Vertex];
                  }
               }
            } // else [kein Schnittpunkt]
```
else Fall komplett

Im Falle dass wir einen neuen Vertex erzeugen dürfen wir natürlich nicht vergessen auch den aktuellen Vertex, also Punkt B, zu klassifizieren und einzusortieren. Viel mehr gibt es hier nicht mehr anzumerken.
So, damit haben wir dann alle Vertices des Polygons durchlaufen und in die entsprechenden Listen einsortiert. Der Rest ist dann wieder reine Pfannkuchenarbeit, denn wir müssen aus den beiden Listen nur noch die beiden Polys erstellen. Wir beginnen diese Arbeit damit, die entsprechenden Attribute des nun gesplitteten Originalpolygons in die beiden neuen Polygone zu kopieren. Insbesondere die verwendete Textur und die Information darüber ob dieses Polygon schon mal ein Splitter war. In diesem Fall dürfen die beiden neuen Polygone auch keine Splitter mehr werden da das geometrisch auch nichts bringen würde.

Split_Poly() Fortsetzung:

```
   // Attribute des alten Polys übernehmen
   FrontSplit->lAnz_Verts =
      BackSplit->lAnz_Verts = 0;

   FrontSplit->nTextur_Index =
      BackSplit->nTextur_Index =
      Poly->nTextur_Index;

   FrontSplit->blnWar_schon_Splitter =
      BackSplit->blnWar_schon_Splitter =
      Poly->blnWar_schon_Splitter;

   for (wLoop=0; wLoop < wAnz_Front; wLoop++) {
      FrontSplit->lAnz_Verts++;
      FrontSplit->vVerts[wLoop] = vFrontList[wLoop];
      }
   for (wLoop=0; wLoop < wAnz_Back; wLoop++) {
      BackSplit->lAnz_Verts++;
      BackSplit->vVerts[wLoop] = vBackList[wLoop];
      }

   BackSplit->lAnz_Indices  = (BackSplit->lAnz_Verts -2)*3;
   FrontSplit->lAnz_Indices = (FrontSplit->lAnz_Verts-2)*3;
```
wird fortgesetzt...

Interessant ist hier noch die Berechnung der Anzahl der Indices der neuen Polygone durch `(Anz-2)*3`. Unser Problem hier ist doch, dass wir die Polygone als Indexlisten durch `DrawIndexedPrimitiveUP()` rendern wollen. Damit müssen wir unser Polygon in Indices auf Dreiecke zerlegen. Das hatten wir ja eigentlich schon so weit, denn in unserer Leveldatei finden sich diese Indices. ABER beim Splitten eines Polys haben wir natürlich immer mindestens zwei neue Vertices in das Originalpolygon eingefügt (*denn wenigstens ein Vertex des Polys liegt ja auf einer anderen Seite der Ebene und daher gibt es mindestens zwei Schnittpunkte mit der Ebene*). Unsere Originalindexlisten stimmen also nicht mehr und wir müssen aus der Liste von Vertices der neuen Polys eine neue Dreickeszerlegung in Indices vornehmen.
Glücklicherweise ist dies einfacher als es klingt...jedenfalls für **konvexe** Polygone. Merken wir uns also erst mal, dass wir in unserer Leveldatei auch nur konvexe Polygone haben sollten, denn sonst können wir beim Splitten dieser Polygone Probleme bekommen. Die Zerlegung ist denkbar einfach. Wir nehmen die ersten drei Vertices aus der Vertexliste des Polygons. Diese ergeben das erste Dreieck. Den ersten Vertex des Polygons behalten wir bei, den zuletzt verwendeten ebenfalls (*in diesem Fall der dritte des Polys*). Den zweiten Vertex des Dreiecks ersetzen wir durch den folgenden aus der Vertexliste. Das machen wir so lange bis wir keine Vertices in der Liste mehr haben. Der erste Vertex des Polys ist also immer der erste Vertex eines Dreiecks der Zerlegung.
Bei einem Poly mit fünf Vertices sieht die Dreieckszerlegung also wie folgt aus:

```
v0, v1, v2
v0, v2, v3
```

`v0, v3, v4`

Und tatsächlich haben wir `(Anz-2)*3 = (5-2)*3 = 3*3 = 9` Indices in der Liste, denn wir haben drei Dreicke mit je 3 Indices. Bewaffnet mit diesem Wissen können wir jetzt die letzten Zeilen des Codes zur Erstellung des BSP Baums schreiben:

Split_Poly() Fortsetzung:

```
   // Berechne die Dreieckszerlegung und speichere sie
   WORD v0,v1,v2;
   for (wLoop=0; wLoop<FrontSplit->lAnz_Indices/3; wLoop++)
      {
      if (wLoop==0) {
         v0=0;
         v1=1;
         v2=2;
         }
      else {
         v1=v2;
         v2++;
         }

      FrontSplit->wIndices[(wLoop*3)+0] = v0;
      FrontSplit->wIndices[(wLoop*3)+1] = v1;
      FrontSplit->wIndices[(wLoop*3)+2] = v2;
      } // for

   //  Backpoly
   for (wLoop=0; wLoop<BackSplit->lAnz_Indices/3; wLoop++)
      {
      if (wLoop==0) {
         v0=0;
         v1=1;
         v2=2;
         }
      else {
         v1=v2;
         v2++;
         }

      BackSplit->wIndices[(wLoop*3)+0] = v0;
      BackSplit->wIndices[(wLoop*3)+1] = v1;
      BackSplit->wIndices[(wLoop*3)+2] = v2;
      } // for

   FrontSplit->xEbene = this->Ebene_von_Poly(&(*FrontSplit));
   BackSplit->xEbene  = this->Ebene_von_Poly(&(*BackSplit));
   } // Split_Poly
/*----------------------------------------------------------*/
```

Strike!

Propaganda, Rendern des BSP für die Zuschauer am Bildschirm

Es ist vorbei...die Schlacht gegen die bösen Mächte der 3D Mathematik ist ein weiteres Mal erfolgreich von uns geschlagen worden. Nun bleibt uns nur noch eines, wir müssen unseren Sieg so richtig auskosten. Was haben

wir von dem besten BSP Baum der Welt wenn wir diesen niemandem zeigen können. Schreiben wir uns also fix ein paar Funktiönchen zum Rendern des Baumes. Beginnen wir mit einer allgemeinen Funktion die der Programmierer aufrufen muss und die dann den Rest erledigt:

```
void XBSP::Render(void) {
   D3DMATRIX matWelt;

   // Weltverschiebung auf 0 setzen
   xUtil_Einheitsmatrix(&matWelt);
   g_lpD3DDevice->SetTransform(D3DTS_WORLD, &matWelt);

   // Vertex Shader einstellen
   g_lpD3DDevice->SetVertexShader(D3DFVF_LVERTEX);

   // D3D Lichtengine ausschalten da Prelit-Vertices
   g_lpD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);

   // Den ganzen BSP Baum durchrendern
   this->Render_rekursiv(&this->xWurzel, xKam.vPos);

   // D3D Lichtengine wieder einschalten
   g_lpD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
   } // Render
/*----------------------------------------------*/
```

In dieser Mantelfunktion wecken wir das Direct3D Device auf und schieben dann die ganze Arbeit auf die `Render_rekursiv` Funktion unserer BSP Klasse weiter. Ebenso wie wir den Baum rekursiv erstellen können, können wir ihn natürlich auch rekursiv durchlaufen. Überlegen wir mal was wir eigentlich rendern müssen. Die gesamte Geometri steckt doch in den Leafs des Baumes. Wir müssen also alle Äste des Baumes ablaufen bis wir in einem Leaf landen und dann dessen Geometrie rendern. Dan tun wir das doch einfach:

```
void XBSP::Render_rekursiv(XNODE *pNode,
                           D3DVECTOR vPosition) {
   int nKlasse;

   // View Frustrum Culling des Nodes
   if (xUtil_Cull_AABB(&pNode->xBox) == BOX_AUSSERHALB)
      return;

   // Falls dieses Element ein Leaf ist dann rendere seine Polys
   if (pNode->blnLeaf) {
      this->Render_Leaf(pNode);
      return;
      }

   // Sonst wandere weiter, je nach Position des Spielers
   nKlasse = this->Klassifiziere_Punkt(pNode->xEbene, vPosition);

   if (nKlasse == BSP_BACK) {
      Render_rekursiv(pNode->pFront, vPosition);
      Render_rekursiv(pNode->pBack, vPosition);
      }
   else {
      Render_rekursiv(pNode->pBack, vPosition);
      Render_rekursiv(pNode->pFront, vPosition);
      }
```

```
   } // Render_rekursiv
/*----------------------------------------------*/
```

In dieser Funktion testen wir dann endlich auch die Bounding Box eines Nodes gegen den View Frustrum. Kommen wir an eine Stelle im Baum wo die Bounding Box ausserhalb des Frustrums liegt so können wir die Arbeit an diesem Ast abbrechen. Treffen wir auf ein Leaf so schieben wir die Arbeit wiederum an eine andere Funktion weiter. Und nun kommt wieder ein kleiner BSP Trick. Wir prüfen jetzt, wenn wir also einen Ast im View Frustrum haben, die Position der Kamera zu der Ebene des Nodes an dem wir uns befinden. Ist die Position der Kamera hinter der Ebene so rendern wir zuerst alle Kindernodes der Frontseite des aktuellen Nodes. *Häh?* Okay, das liegt daran dass alle Polygone auf der Frontseite dann weiter von der Kamera entfernt sind als diejenigen auf der Backseite. Und wir wollen die Polygone ja von hinten nach vorne rendern um beispielsweise Transparanzeffekte korrekt darstellen zu können. Umgekehrt rendern wir zuerst die Kindernodes der Backseite wenn die Kamera auf der Frontseite liegt.

Render, render, render. Nun kommt endlich die Funktion die tatsächlich etwas auf den Bildschirm malt:

```
void XBSP::Render_Leaf(XNODE *pNode) {
   XPOLY *pPoly = pNode->pPolys;

   // Durchlaufe alle Polys im Leaf und rendere sie
   for (int i=0; i<pNode->lAnz_Polys; i++, pPoly++) {
      // Textur des Polys einstellen
      #define n pPoly->nTextur_Index
      g_lpD3DDevice->SetTexture(0, this->aTextur[n].lpTextur);

      g_lpD3DDevice->BeginScene();
      g_lpD3DDevice->DrawIndexedPrimitiveUP(
               D3DPT_TRIANGLELIST,    // echte Dreiecke
               0,                     // Min Index
               pPoly->lAnz_Verts,     // Anz Vertices
               pPoly->lAnz_Indices/3, // Anz Primitive
               pPoly->wIndices,       // Indexliste
               D3DFMT_INDEX16,        // 16 Bit Indices
               pPoly->vVerts,         // Vertexliste
               sizeof(D3DLVERTEX));   // Vertexgrösse
               g_lpD3DDevice->EndScene();
      } // for
   } // Render_Leaf
/*----------------------------------------*/
```

Boah...langweilig oder? Wir müssen einfach nur alle Polygone in der Liste des Leafs durchlaufen. Für jedes Polygon setzen wir dann die entsprechende Textur und rendern es als indizierte Primitive. Bitte mal die Hand heben wer jetzt enttäuscht ist dass ein BSP Baum eigentlich so einfach ist **:-)**

Beseitigung der Trümmer

So, nachdem wir dann die Schlacht geschlagen haben müssen wir nachher auch die Trümmer aufräumen, sprich den allokierten Speicher wieder freigeben. Das ist hier natürlich denkbar einfach. Alle Texturen des BSP Baums müssen gekilled werden und alle Polygonlisten in den Leafs müssen ihres Speichers beraubt werden. Das machen wir mit den folgenden beiden Funktionen.

```
void XBSP::Kill(void) {
   // Texturen freigeben
   for (int i=0; i<this->Anz_Texturen; i++) {
      this->aTextur[i].lpTextur->Release();
      this->aTextur[i].lpTextur = NULL;
```

```
      } // for

   // Speicher der Polygonlisten freigeben
   this->Kill_rekursiv(&this->xWurzel);
   } // Kill
/*----------------------------------*/

void XBSP::Kill_rekursiv(XNODE *pNode) {
   // Falls Element ein Leaf Speicher freigebn
   if (pNode->blnLeaf) {
      free(pNode->pPolys);
      pNode->pPolys = NULL;
      }

   // Falls Kinder dann diese auch killen
   if (pNode->pBack)
      this->Kill_rekursiv(pNode->pBack);
   if (pNode->pFront)
      this->Kill_rekursiv(pNode->pFront);
   } // Kill_rekursiv
/*----------------------------------*/
```

Der BSP Baum in unserem Code

Let's rock! Jetzt haben wir den BSP Baum sauber gekapselt und können ihn wunderbar einfach in unserem Code verwenden. Dazu brauchen wir lediglich ein Objekt der Klasse zu deklarieren:

```
XBSP BSP_Baum;
```

Dieses werden wir dann in Phase 1, also bei der Initialisierung des Programms, mit den Daten füllen bzw. den BSP Baum berechnen lassen:

```
// Initialisiere den BSP Baum
blnErg = BSP_Baum.Init("testlevel.zfx");
if (blnErg == FALSE) {
   fprintf(Protokoll,"SpielInit: BSP failed \n");
   g_blnBeenden = TRUE;
   return FALSE;
   }
```

In der zweiten Phase, also dem Laufen des Programms, können wir den Baum dann ganz einfach in den BackBuffer rendern:

```
// Indoorlevel rendern
BSP_Baum.Render();
```

Zu guter Letzt müssen wir dann in der Phase 3, also dem Beenden des Programms, auch brav alles aufräumen und geordneten Speichern hinterlassen:

```
// BSP Baum Speicher freigeben
BSP_Baum.Kill();
```

Pfannkuchen.

Was kommt denn jetzt noch?

Keine Panik, jetzt haben wir wirklich alles komplett was wir an Quellcode brauchen und unser Demo ist damit wirklich fertig. Bevor ich Euch aber den Code downloaden lassen will ich noch einen Ausblick geben wie wir den Code verbessern können um die Technik des BSP Baumes noch schneller zu machen als sie jetzt schon ist.

Noch schneller rendern durch zwei BSP Bäume...

Eines der Probleme unseres aktuellen BSP Baums ist der Overdraw der Trotz des Bounding Box Tests auftritt. Befindet sich die Kamera etwa in einem sehr kleinen Raum und dahinter befindet sich ein sehr grosser konvexer Raum der ein einzelnes Leaf des Baumes ist, so liegt die Bounding Box dieses Leafs eventuell nur um wenige Millimeter im View Frustrum, wir zeichnen dann aber blind alle Polygone des Raums. Dabei senden wir auch viele Triangle an das Device die eh bereits ausserhalb des View Frustrums liegen.
Diesem Problem widmet sich der Ansatz der **View Space** Method. (siehe Eberly, 2000)

Nach der Erstellung des BSP Baumes unserer Welt, genannt Welt-Baum, erzeugen wir in jedem Frame einen zweiten BSP Baum, genannt Sichtbarkeits-Baum. Dieser Baum enthält bei seiner Erstellung zunächst nur sechs Partioning Planes, nämlich genau die sechs Ebenen des View Frustrums. Dann beginnen wir damit unsere Welt-Baum zu durchlaufen. Alle Polygone die wir in den Welt-Baum Leafs finden (*idealerweise kombiniert mit dem Bounding Box Test*) senden wir dann an die Wurzel des Sichtbarkeits-Baums und lassen sie durch diesen Baum wandern. Alle Polygone werden nun durch die View Frustrum Ebenen klassifiziert und in der Frontliste der sechsten View Frustrum Ebene landen dann tatsächlich nur genau die Polygone (*oder Teilpolygone nach entsprechenden Splits*) die wirklich im View Frustrum liegen. Das gute an dieser Methode ist, dass wir die Funktionen für die Berechnung des BSP Baumes bereits zur Verfügung haben, ebenso wie die Ebenen des View Frustrums. Die Nachteile dieser Methode sind, dass wir den Sichtbarkeits-Baum wirklich in jedem Frame erstellen müssen was bei modernen Grafikkarten nicht unbedingt schneller ist als das blinde Rendern aller Polys aller Bounding Boxen die wenigstens teilweise im View Frustrum liegen.

Da muss es doch noch eine bessere Methode geben? Und die gibt es auch. Man kann nämlich tatsächlich vorherberechnen, welche anderen Leafs von dem Leaf aus gesehen werden können in dem sich die Kamera befindet. Und das nennt sich PVS.

...oder durch Vorberechnung des PVS

Ein kommerzielles Spiel in der Qualität von Quake braucht natürlich mehr. In diesem Tutorial werden wir dieses *mehr* aus Platzgründen nicht behandeln, aber ich möchte wenigstens erklären was genau dieses *mehr* macht.

Das Zauberwort heisst PVS, oder auch **P**otential **V**isibility **S**et. Unser BSP Baum beruht bisher auf der fundamentalen Basis, dass wir den Baum mehr oder weniger vollständig durchlaufen. Wir besuchen wenigstens alle Äste die im Sichtbereich des Spielers liegen wobei sich auch von diesen wieder sehr viele gegenseitig verdecken werden und zu Overdraw führen. An genau diesem Problem setzt das PVS System an, und es funktioniert wie folgt. Zuerst berechnen wir den vollständigen BSP Baum für unsere Leveldaten. Danach durchläuft das PVS System den Baum und stoppt an jedem Leaf, also an jenen Baumelement die eine Teilmenge von Polygonen konvexer Anordnung enthalten. Nun berechnet das PVS System für jedes einzelne Leaf *exakt* welche anderen Leafs von einem Leaf aus gesehen werden können.

Gehen wir einmal davon aus, dass wir unsere Leafs durchnummeriert haben. Dann berechnet das PVS beispielsweise: *Wenn der Spieler sich in Leaf Nummer 234 befinden, dann kann er von dort aus maximal Leaf Nummer 433, 235, 654, 123 und 456 sehen*. Egal an welcher Stelle im Leaf 234 sich der Spieler befindet und wie er sich auch dreht und wendet. Um uns das besser vorzustellen nehmen wir einmal an das Leaf 234 sei ein ganzes Zimmer mit einer offenen Tür. Egal wie der Spieler sich dreht und wendet er kann auf alle Fälle nur die Leafs sehen die man auch in der realen Welt physisch durch die Tür sehen könnte, keine anderen, selbst wenn diese im View Frustrum des Spielers liegen, denn sie werden durch die Wände des Raumes verdeckt.

Das PVS System berechnet also für jedes Leaf ganz exakt welche anderen Leafs ein Leaf potentiell *sehen* kann und speichert in jedem Leaf eine entsprechende Liste von sichtbaren Leafs. Wenn wir den BSP Baum nun rendern wollen, oder Kollisionsabfragen o.a. durchführen, so laufen wir den Baum nur *einmal*, einen Ast herunter bis wir das Leaf finden in dem der Spieler sich befindet. Dann nehmen wir die dort gespeicherte Liste und wissen sofort ohne einen einzigen Test durchzuführen welche anderen Leafs wir rendern müssen. Selbst bei einem Level in Quake Grösse mit 10'000 oder mehr Polygonen erhalten wir so in rasender Geschwindigkeit eine Menge von sagen wir mal 800 Polygonen (*bei sehr offener Geometrie, sonst wesentlich weniger*) die potentiell sichtbar sind. Was aber viel wichtiger ist: *Alle* anderen 9'200 Polygone des gesamten Levels sind *auf alle Fälle nicht sichtbar* und müssen nicht transformiert, gerendert oder sonst irgendwie beachtet werden.
Für die potentiell sichtbaren Leafs führen wir dann auch noch die Bounding Box Tests durch. Der Raum in dem der Spieler sich befindet kann beispielsweise an zwei gegenüberliegenden Wänden je eine Türöffnung haben. Das PVS System speichert dann in der Liste der sichtbaren Leafs für diesen Raum alle Leafs die man durch beide Türöffnungen sehen kann. Physisch kann sich der Spieler aber nie so positionieren, dass er gleichzeitig durch beide Türen blicken kann. Daher enthält die vom PVS generierte Liste eventuell auch Leafs, die nicht sichtbar sind, daher die Bezeichnung Potential Visibility und die Notwendigkeit des Bounding Box Tests. Wir müssen aber nun nicht mehr alle Nodes des gesamten Baumes testen, sondern nur noch die paar Leafs die in der PVS Liste gespeichert sind. Damit sparen wir enorm viel an Berechnungen und unsere Level können theoretisch unendlich gross sein ohne dass die Framerate kleiner wird.

So, das war ein kleiner Theoriekurs in Sachen PVS. Sowohl die Erstellung des BSP Baumes als auch die PVS Berechnung (*für die man Portale verwendet*) finden natürlich normalerweise vor dem Start unseres Programms statt, da die entsprechenden Berechnungen sehr zeitaufwendig sind. Wenn wir also unser Quake starten, so wird der vorher berechnete und gespeicherte, fertig kompilierte BSP Baum mit PVS Liste in den einzelnen Leafs von der Festplatte als solcher einfach geladen und dann verwendet. Das ist auch der Grund, warum sich die Architektur während des Spiels nicht ändern darf. Zum einen geht dabei die Übereinstimmung des BSP Baumes mit der Levelgeometrie verloren und zum anderen stimmen natürlich die PVS Daten nicht mehr, wenn sich die Leafs (*oder schlimmer noch Polygone verschiedener Leafs*) bewegen.

Und wie geht es jetzt weiter?

Okay...wow, Zeit sich etwas abzukühlen bevor wir diese Frage klären. Wir haben nun eine stabile 3D Engine mit der wir unsere 3D Indoor Level am Screen in sinnvoller Geschwindigkeit rendern können, selbst ohne fancy 3D Beschleuniger. Es gibt allerdings noch eine Anmerkung zum Rendern. Wir rufen die Renderfunktion `DrawIndexedPrimitiveUP()` für jedes Leaf des Baumes einzeln auf. Das ist nicht sehr schnell, eher im Gegenteil. Idealerweise würden wir die Indizes auf die Vertices aller sichtbaren Leafs in einer grossen Liste speichern und dann noch nach Texturkoordinaten ordnen. Dann könnten wir alle Indizes in einem Frame die dieselbe Textur verwenden durch einen einzigen Funktionsaufruf rendern und hätten nur so viele Aufrufe der Renderfunktion wie wir verschiedene Texturen haben. Das würde die Geschwindigkeit unserer Engine noch einmal beträchtlich erhöhen.

Zusammen mit dem Kamerasystem des vorletzten Artikels kann der Betrachter auch in dem Level manövrieren und sich alles betrachten. Man wird jedoch schnell feststellen, dass man durch die Wände gehen kann und dass auch die Höhe des Spielers über dem Boden nicht angepasst wird. Beides sind Probleme der sogenannten Kollisionsabfrage. Unsere Polygone am Bildschirm sind natürlich nur bunte 2D Grafiken und keine wirklich soliden Wände durch die man nicht hindurch gehen kann. Der Computer tut schliesslich nur was man ihm sagt und wenn wir ihm sagen bewegen die Kamera nach vorne so tut er das. Ob da eine Grafik ist oder nicht. Wir müssen dem Computer also extra testen lassen ob das eine Wand ist oder nicht.

Denken wir erst mal an die automatische Anpassung der Höhe des Spielers über dem Boden. Alles was man dazu tun muss ist folgendes: In jedem Leaf welches gerendert wird prüft man noch, ob sich die Kameraposition innerhalb der Bounding Box dieses Leafs befindet. Dabei prüft man aber lediglich die x und z Werte voll und prüfen bei dem y Wert lediglich ob die Kamera in oder über der Box ist. Schliesslich kann der Spieler sich ja auch über der Box befinden, sollte aber direkt auf einem Polygon in der Box stehen. Also müssen wir auch alle Boxen testen die unter der Kamera liegen. Haben wir es mit einer solchen Box zu tun, in oder über der sich die Kamera befindet, dann durchlaufen wir die Liste der Polygone dieses Leafs. Von der Position der Kamera aus

starten wir eine Linie über eine genügend grosse Distanz nach unten und prüfen welches der Polygone (*nicht die Ebenen*) diese Linie schneidet. Von allen Polygonen die diese Linie schneiden wählen wir dasjenige aus welches am nähesten unter der Kamera liegt und legen die y Position der Kamera auf den y Wert des Schnittpunktes fest, plus einem Wert von 1.7f für die Augenhöhe über dem Boden. Ich gebe zu dass das mit etwas Mathematik verbunden ist...
Die Kollision mit Wände ist ebenfalls leicht :-) in den Leafs in den die Kamera sich befinden könnte klassifiziert man die Position der Kamera (*nachdem die Kamera nach der Neuberechnung der Höhe immer noch in diesem Leaf ist*) gegen alle Ebenen der Polys in dem Leaf. Die Kamera darf nie auf eine Backseite eines Polys kommen, sonst hat der Spieler gerade eine Wand durchlaufen. Ist dies der Fall so muss die Kamera zurück auf die Frontseite des Polys gesetzt werden (*Position der Kamera plus Normalenvektor der Ebene*). Fertig!

Als nächstes fehlt es der Engine an der Möglichkeit, statische Objekte einzubinden, beispielsweise Tische, Schränke, Lampen, usw. Hierbei gilt es anzumerken, dass das Beispiellevel **NICHT** sehr gut durchdacht ist. In dem Beispiellevel habe ich bereits zu viele detaillierte Objekte eingebaut, die den BSP Baum unnötig vergrössern und die man besser nach der Erstellung des Baumes als statische Objekte eingefügt hätte (*z.B. die Säulen in dem Startraum, Holzstützen im Kellergeschoss, Computerkonsolen im Hangarkontrollzentrum*). Alle diese Objekte unterteilen das Level nicht kritisch, denn sie sind nur Objekte in einem Raum und verdecken andere Geometrie in diesem Raum nur minimal, so dass sich das Einbinden der Objekte in den BSP im Vergleich zur Alternative des Overdraws nicht lohnt. Wie machen wir das also besser?
Die Leveldatei sollte wirklich nur die Architektur aus Wänden, Decken, Böden und Treppen enthalten. Danach erstellen wir den BSP Baum und speichern ihn auf Diskette. Als nächstes haben wir ein Programm mit welchem wir den fertigen BSP Baum laden können. In diesem Programm erstellen wir dann die Objekte wie Tische, Lampen, Computerkonsolen und speichern in dem BSP Baum dann Informationen über die Objekte. Beispielsweise kann man ein Objekt und dessen Bounding Box von der Wurzel ab durch den BSP Baum jagen. Wir prüfen an jedem Node ob die Bounding Box noch komplett in die Bounding Box eines der beiden Kinder passt. Ist dies der Fall so schicken wir das Objekt an diesen Kind-Node. Passt die Box irgendwann nicht mehr in ein Kind-Node so speichern wir das Objekt an dem Node wo es zuletzt in die Bounding Box gepasst hat. Beim Durchlaufen des BSP Baumes für das Rendern prüfen wir dann zusätzlich jeden Node des Baumes auf eine Liste mit Objekten. Finden wir dort welche rendern wir sie einfach, der Z Buffer wird's schon richten **:-)**

Dieselbe Methode verwendet man übrigens auch für bewegte Objekte in einem BSP Level. Türen oder Fahrstühle könnte man beispielsweise in denjenigen Node des Baums einsortieren in dessen Bounding Box das Objekt mitsamt seines gesamten möglichen Bewegungspfades passt. Also eine Tür gehört in den Node in dessen Bounding Box sie sowohl offen als auch geschlossen ganz hineinpasst.
Bei echten animierten Objekten die sich uneingeschränkt bewegen können, wie beispielsweise Gegner, gibt es auch wieder viele Alternativen. Für unseren BSP Baum wäre es vielleicht am sinnvollsten diese Objekte nicht unbedingt in den Baum einzusortieren, sondern eine getrennte Liste für diese zu haben. Vor oder nach dem Rendern des BSP Baums komen dann diese Objekte an die Reihe und werden mit ihren eigenen Bounding Volumes gegen den Viewport getesten und bei potentieller sichtbarkeit gerendert. Erst bei einem PVS macht es Sinn diese Objekte in jedem Frame neu in den BSP Baum einzusortieren und nur diejenigen in den sichtbaren Leafs zu rendern.

Ja...das war eigentlich auch schon das Wichtigste was dem Programm noch fehlt um 3D Indoorlevel von hoher Detailstufe mit vielen Objekten zu bauen und zu verwenden. Ein PVS System zu integrieren wäre natürlich auch eine feine Sache, aber das ist eine Übung für später. Im dritten Band meiner Bücherreihe über 3D Spieleprogrammierung werde ich dann das Beispiel-Spiel ***Pandoras Box*** entwickeln und erläutern in dem natürlich auch statische und animierte Objekte vorkommen und der Spieler sich durch seine Waffen gegen die hartnäckigen Angriffe von Gegnern verteidigen darf. Dazu werden die notwendigen Tools wie beispielsweise ein Leveleditor entwickelt und erläutert und es gibt natürlich die Möglichkeit, den kompilierten BSP Baum mit den Objektlisten auf Festplatte zu speichern und in das Spiel zu laden.

Level mit AC3D bauen

Nun auf ein Wort zum Download: Darin enthalten ist auch ein AC3D Plugin `zfxexport.p` für das Exportieren von Leveldaten aus AC3D in das richtige Format für dieses Tutorial. Kopiert die Datei einfach in das PLUGINS Verzeichnis von AC3D. Es gilt hierbei dass AC3D Level wegen des potentiellen Polygonsplittings nur aus

konvexen Polygonen gebaut werden sollten und alle Polygone einzelne Objekte sind (*keine Gruppen oder Objekte mit mehreren Polys*). Die Farben die man für die Polys festlegt bestimmen dann die endgültigen Farben im gerenderten BSP. Ein weisses Poly hat also helles weisses Licht, ein schwarzes Poly hat kein Licht, ein rotes Poly wird von rotem Licht angestrahlt, usw. Ach ja, nach dem Export muss man sich die Leveldatei mit der Endung *.zfx noch einmal ansehen und die Pfadangaben bei den Texturen korrigieren. AC3D exportiert hier meistens den kompletten Pfad `c:\programme\ac3d` usw. Dann kann man in jedem einfachen Texteditor die Funktion Ersetzen wählen und diesen Pfad durch nichts ersetzen lassen.
Im Download gibt es auch das Beispiellevel das ich erstellt habe. Das ist zwar nicht besonders gross, aber ich dachte mir bevor ich nach tagelang daran baue bringe ich die Sachen lieber online. Ein Raum (sollte mal eine Kirche werden) ist auch noch nicht ordentlich textuiert und schattiert, aber dann könnt Ihr auch gleich ein wenig mit AC3D das Levelbauen über. Der Beweis für die Lauffähigkeit des Codes ist also erbracht und das Level bietet auch komplexe 3D Geometrie um zu zeigen dass das hier kein Fake mit 2,5D statt 3D ist **:-)**

Und hier gibt es den Code des ***gesamten*** Tutorials zum Download: Projektdateien

ACHTUNG: Leider gab es bei AC3D ein paar Probleme mit der Beleuchtung, daher mag der Level nicht sehr schön beleuchtet erscheinen. In AC3D wird eine Punktlichtquelle verwendet die alle Geometrie abhängig von der Rotation der Objekte in der Welt berechnet. Daher kann man den korrekten Farbwert der Polxgone nie so erkennen wie er später in diesem Code hier erscheinen wird.

Weiter geht's zum Kapitel 10...

SPIELEENTWICKLUNG MIT DIRECT3D IM - MINIKAPITEL 10

von Stefan Zerbst - Guestwriter "TheWanderer"

"Natürlich ist das ein Wheinachtsbaum, oder meinst Du damit wird die Brücke getarnt?"
Das Boot

Die 'Cat' in neuem Licht VORWORT VON STEFAN ZERBST

Wer sich die Mühe gemacht hat, die Beispiele dieses Tutorials herunter zu laden und auch zu testen der wird unter Umständen etwas enttäuscht gewesen sein was das 6. Kapitel zum Laden und Rendern von X Files betrifft. Man hat einfach den Eindruck, dass das gezeigte Modell nicht realistisch wirkt. Auf den ersten Blick wird man wohl nicht direkt sagen können, warum das so ist. Das Auge merkt einfach, dass da etwas nicht stimmt. Doch was genau ist dieses etwas...?

Es ist natürlich das Licht. Die Quelle allen Lebens und vor allem auch der Grund warum das Auge eine künstliche Szene als realistisch wahrnimmt oder nicht. In der realen Welt gibt es immer und überall Licht und Schatten, und nicht nur das, sondern vor allem viele verschiedene Helligkeitsstufen der Lichtintensität. In dem bisherigen Code habe ich immer nur ambientes Licht verwendet, welches an jeder Stelle und aus jeder Richtung in der virtuellen Welt gleich stark scheint. Mal abgesehen vom 9. Kapitel, wo das BSP Level bereits mit entsprechenden Schattierungen gespeichert war.

In diesem Minikapitel machen wir uns nun daran, diesen optischen Misstand zu korrigieren und Direct3D Lichtobjekte zu erzeugen. Wie man sehen wird erscheint das Demo dann sofort in einem *ganz anderen Licht*. Dem Auge ist es einfach viel angenehmer einen wenigstens halbwegs real beleuchtete Szene zu betrachten und es wird dem Gehirn melden, dass das hier schon eher nach der Realität aussieht.
Ein besonderes Vergnügen ist es für mich, dass dieses kleine Kapitel von einem Leser geschrieben wurde. So stelle ich mir die aktive Arbeit mit meinen Tutorials vor, nicht nur der rein passive Konsum von Informationen. Ganz nebenbei verdanken wir dem Autor, TheWanderer, auch ein sehr gutes, umfangreiches Tutorial über C++ Klassen und deren Einsatz in Computerspielen welches sich in meiner Tutorial Sektion findet. Danke an TheWanderer **:-)**

Über diese Ergänzung

Ich habe mich in der letzten Zeit mal mit dem Thema Beleuchtung befasst und einige Ernüchterungen dabei erlebt. Aber durch Fragen im Forum und im Chat bin ich letztendlich doch zum gewünschten Ziel gekommen. Ich habe leider vergeblich gute Tutorials zum Thema Lichtquellen gesucht und musste mir alles aus der SDK-Help und kleinen Beispiel-Codes aus Büchern zusammensuchen. Doch was letztendlich durch lesen und fragen herauskam, kann sich für den Anfang wirklich sehen lassen. Ich möchte noch erwähnen, das dies hier KEIN Tutorial ist, sondern nur eine kurze Beschreibung. Ich will euch hier zeigen wie man Zerbies Quellcode aus dem 6.ten Kapitel seiner Tutorial-Reihe so erweitert, das Lichtquellen (genau gesagt drei Scheinwerfer) auf die CAT strahlen. Das mit den Lichtquellen hat mich seit dem ersten kleinen Erfolg in den Bann gezogen und deshalb möchte ich hier wenigstens meine ersten Erfahrungen zu diesem Thema an euch weitergeben.
An dieser Stelle möchte ich mich besonders bei TalisA und Zerbie bedanken, die mir dabei eine grosse Hilfe waren.

Für ganz eilige....die Tastaturbelegung des Demos

In dieser Demo gibt es drei Spotlights welche auf die Cat ausgerichtet sind. Einer für den linken Flügel, einer für den rechten Flügel und einer für den Bereich des Cockpits. Für jeden Spot kann man rotes, grünes und blaues Licht verwenden. Für den mittleren Spot gibt es auch weisses Licht und sogar Schatten. Ausserdem kam man jeden Spot ein- und ausschalten. Auch das gesamte Umgebungslicht kann in seiner Intensität verändert werden.

Spotlight für den linken Flügel:

1 - Licht ein/aus
q - rotes Licht
a - grünes Licht
y - blaues Licht

Spotlight für den rechten Flügel:

3 - Licht ein/aus
e - rotes Licht
d - grünes Licht
c - blaues Licht

Spotlight für den mittleren Flügel:

2 - Licht ein/aus
w - rotes Licht
s - grünes Licht
x - blaues Licht
r - weisses Licht
t - negatives Licht (Schatten)

Umgebungslicht:

o - Umgebungslicht dunkler
p - Umgebungslicht heller

Escape - Programm beenden

PS: desto dunkler ihr das Umgebungslicht stellt, desto schöner ist der Licht-Effekt.

Der Weg des Lichts....

Eines gleich vorweg: Dies ist wie gesagt **KEIN** Tutorial. Der von mir hinzugefügte Quelltext ist "Quick'n Dirty", aber ausreichend kommentiert. Ich werde euch hier zeigen, welche Änderungen ich vorgenommen habe. Sicherlich kann man noch einiges mehr draus machen. Also los gehts:
Als erstes habe ich die Position der CAT etwas verändert. Sie befindet sich nun mittig auf dem Bildschirm. Die Position wird in "`XMain.cpp`" im case-Zweig "`Spiel laeuft`" durch die Funktion "`XUtil_transformiere_xObjekt`" gesetzt. Danach habe ich den Mauspfeil ausgeschaltet (`ShowCursor(false);`) und den Hintergrund auf Schwarz gesetzt.

Einigen von euch wird aufgefallen sein, das die Abfrage der Rotationswinkel falsch war. Ein Vollkreis ist 2*PI, und nicht 360/2*PI wie es im original Quellcode von Zerbie stand. Das Thema wurde schon im Forum unter "Hausaufgaben" behandelt.

Zerbie hat den Quellcode nochmal durchgesehen und mir folgende Ergänzung nahegelegt:
Die Funktion "`void xUtil_render_XObjekt(XOBJEKT *pXObj)`" kann in der Version wie sie bisher im Tutorial vorhanden war, zu Komplikationen führen. Manche Grafikkarten haben da ihre Probleme. Das liegt daran, dass die Szene nicht gestartet wird. Deshalb sind noch die beiden Zeilen "`g_lpD3DDevice->BeginScene();`" und "`g_lpD3DDevice->EndScene();`" hinzugefügt. Damit sollte das Problem behoben sein. Danke an Zerbie!

```
/**
 * Rendert das DXMESH Objekt im X3DMODELL
 */
void xUtil_render_XObjekt(XOBJEKT *pXObj) {
   // Verschiebung und Rotation für Device einstellen
   g_lpD3DDevice->SetTransform(D3DTS_WORLD, &pXObj->matWelt);

   // 3D Modell des XOBJEKT rendern
   g_lpD3DDevice->BeginScene();
      xMod_rendere(&pXObj->Modell);
   g_lpD3DDevice->EndScene();
   } // xUtil_render_XObjekt
/*-------------------------------------------------------*/
```

Als nächstes braucht man natürlich einige Flags, welche den ein/aus-Zustand der Spotlights speichern. Diese drei sind `bool`-Variablen mit den Namen "**Spot1**, **Spot2** und **Spot3**". Deklariert habe ich diese in der **`XMain.cpp`**. Da das Umgebungslicht variabel sein soll, braucht man natürlich eine Variable für dessen Intensität. Diese Variable nennt sich `int ambient` und ist ebenfalls in der XMain.cpp-Datei deklariert.
Um die verschiedenen Licht-Objekte zu erstellen, benötigt man eine **`D3DLIGHT8`** Struktur. Die Strukturen für die 3 Spots werden in der Datei "**`XD3DInit.cpp`**" innerhalb der Funktion "**`XD3D_Szene_initialisieren`**" mit den entsprechenden Werten gefüllt. Die Parameter der Strukturen werde ich am Ende noch etwas genauer erklären. Hat man nun ein Licht-Objekt namens "light" mit den Werten gefüllt, muss man es noch mittels "`g_lpD3DDevice->SetLight (0, &light);`" dem D3DDevice zuweisen. Die 0 Steht hier für die Nummer des Lichts. (Also ist das zweite Licht die Nummer 1 und das dritte Licht die Nummer 2).
Um die Lichter einzuschalten wird "`g_lpD3DDevice->LightEnable (0, true);`" verwendet.
Nun zu der Steuerung der Demo. Ich habe die Tastaturabfrage in der CALLBACK-Funktion (XMain.cpp) erweitert. Das Ändern der Farben und das ein/aus-schalten der Lichter sollte sich von selbst erklären. Einzig das Ändern des Umgebungslichtes ist hier neu. Dies funktioniert mittels "`SetRenderState (D3DRS_AMBIENT,D3DCOLOR_XRGB(R, G, B));`"

Die "D3DLIGHT8 - Struktur"

Die Struktur hat folgende Parameter, welche ich hier nur ganz kurz beschreiben werde (wie gesagt: KEIN Tutorial):

```
D3DLIGHTTYPE  Type;
D3DCOLORVALUE Diffuse;
D3DCOLORVALUE Specular;
D3DCOLORVALUE Ambient;
D3DVECTOR     Position;
D3DVECTOR     Direction;
float Range;
```

```
float Falloff;
float Attenuation0;
float Attenuation1;
float Attenuation2;
float Theta;
float Phi;
```

- **Type** bestimmt die Art der Lichtquelle. Es gibt Punktlicht (`D3DLIGHT_POINT`), Direktes Licht (`D3DLIGHT_DIRECTIONAL`) und Scheinwerferlicht (`D3DLIGHT_SPOT`).
- **Diffuse**, **Specular** und **Ambient** sind die RGB-Werte des jeweiligen Lichttyps. In der Demo wird ausschliesslicht Ambient verwendet (Umgebungslicht).
- **Position** beinhaltet den x, y und z-Wert der Position des Lichtes.
- Der **Direction**-Eintrag benötigt einen **`D3DVECTOR`**. Diesen habe ich auf (0,0,1) gesetzt, damit das Licht auf die Cat gerichtet ist.
- **Range** bestimmt die maximale Reichweite des Lichts.
- **Theta** ist die Grösse des Inneren Lichtkegels und **Phi** die Grösse des äusseren Lichtkegels (nur bei Spotlight benötigt).
- **Falloff** bestimmt die Abschwächung des Lichts zwischen den beiden Kegeln. Man kann somit harte und weiche Übergänge an den Rändern erstellen.
- **Attenuation0**: konstante Abschwächung des Lichts
- **Attenuation1**: linieare Abschwächung des Lichts
- **Attenuation2**: quadratische Abschwächung des Lichts

Eine genauere Beschreibung der einzelnen Elemente findet man in Zerbies Buch oder in der DX8-SDK Doku.

Wenn ihr Fehler im Quelltext findet, oder vielleicht irgendwelche guten Verbesserungsvorschläge oder auch Fragen habt, dann schickt mir doch eine E-Mail.

RealWanderer@gmx.net

Und hier gibt es den Code des *gesamten* Tutorials zum Download: Projektdateien

---

Weiter geht's zum Kapitel 11...

SPIELEENTWICKLUNG MIT DIRECT3D IM - KAPITEL 9

---

von Stefan Zerbst

"Dodge this."
The Matrix, Trinity

Solide BSP Bäume und Kollisionsabfragen

Erfreulicherweise hat dieses Tutorial für so viele Nachfragen gesorgt, dass es kaum auf eine Kuhhaut passt - die meisten Fragen kamen natürlich zum neunten Kapitel, den BSP Bäumen. Auf Platz 1 der Frageliste war natürlich, wie man eine Kollisionsabfrage in den BSP Baum integriert. Ein wenig Theorie in den Raum zu werfen hat anscheinend nicht immer gereicht, bzw. ermutigte Ansätze in die falsche Richtung. Daher entstand dieses Kapitel darüber, wie man den BSP Baum aus dem neunten Kapitel so erweitert, dass er Kollisionen an der Levelgeometrie erkennt. Die Lösung für dieses Problem ist der sogenannte **Solid Leaf** BSP Tree. Erstaunlicherweise ist es überhaupt gar nicht so schwer, eine Kollisionsabfrage in den BSP Baum zu integrieren. Daher stellt sich gleich die Frage, warum ein eigenes Kapitel dazu? Warum kein Update des neunten Kapitels?

Die Antwort auf diese Frage ist sehr einfach zu geben. Es gibt mindestens ebenso viele Möglichkeiten, eine Kollisionsabfrage in den Baum zu integrieren wie es schlechte Boygroups gibt. Aus dieser schier unendlichen Anzahl muss man sich für einen Ansatz entscheiden und dieser hat natürlich Konsequenzen auf den Rest des Codes. Die Möglichkeit, die wir hier kennen lernen werden, birgt sowohl Vorteile als auch Nachteile. Das ist der eine Grund, warum ich den das neunte Kapitel erst mal so stehen lasse wie es war. Wer einen anderen Ansatz wählen will, der kann ausgehend von dem nackten Leaf BSP Tree seinen eigenen Weg gehen.
Der andere Grund ist, dass wir hier einige gedankliche Spagats machen werden und es ist gut, wenn man erst mal den reinen BSP Algorithmus so weit verinnerlicht hat, dass man den Kopf frei hat für andere Dinge. Um dem Schrecken gleich wieder den Wind aus den Segeln zu nehmen sei folgendes zur Beruhigung angemerkt: Wir werden den Code des reinen BSP an einer Stelle minimal verändern und dazu noch zwei neue Funktionen implementieren, die beide recht kurz sind. Es ist also lediglich gedankliche Arbeit und keine riesige Fleissaufgabe.

Nun zu dem Ansatz den wir gehen werden und warum ich mich dafür entschieden habe. Der im folgenden präsentierte Algorithmus hat einen gravierenden Nachteil. Er wird dafür sorgen, dass sich unser bisheriger BSP Baum bei gleicher Levelgeometrie um einiges vergrössert. Die Äste des Baumes werden also länger und die Leaves werden tiefer im Baum liegen. Daraus entsteht der Nachteil, dass wir mehr Zeit brauchen, um den Baum zu durchlaufen. Auf der anderen Seite steht aber ein grosser Vorteil. Wir brauchen keine komplizierten, aufwendigen Berechnungen mit Punkten gegen Polygone oder so etwas. Alles was wir für die Kollisionsabfrage benötigen ist ein simpler Test bei dem wir einen Punkt gegen eine Ebene klassifizieren. Hierbei sparen wir also wieder einiges an Zeit ein, da die Kollisionsabfrage quasi ohne Berechnung von statten geht.

Aber wiegt dieser Vorteil nun den Nachteil auf? Ein bisschen rechnen müssen wir ja doch und wenn der Baum sooooooooooo viel länger wird...! Nun, das führt uns zurück auf den ersten Grund warum dies ein eigenes Kapitel ist. So wie sie hier präsentiert wird ist die Kollisionsabfrage ein Kompromiss aus einem Vorteil und einem Nachteil. Man könnte sich also auch für einen anderen Ansatz entscheiden, der den Baum nicht verlängern würde. **Aber**: Wenn wir ein PVS System in unser Programm integrieren, dann durchlaufen wir den Baum ja gar nicht mehr beim Rendern. Und was viel wichtiger ist. Um ein PVS zu integrieren eignet sich ein Solid Leaf BSP Tree auf alle Fälle wesentlich besser, als ein reiner Leaf BSP Tree. Für alle, die ein PVS erzeugen wollen, ist dieser Weg der Kollisionsabfrage also der Weg, der zu gehen ist.

Nachdem das nun geklärt ist, könnten wir ja eigentlich anfangen...

Was ist ein Solid Leaf BSP Tree?

Jetzt haben wir schon ein paar Dutzend mal um den heissen Brei herumgeredet und wissen immer noch nicht, was *genau* denn der Ansatz ist den wir gehen werden. Zur Erinnerung schauen wir uns noch mal das Demolevel aus dem 9. Kapitel an und wie der Leaf BSP Tree aussah, den wir daraus erzeugt haben. Man beachte, dass die kleinen schwarzen Striche an den Polygonen zu deren Front Seite zeigen und die hochgestellten Punkte hinter den Polygon Nummern in dem Baum angeben, welche der Polygone als Splitter verwendet worden. Hier waren das die Polygone 3, 8 und 9.

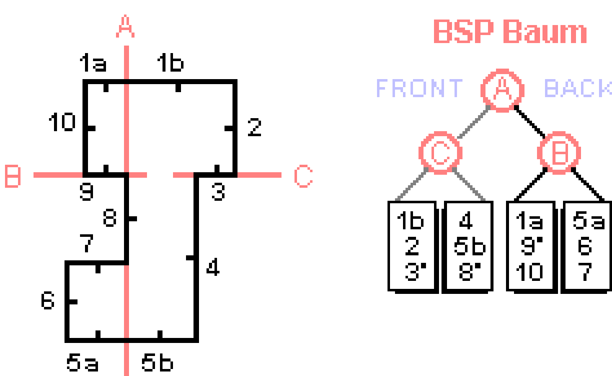


Abbildung 1: Leaf BSP Tree aus Kapitel 9

Wie man sehen kann benötigten wir bei dieser Geometrie lediglich drei Ebene (A, B und C) die den Level in konvexe Häppchen teilen. An den drei Knoten des Baumes, die über den vier Leaves stehen, haben wir also Informationen über die Ebenen der Teilungspolygone (*Splitter*) gespeichert. Wenn wir ausgehend von der Wurzel also in den linken Ast rutschen, dann wissen wir am Knoten C welche Ebene hier ein *Hindernis* dargestellt hat welches die Konvexität der Geometrie verhindert hat. Diese Information werden wir nun für die Kollisionsabfrage nutzbar machen. Die Ränder eines Leafs sind schliesslich genau die Ebenen, mit denen die Kamera kollidieren könnte.
Schauen wir uns mal das Leaf an, welches die Polygone 4, 5b und 8 enthält. Das Leaf wird einmal durch seine Polygone begrenzt, und zum anderen durch die Splitterebenen A, B und C (wobei B und C identisch sind). Weiter oben hatten wir bereits erwähnt, dass es sehr aufwendig ist eine potentielle Kollision mit einem Polygon zu berechnen. Vielmehr wäre es sinnvoller, wenn wir den zu prüfenden Punkt einfach gegen die Ebenen aller Polys in dem Leaf testen würden. Wir klassifizieren den Punkt (*beispielsweise die Kameraposition*) also einfach gegen die Ebenen der Polygone. Ist der Punkt z.B. im Back Bereich des Polys 5b, so ist das ausserhalb der Levelgeometrie und der Punkt darf sich dorthin nicht bewegen. Sehr schön...aber halt. Wenn der Punkt im Back Bereich des Polys 8 oder 4 ist, dann kann der Punkt zwar ausserhalb des Leafs sein, aber dennoch innerhalb gültiger Geometrie des Levels landen, nämlich in den konvexen Happen aus 7, 6 und 5a oder 1b, 2 und 3. So einfach ist es also nicht.
Wir brauchen als erstes eine Möglichkeit, zwischen gültigem Raum innerhalb der Levelgeometrie und ungültigem Raum ausserhalb der Levelgeometrie zu unterscheiden. Diesen ungültigen Raum nennt man im Fachjargon **Solid Space** (*solider Raum*). Jeder Bereich im 3D Raum, der nicht von Geometrie (*Wänden, Boden, Decke*) umschlossen ist, darf nicht betreten werden können und wird daher als solide bezeichnet. Der gesamte andere Raum, also der gültige Raum, wird allgemein hin als Empty Space (*leerer Raum*) bezeichnet. Nur in diesem leeren Raum kann sich ein Objekt der Welt aufhalten. Sehen wir uns mal an, wie das in dem Demo Level aussehen würde.

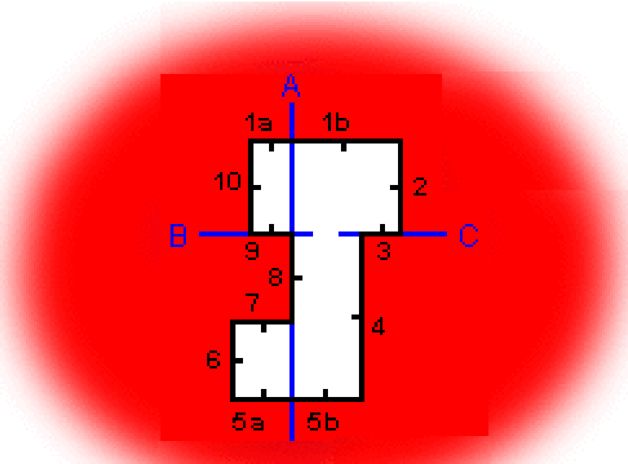


Abbildung 2: Solider Raum des Demo Levels

Alles was in der **Abbildung 2** rot gefärbt ist, ist solider Raum den kein Objekt je betreten kann. Alles was weiss gefärbt ist, ist leerer Raum. Für unsere Kollisionsabfrage heisst dies also, ein Punkt darf sich überall in dem Level hin bewegen wo es nicht rot ist. Und genau diesen Test können wir sehr einfach und ohne zusätzlichen Rechenaufwand ausführen. Wir werden unseren BSP Baum so umgestalten, dass wir auch Informationen über den soliden Raum in diesem speichern. Bleiben wir bei dem oben bereits herausgegriffenen Leaf aus den Polygonen 4, 5b und 8.

In solidem Raum befindet sich ein Punkt genau dann, wenn als '*im Back Bereich*' einer Ebene des Leafs aus dem er kommt klassifiziert wird, die nicht als Splitting Plane verwendet wurde.

Wenden wir uns einem Beispiel zu. Der Punkt der auf Kollision zu prüfen ist, ist die Kamera. Diese befindet sich in dem oben erwähnten Leaf. Bewegt sie sich auf die Backseite des Polygons 5b so ist sie auf alle Fälle in solidem Raum. Dasselbe gilt für das Polygon 4. Die Kamera kann aus diesem Leaf heraus nicht auf die Back Seite der Ebene des Polygons 4 kommen, ohne in solidem Raum zu landen. Wäre die Kamera in dem Leaf oben rechts aus den Polygonen 1b, 2 und 3 so könnte sie sehr wohl in den Back Bereich der Ebene von Poly 4 gelangen und dennoch in gültigem Raum (*dicht links von Poly 2*) sein. Aber lesen wir obigen Merksatz noch mal: Der Punkt wird nur gegen die Ebene der Polygone getestet die zu dem Leaf gehören aus dem er kommt. Und nun zum Poly 8. Wenn die Kamera aus dem Leaf kommt kann sie auf die Back Seite der Ebene von Poly 8 kommen und dennoch in gültigem Raum landen. Nämlich in dem Leaf aus den Polys 5a, 6 und 7. Es findet also keine Kollision statt. Und auch das stimmt mit unserer Regel überein. Denn wie wir aus **Abbildung 1** ersehen können ist Polygon 8 bereits als Splitter im normalen Leaf BSP Algorithmus verwendet worden, und in diesem Fall darf die Kamera auf dessen Back Seite gelangen.

Und nun bitte an den Stuhllehnen festhalten...das war bereits alles, was wir über Kollisionsabfragen in Leaf BSP Bäumen wissen müssen. Also machen wir uns daran, einen Algorithmus zu entwerfen mit dem wir den Baum so umgestalten können, dass wir das alles schnell und zügig prüfen können.

Theorie und Algorithmus der soliden BSP Bäume

So, was ist nun zu tun? Wir haben die obige Regel für Kollisionen (*rot markierter Merksatz*) ja bereits formuliert. Die Frage ist, wie implementieren wir das ganze geschickt? Wir könnten nun einfach den BSP Baum so lassen. Jedes mal wenn die Kamera sich bewegen will, dann schauen wir nach in welchem Leaf sie sich befindet. In diesem Leaf laufen wir dann durch die Liste der dort gespeicherten Polygone und klassifizieren die Kameraposition einfach gegen die Ebenen aller Polygone in diesem Leaf, die noch nicht als Splitter markiert sind. Kommt bei einem dieser Tests `BSP_BACK` heraus, so ist die Kamera in solidem Raum gelandet.

Natürlich werden wir nicht die aktuelle Kameraposition auf Kollision prüfen, sondern die Position an die der Spieler die Kamera in diesem Frame bewegen möchte. Landet die Kamera mit dieser gewünschten Position in solidem Raum, oder müsste solchen durchqueren um zur angestrebten Position zu kommen (*z.B. durch eine dünne Wand*), so sagen wir, dass eine Kollision aufgetreten ist und belassen die Kamera auf ihrer aktuellen Position.

In diesem Fall benötigten wir aber eine Schleife, und damit Overhead für eine Schleifenvariable und so weiter. Daher implementiert man den soliden Raum normalerweise ein wenig anders. Wir konstruieren unseren BSP Baum genauso wie bisher auch. Dann nehmen wir jedes Leaf und starten eben jene Schleife von der wir grad gesprochen haben. Das Leaf ist nun auch kein Leaf mehr, sondern ein fast ganz normaler Node in dem BSP Baum. Immer dann, wenn wir ein Polygon gefunden haben das bisher noch kein Splitter war, dann erzeugen wir einen neuen Node für den Front Pointer des aktuellen Nodes. ***<u>Alle</u>*** Polygone des ursprünglichen Nodes (*der ja bisher unser Leaf war*) werden in die Frontliste kopiert. Unser bisheriges Leaf bestand ja eh nur noch aus konvexer Geometrie und da liegen alle Polygone in Front von einander. Und wenn ich sage ***<u>alle</u>***, und das noch dick und kursiv unterstreiche, dann meine ich auch alle, einschliesslich des Splitters. Wie gehabt wird die Ebene des eben als Splitter gewählten Polygons in dem Node gespeichert und im Polygon wird vermerkt, dass es nun auch Splitter war. Und nun die Besonderheit: Die Backliste des Nodes (*der bisher unser Leaf war*) erhält kein Node Objekt, sondern wird auf `NULL` gesetzt. Und eben dieser Back Pointer der auf `NULL` steht und nicht auf ein Node Objekt zeigt, repräsentiert soliden Raum in unserem BSP.
Diesen Vorgang wiederholen wir so lange, bis alle Polygone unseres bisherigen Leafs, einmal Splitter gewesen sind. Wir erzwingen also die Verwendung aller Polygone der Geometrie als Splitter und verwenden nicht nur diejenigen als Splitter, die der normale BSP Algorithmus ausgewählt hätte. Bei den normalen Splittern haben wir wie gehabt die üblichen Front- und Backlisten. Bei den erzwungenen Splittern ist die Backliste immer `NULL` und die Frontliste zeigt auf den nächsten erzwungenen Splitter und schlussendlich auf ein Leaf. Ein Leaf in einem Solid Leaf BSP Tree haben wir dann erreicht, wenn alle Polygone des konvexen Happens, den dieser Ast repräsentiert, als Splitter verwendet wurden. In dem Leaf findet sich dann wie gehabt die Liste aller Polygone des konvexen Happens.

Hui...da gibt es erst mal wieder viel zu verdauen, obwohl es wirklich viel einfacher ist als es zunächst klingt. Schauen wir uns mal an, wie der Solid Leaf BSP Tree für das Demo Level aussehen würde.

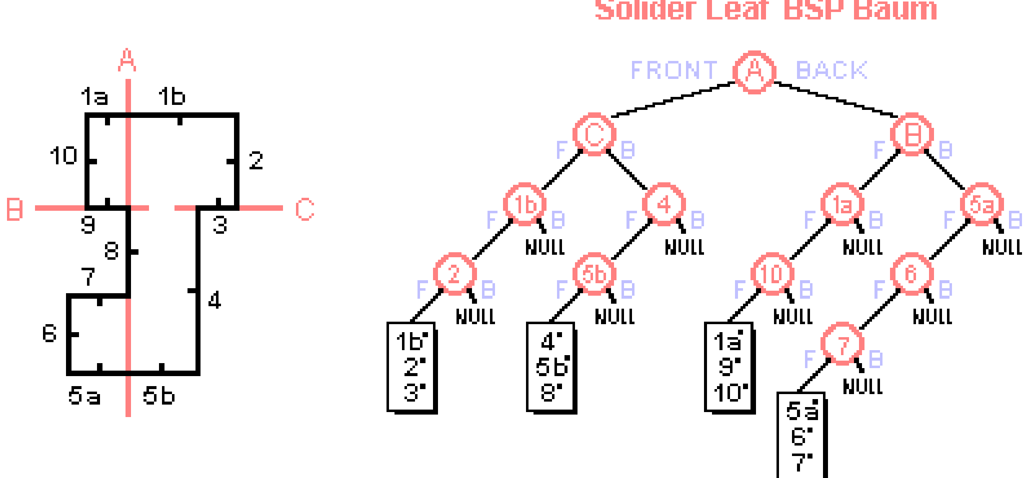


Abbildung 3: Kompletter Solid Leaf Tree des Demo Levels

Hier sieht man auf den ersten Blick, wie wuchtig der Baum geworden ist. Logischerweise gibt es nun für jedes Polygon in dem Level auch einen eigenen Node. Als nächstes fällt auf, dass der Baum so lange mit dem normalen Leaf BSP Baum identisch ist, bis die ursprünglichen Leafs dieses Baumes erreicht sind. Und nun passiert das, was wir schon besprochen haben. Jedes der ursprünglichen Leafs wird weiter unterteilt, die Polygone rutschen in die Frontliste und die Backliste wird auf `NULL` gesetzt, um soliden Raum kenntlich zu machen.
Die roten Zahlen in den neuen Nodes geben an, die Ebene welches Polygons hier als Splitter gespeichert wurde. Diese Splitter teilen nun nicht mehr die Geometrie, wie die Splitter A, B und C dies getan haben, sondern sie unterteilen den Raum in gültigen Levelraum (*Empty Space*) in den Frontlisten und in ungültigen Raum (*Solid Space*) in den Backlisten. Diese Unterteilung gilt aber wie ganz oben erläutert nur innerhalb eines Astes, also ab

unterhalb der normalen Splitter (mit Buchstaben markiert), denn unser roter Merksatz besagt ja, dass die Ebenen der Nicht-Normalen Splitter soliden Raum nur innerhalb desselben Leafs anzeigen.

In welcher Reihenfolge wir die Polygone der ursprünglichen Leafs als Splitter erzwingen ist übrigens vollkommen egal. Hier bin ich einfach nach der Nummerierung gegangen, und in der Regel wird man die Polygone so abarbeiten, wie sie in der Polygonliste der konvexen Happen stehen.

Okay, damit haben wir auch schon den Algorithmus zur Konstruktion des Solid Leaf BSP Baums vollständig besprochen. Also beginnen wir mit der Implementierung basierend auf dem Code aus Kapitel 9.

Zwangsarbeit - Jeder muss mal splitten

Den bisherigen Code übernehmen wir unverändert. Unser alter Algorithmus erzeugt uns also den normalen Leaf BSP Tree. Unser Job ist es nun, an der Stelle einzugreifen an der der alte Algorithmus ein Leaf erzeugt hatte. Das passierte in der Funktion `XBSP::Erstelle_BSPBaum`. Hier gab es eine `if` Abfrage, ob die boolsche Variable `blnErgb` anzeigt, dass wir einen Splitter gefunden haben. War dies nicht der Fall, so hatten wir es mit einem konvexen Happen Geometrie zu tun und konnten den behandelten Node als Leaf markieren und hatten einen rekursiven Ast beendet.

Genau an dieser Stelle setzen wir jetzt an, und erzwingen die weitere Verästelung des BSP Baumes, indem alle Polygone des konvexen Happens einmal Splitter gewesen sein müssen. Dafür implementieren wir die Funktion `XBSP::Mache_Leaf_solide` und rufen sie an der eben erwähnten Stelle wie folgt auf:

```
void XBSP::Erstelle_BSP_Baum(XNODE *pNode) {
   XNODE *pFrontnode, *pBacknode;
   LONG   lAnz_F=0, lAnz_B=0;
   BOOL   blnErgb;
   int    nKlasse;

   // Berechne die Bounding Box um alle Polygone des Nodes
   pNode->xBox = this->Erstelle_BoundingBox(pNode->pPolys,
   pNode->lAnz_Polys);
   // Finde die beste Teilungsebene dieses Nodes
   blnErgb = this->Finde_besten_Splitter(pNode->pPolys,
                                         pNode->lAnz_Polys,
                                         &pNode->xEbene);

   // Falls keine Ebene gefunden ist dieser Node ein Leaf
   if (!blnErgb) {
      // Das Leaf machen wir solide für Collision/LineOfSight
      this->Mache_Leaf_solide(pNode);
      return;
      }
   [...]
```

Die Implementierung dieser Funktion ist...ich bin fast versucht zu sagen '*so einfach wie Pfannkuchen essen*' *g*. Wir nehmen einfach den Node der der Funktion übergeben wurde und durchlaufen alle Polygone die in dessen Liste gespeichert sind in einer Schleife. Sobald wir eines finden, welches noch kein Splitter war, setzen wir den Back Pointer des Nodes auf `NULL` um soliden Raum zu markieren und erzeugen einen neuen Node für die Front dieses Nodes. In dem Node selber speichern wir dann die Ebene des Polygons, welches wir eben gefunden haben und nun als Splitter erzwingen. Die Liste der Polygone speichern wir komplett um in den neu erzeugten Frontnode und brechen die Schleife über die Anzahl der Polygone ab. Dabei rufen wir schnell noch die Funktion rekursiv auf, denn nur so erhalten wir alle bisher noch nie als Splitter verwendeten Polygone. Sehen wir uns das einmal im Code an:

```
void XBSP::Mache_Leaf_solide(XNODE *pNode) {
   BOOL blnSplit=FALSE;
```

```
   // Gibt es ein Poly, dass noch kein Splitter war
   for (int i=0; i<pNode->lAnz_Polys; i++) {
      if (pNode->pPolys[i].blnWar_schon_Splitter == FALSE) {
         blnSplit = TRUE;

         // Neuen Node erstellen
         if (g_lPool_Index >= (BSP_NODES_MAX-1)) {
            fprintf(Protokoll, "Fehler: MAX_NODES erreicht\n");
            return;
            }
         pNode->pFront = &g_aNode_Pool[g_lPool_Index++];

         // Dieses Poly wird nun der Splitter
         pNode->pPolys[i].blnWar_schon_Splitter = TRUE;

         // Ebene des Splitters im Node speichern
         pNode->xEbene = pNode->pPolys[i].xEbene;

         // Attribute von diesem Node kopieren
         pNode->pFront->lAnz_Polys = pNode->lAnz_Polys;
         pNode->pFront->blnLeaf    = FALSE;
         pNode->pFront->pBack      = NULL;
         pNode->pFront->pFront     = NULL;
         memcpy(&pNode->pFront->xBox, &pNode->xBox, sizeof(XBOX));

         // Alle Polys der Liste sind in Front voneinander
         pNode->pFront->pPolys = (XPOLY*)malloc(sizeof(XPOLY)
                                   * pNode->lAnz_Polys);
         memcpy(pNode->pFront->pPolys, pNode->pPolys,
                sizeof(XPOLY) * pNode->lAnz_Polys);

         // Speicher freigeben
         free(pNode->pPolys);

         // Backliste ist SOLIDE
         pNode->pBack = NULL;

         // Node rekursiv zerlegen bis alle Splitter waren
         this->Mache_Leaf_solide(pNode->pFront);

         // Schleife beenden
         break;
         } // if
      } // for

   // Leaf Flags setzen falls kein Split mehr nötig war
   if (blnSplit == FALSE) {
      pNode->blnLeaf = TRUE;
      pNode->pBack = NULL;
      pNode->pFront = NULL;
      }
   return;
   } // Mache_Leaf_solide
/*----------------------------------------------------------------------*/
```

Die Variable `blnSplit` steuert das Ende der Rekursion. Wenn wir es schaffen, einmal die Polygonliste eines Node Objektes zu durchlaufen, ohne dabei ein Polygon zu finden das noch kein Splitter war (*normal oder*

erzwungen), dann sind wir an der Stelle angelangt, wo wir ein Leaf des Solid Leaf BSP Baums erzeugen können. Dazu setzen wir einfach die entsprechenden Flags des Nodes und die Arbeit ist getan.

Zum wiederholten Entsetzen aller sage ich es mal wieder: Das war schon alles! Dieses winzige Funktiönchen sorgt dafür, dass aus unserem Leaf BSP Tree ein Solid Leaf BSP Tree wird. Wir müssen lediglich beim Rendern acht geben, dass wir jetzt auch prüfen müssen, ob der Back Pointer eines Nodes `NULL` ist, bevor wir die Renderfunktion rekursiv mit dem Back Pointer aufrufen, da dieser Fall nun eintreten kann. Sonst ändert sich aber nichts und unser Demo ist fertig um getestet zu werden.

Man sollte keinerlei Unterschied zu dem vorherigen Demo bemerken. Aber damit es nicht zu langweilig und überhaupt alles zu einfach wird, erledigen wir jetzt noch das, weswegen wir eigentlich hier sind: Die Implementierung der Kollisionsabfrage.

Warum alles doch nicht so einfach ist

Soooooooooooo, jetzt haben wir bisher so viele leichte Spaziergänge gehabt, davon sollten wir uns nicht allzu schnell täuschen lassen. Die Abfrage einer Kollision mit der Levelgeometrie ist wirklich so einfach, wie oben dargestellt. Aber implementierungstechnisch betrachtet müssen wir dabei ein paar Dinge berücksichtigen. Wir werden also mit einer Kollisionsfunktion enden, die etwas wuchtig aussieht. Davon sollte man sich aber nicht abschrecken lassen, denn dabei handelt es sich um viel fast identischen Code diverser Fallunterscheidungen die wir treffen müssen.
Zuerst mal korrigieren wir die Aussage, dass eine Kollision stattgefunden hat, wenn ein Punkt nach seiner Bewegung im soliden Raum liegt. Beziehungsweise, diese Aussage ist korrekt aber der Umkehrschluss, dass keine Kollision stattgefunden hat wenn der Punkt vor und nach seiner Bewegung im leeren, gültigen Raum liegt, ist falsch. Im folgenden ist eine Situation noch mal grafisch dargestellt, wo man erkennen kann warum das so ist.

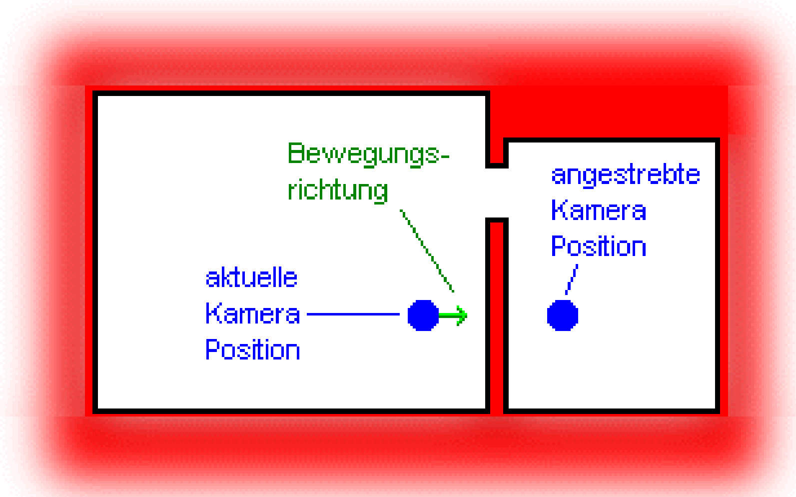


Abbildung 4: Kollision trotz gültiger Kameraposition

Wenn die Schrittweite der Kamera sehr gross ist, dann kann die Kamera nämlich durch dünne Wände gehen. Sagen wir, unsere Kamera darf mit einer Geschwindigkeit von 0.5 Einheiten je Frame rennen. Wenn eine Wand in unserem Level aber beispielsweise nur 0.3 Einheiten dick ist, dann kann die Kamera vor der Wand stehen und im nächsten Frame mit einem grossen Schritt durch die Wand gegangen sein und sich wieder ganz unschuldig im gültigen Raum befinden. Die beiden blauen Punkte in **Abbildung 4** symbolisieren jeweils zwei gültige Kamerapositionen in dem Level, zwischen denen die Kamera in einem Schritt wechseln kann. Dass zwischen den beiden gültigen Positionen aber eine Wand liegt würden wir nicht bemerken.

Daraus lernen wir, dass es nicht reicht die gewünschte Position des zu testenden Punktes zu prüfen. Vielmehr müssen wir eine Linie ziehen, die von der aktuellen Position des Punktes zu der gewünschten Position des Punktes verläuft. Dieses Liniensegment darf mit keinem Millimeterchen im soliden Raum liegen, sondern muss komplett innerhalb gültigen Raums sein. Und genau so eine Funktion werden wir für unsere Kollisionsabfrage implementieren. Die **Abbildung 5** zeigt noch einmal grafisch, wie wir bei selbiger Situation wie oben dennoch eine

Kollision erkennen. Das Liniensegment zwischen den beiden Positionen darf an keiner Stelle im soliden Raum liegen, sonst gibt es ein Hindernis auf dem gradlinigen Weg von einer Position zur nächsten.

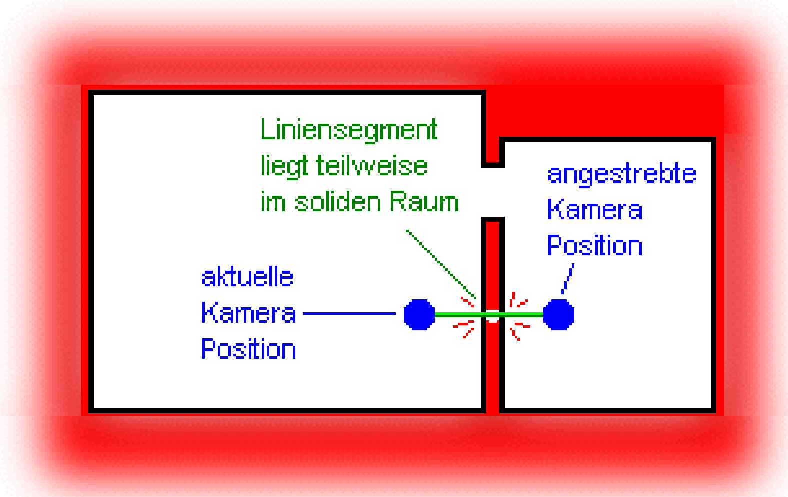


Abbildung 5: Liniensegment zwischen zwei Punkten

Und nun noch ein kleiner Leckerbissen den wir als Bonus erhalten, wenn wir die Kollisionsabfrage durch einen Solid Leaf BSP Baum implementieren. Machen wir uns noch einmal klar, was die Funktion machen wird die wir gleich implementieren. Sie wird prüfen, ob eine Linie zwischen zwei gegebenen Punkten im 3D Raum besteht die nicht durch soliden Raum läuft. Anders ausgedrückt prüft diese Funktion, ob eine *Sichtlinie* zwischen den beiden Punkten besteht. Oder um es noch anders auszudrücken, und nebenbei mit einem ganzen Zaun zu winken, prüft diese Funktion, ob der eine Punkt den jeweils anderen *sehen* kann, ohne dass eine Wand dazwischen ist. Diese Funktion eignet sich also nicht nur dafür zu testen, ob die Kamera (*oder ein beliebiges anderes Objekt*) gegen eine Wand gelaufen ist. Vielmehr kann diese Funktion dazu eingesetzt zu werden um zu entscheiden, ob sich zwei Objekte in einem 3D Level sehen können.
Nehmen wir einmal an, dass der eine Punkt die Position eines Monsters in dem Level ist. Der andere Punkt ist die Position der Kamera. Nun prüfen mit unserer Funktion, ob eine Sichtlinie zwischen den beiden Punkten besteht. Ist dies der Fall, dann sollte das Monster auf den Spieler reagieren. Besteht keine Sichtlinie so wissen wir, dass sich zwischen dem Monster und dem Spieler noch Wände des Levels befinden und sie sich nicht gegenseitig sehen können. Die künstliche Intelligenz, welche das Monster steuert, sollte wenigstens so fortschrittlich sein, dass das Monster auch wirklich in einem Zustand ist, wo es nicht weiss, wo der Spieler ist da es ihn nicht sehen kann. Es sollte also nicht bewusst in Richtung des Spielers laufen und ihn jagen weil es aufgrund schlechter Programmierung *allwissend* ist. Das wäre dem Spieler gegenüber unfair, der wirklich nur weiss, was er auch sehen kann.

Aber jetzt drifte ich ab in Richtung KI Programmierung. Zurück auf den Boden der Tatsachen. Für alle intelligent agierenden Objekte, die wir später in unser Level setzen, können wir durch unsere Funktion prüfen, ob sie sich gegenseitig oder den Spieler sehen können. Und nun ist es Zeit, dieses Wunderding endlich zu implementieren.

Sichtlinie zwischen zwei Punkten im soliden BSP Baum

Wie jede Implementierung beginnt auch diese hier mal wieder mit dem ordentlichen Nachdenken. Im Grunde genommen wollen wir ja nichts anderes tun, als Punkte gegen Ebenen zu klassifizieren. Und zwar genau zwei Punkte. Glücklicherweise haben wir dafür bereits die Funktion `XBSP::Klassifiziere_Punkt` zur Verfügung. Diese kennt die folgenden drei Klassifizierungsmöglichkeiten:

(1) `BSP_FRONT` (2) `BSP_BACK` (3) `BSP_PLANAR`

Bevor es weitergeht sollte ich vielleicht noch mal genau sagen, was wir machen wollen. Wir möchten gerne der Funktion zwei Punkte übergeben, die der Start- beziehungsweise der Endpunkt eines Liniensegmentes sind. Die Funktion nimmt nun diese beiden Punkte und schickt sie von der Wurzel an in den BSP Baum. An jedem Node (*einschliesslich der Wurzel*) werden die beiden Punkte gegen die dort gespeicherte Ebene (*des Splitters*)

klassifiziert. Und hier ein paar logische Regeln für den weiteren Verlauf.

Sind beide Punkte in Front so testen wir im entsprechenden Frontnode weiter (*Rekursion*). Gibt es keinen Frontnode mehr, weil der Node ein Leaf ist dann ist die Sichtlinie gültig. Rückgabewert `TRUE`.

Sind beide Punkte im Back so testen wir im entsprechenden Backnode weiter (*Rekursion*). Ist der Backnode `NULL` dann gibt es keine Sichtlinie weil die Punkte im soliden Raum liegen. Rückgabewert `FALSE`.

Ups...da habe ich doch glatt eine *Winzigkeit* unterschlagen. Es gibt ja auch den (*wohl häufigsten*) Fall, dass die Punkte auf verschiedenen Seiten der Ebene liegen, also einer in Front und der andere in Back. Nun können wir nicht mehr mit beiden Punkten rekursiv in den selben Ast, Front oder Back, absteigen. Denn die Informationen über soliden Raum besitzen ja nur Gültigkeit innerhalb eines Leaf Astes im BSP Baum. Was tun wir nun?
Naja, mit ein wenig Nachdenken ist die Sache ganz einfach (*so wie immer*). Wir machen es wie bei den Polygonen die sich nicht einigen können ob Front oder Back. Wir splitten das Liniensegment in zwei Teile auf und schicken die entsprechenden Teile einfach in die jeweilige Liste Front und Back. Damit verzweigt sich unsere Funktion nicht in einen, sondern in gleich zwei rekursive Aufrufe. Und nur dann wenn beide Liniensegmente im gültigen Raum sind (*beide rekursive Aufrufe geben* `TRUE` *zurück*) dann geben wir für die gesamte Linie auch `TRUE` zurück, denn anderenfalls liegt mindestens ein Stück des Liniensegmentes im soliden Raum und es gibt folglich keine Sichtlinie.

So nun haben wir auch schon die gesamte Theorie unserer Sichtlinienfunktion erarbeitet. Zur Verdeutlichung noch mal zu den Rückgabewerten von `XBSP::Klassifiziere_Punkt`. Wenn wir nun ein Pärchen aus zwei Punkten klassifizieren so erhalten wir zwei Klassifizierungen die wir, wie eben besprochen, nicht unabhängig voneinander betrachten dürfen, wenn wir uns an die Ergebnisauswertung machen. Folglich haben wir bei der Implementierung ein paar Fallunterscheidungen in unserer Funktion, und zwar vier Stück. Eigentlich gibt es ja neun Möglichkeiten der Ergebnispärchen, aber ein paar Fälle werden wir gleich behandeln. Die folgende Tabelle gibt eine Übersicht darüber, welche Fälle wir behandeln und wie das grafisch aussieht.

Fall	Startpunktklasse	Endpunktklasse	Abbildung
1	**BSP_FRONT**	**BSP_BACK**	Klassifizierungsebene FRONT BACK Start Ende
2	**BSP_BACK**	**BSP_FRONT**	

3.1 3.2 3.3 3.4	**BSP_FRONT** BSP_PLANAR BSP_FRONT BSP_PLANAR	**BSP_FRONT** BSP_FRONT BSP_PLANAR BSP_PLANAR	Klassifizierungsebene FRONT BACK Start Ende **Fall 2** Klassifizierungsebene FRONT BACK Start Ende
4.1 4.2 4.3	**BSP_BACK** BSP_PLANAR BSP_BACK	**BSP_BACK** BSP_BACK BSP_PLANAR	Klassifizierungsebene FRONT BACK Start Ende **Fall 4**

Die ersten beiden Fälle sind natürlich die kompliziertesten, weil wir hier das Liniensegment teilen und in zwei getrennte rekursive Aufrufe schicken müssen, deren Rückgabeergebnis wir zusammenfassen. Die anderen beiden Fälle sind genauso leicht, wie man sich das erhoffen kann. Es handelt sich dabei um die beiden, ein paar Zeilen

weiter oben rot markierten, Fällen in denen die Funktion sich einfach rekursiv verzweigt beziehungsweise einen echten Ergebniswert `TRUE` oder `FALSE` zurück geben kann. Die Klasse `BSP_PLANAR` wird hier ein wenig mit Ignoranz gestraft. Sind beide Punkte planar, so behandeln wir diesen Fall ebenso, als wenn beide in Front wären. Ist nur einer der beiden planar, so wird er ebenso behandelt als wäre er in derselben Klasse wie der andere.

Jetzt aber, ran an die Tasten! Starten wir die Implementierung mit dem ersten Fall damit wir das hinter uns haben. Hier erst mal der Anfang der Funktion zum warm werden.

```
BOOL XBSP::Sichtlinie(XBSP *pBSP,       // BSP Baum des Aufrufers
                      D3DVECTOR vPos_1, // Startpunkt der Linie
                      D3DVECTOR vPos_2, // Endpunkt der Linie
                      XEBENE *pEbene,   // Ebene an der Kollidiert wurde
                      XNODE *pNode) {   // Node bei rekursivem Durchlauf
   D3DVECTOR  vSchnittpkt;
   XNODE     *pTestnode;
   float      fDummy;
   int        nKlasse_Pos_1,
              nKlasse_Pos_2;

   // Node ist die Wurzel falls nicht anders angegeben
   if (pNode == NULL) pTestnode = &pBSP->xWurzel;
   else pTestnode = pNode;

   // Sonst wandere weiter, je nach Position des Spielers
   nKlasse_Pos_1 = this->Klassifiziere_Punkt(pTestnode->xEbene, vPos_1);
   nKlasse_Pos_2 = this->Klassifiziere_Punkt(pTestnode->xEbene, vPos_2);
```

Für den ersten Parameter muss der Aufrufer einen Zeiger auf den BSP Baum angeben der verwendet wird. Der letzte Parameter wird nur beim rekursiven Aufruf durch die Funktion verwendet um den aktuellen Node anzugeben. Der Aufrufer gibt hier `NULL` an. Parameter zwei und drei enthalten den Startpunkt und den Endpunkt des zu testenden Liniensegmentes und im vierten Parameter speichern wir noch die Ebene, an der die Kollision stattfand für den Fall, dass der Aufrufer damit weiterarbeiten möchte. Als erste Operation in der Funktion sehen wir nach, ob ein BSP Baum angegeben wurde und wir mit dessen Wurzel arbeiten müssen. Anderenfalls kamen wir aus einem rekursiven Aufruf der Funktion selber und verwenden den Parameter fünf als Node mit dem wir arbeiten. Und nun geht es richtig lost. Wir klassifizieren die beiden Punkte des Liniensegmentes und gleiten geschmeidig rein in den ersten Fall:

```
   // 1.Fall: Startpunkt in Front && Endpunkt in Back
   if ((nKlasse_Pos_1==BSP_FRONT) && (nKlasse_Pos_2==BSP_BACK)) {
      // Ein Punkt Back, falls Back solide dann keine Sichtlinie
      if (pTestnode->pBack == NULL) {
         if (pEbene != NULL) *pEbene = pNode->xEbene;
         return FALSE;
         }

      // Berechne den Schnittpunkt der Linie mit der Ebene
      xUtil_Schnittpunkt(&vPos_1, &vPos_2, &pTestnode->xEbene.vAuf_Ebene,
                         &pTestnode->xEbene.vNormal, &vSchnittpkt, &fDummy);

      // Teste die beiden Liniensegmente vor bzw. hinter der Ebene getrennt
      // ob sie durch soliden Raum laufen und fasse das Ergebnis zusammen
      return (this->Sichtlinie(NULL, vPos_1, vSchnittpkt, pEbene, pTestnode->pFront)
              &&
              this->Sichtlinie(NULL, vSchnittpkt, vPos_2, pEbene, pTestnode->pBack));
      }
```

Als ersten fragen wir hier mal ab, ob der Backpointer auf soliden Raum zeigt. Ist dies der Fall, dann liegt ein Teil der Sichtlinie (*hier die Seite mit dem Endpunkt des Liniensegmentes*) im soliden Raum und es gibt keine Sichtlinie

zwischen den zwei Punkten. Daher geben wir `FALSE` zurück. Ist dies nicht der Fall, so schneidet das Liniensegment aber dennoch die Ebene an der wir gerade klassifiziert haben. Nun müssen wir das Liniensegment an der Ebene splitten und mit den beiden so entstandenen Teilsegmenten die Funktion rekursiv aufrufen. Dazu berechnen wir den genauen Schnittpunkt durch die Funktion xUtil_Schnittpunkt, die wir aus dem neunten Kapitel bereits kennen.
Unserer ursprüngliches Liniensegment aus Startpunkt und Endpunkt wird nun ersetzt durch die beiden Liniensegmente Startpunkt bis Schnittpunkt und Schnittpunkt bis Endpunkt. Mit diesen beiden neuen Segmenten starten wir zwei rekursive Aufrufe um sie getrennt auf Gültigkeit der Sichtlinie zu prüfen. Die beiden Rückgabewerte der Rekursionen werden dann durch logisches `UND` verknüpft, damit wir auch nur `TRUE` erhalten, wenn beiden Rekursionen das ergeben.

Hoppla, das war's schon **:)** Und weil's so einfach war, machen wir das ganze gleich noch mal. Fall 2 ist nämlich vollkommen analog, wir müssen lediglich beachten, dass das erste Liniensegment (*Startpunkt bis Schnittpunkt*) diesmal im Back der Ebene lag und wir daher rekursiv in den Backnode gehen und mit dem zweiten Segment in den Frontnode. Pfannkuchen...

```
   // 2.Fall: Startpunkt in Back && Endpunkt in Front
   if ((nKlasse_Pos_1==BSP_BACK) && (nKlasse_Pos_2==BSP_FRONT)) {
      // Ein Punkt Back, wenn Back solide dann keine Sichtlinie
      if (pTestnode->pBack == NULL) {
         if (pEbene != NULL) *pEbene = pNode->xEbene;
         return FALSE;
         }

       //  Berechne  den  Schnittpunkt  der  Linie  mit  der  Ebene
      xUtil_Schnittpunkt(&vPos_1,  &vPos_2,  &pTestnode->xEbene.vAuf_Ebene,
                         &pTestnode->xEbene.vNormal,  &vSchnittpkt,  &fDummy);

       //  Teste  die  beiden  Liniensegmente  vor  bzw.  hinter  der  Ebene getrennt
       //  ob  sie  durch  soliden  Raum  laufen  und  fasse  das  Ergebnis  zusammen
       return (this->Sichtlinie(NULL,
vPos_1, vSchnittpkt, pEbene, pTestnode->pBack)
              &&
             this->Sichtlinie(NULL, vSchnittpkt,
vPos_2, pEbene, pTestnode->pFront));
       }
```

Und da wir gerade bei Pfannkuchen sind, hab ich hier noch einen. Wenn beide Punkte in Front sind oder wir planar als in Front annehmen, dann gehen wir rekursiv in den Frontnode. Es sei denn wir sind bereits in einem Leaf, dann gibt es eine gültige Sichtlinie. (*Man beachte, dass dieser Fall natürlich auch mit einem Teilsegment aus einer Rekursion aus dem Fall 1 oder 2 entstanden sein kann und daher nicht auf eine gültige Sichtlinie des ursprünglichen Segments hindeuten muss.*)

```
   // 3.Fall: Beide planar || einer Front anderer planar || beide Front
   if ( ( (nKlasse_Pos_1==BSP_PLANAR) && (nKlasse_Pos_2==BSP_PLANAR) ) ||
        ( (nKlasse_Pos_1==BSP_FRONT)  || (nKlasse_Pos_2==BSP_FRONT)  )   ) {
      // Wenn kein Leaf, dann weitertesten in der Frontliste
      if (pTestnode->blnLeaf == FALSE) {
         return this->Sichtlinie(NULL, vPos_1, vPos_2, pEbene, pTestnode->pFront);
         }
      // Sonst sind wir mit beiden Punkten in einem gültigen Leaf
      else {
         return TRUE; // Es gibt Sichtlinie ohne Unterbrechung
         }
      }
```

Und noch ein Pfannkuchen. Fast analog zum 3. Fall. Sind beide Punkte im Back, beziehungsweise wir interpretieren einen planaren auch als Back, dann können wir `FALSE` zurück geben, wenn der Backpointer auf

soliden Raum zeigt. Anderenfalls gehen wir mit beiden Punkten rekursiv in den Backnode und testen dort weiter.

```
   // 4.Fall: Beide in Back oder einer in Back und anderer planar
   else {
      // Falls Back solide dann keine Sichtlinie ohne Unterbrechung
      // weil mindestens einer beider Punkte im soliden Raum
      if (pTestnode->pBack == NULL) {
         if (pEbene != NULL) *pEbene = pNode->xEbene;
         return FALSE;
         }
      // Falls Backliste existiert weiter den Baum entlang prüfen
      else {
         return this->Sichtlinie(NULL, vPos_1, vPos_2, pEbene, pTestnode->pBack);
         }
      }
   } // Sichtlinie
/*----------------------------------------------------------------------*/
```

Nun haben wir aber genug gegessen, die Funktion ist nämlich schon fertig. Wie wir sehen haben und zwei bereits vorher implementierte Funktionen sämtliche Arbeit abgenommen. Alles was wir noch tun musste war, ein wenig nach zu denken. Das hält nämlich das Hirn fit **:-)**

Kollisionsabfrage der Kamera

Im letzten Schritt dieses Tutorials werden wir noch schnell sehen, wie wir die Funktion zur Sichtlinienprüfung verwenden, um eine Kollisionsabfrage für die Kamera zu implementieren. Logischerweise setzen wir dazu in der Funktion `xKamera_verschieben()` an. Nur wenn die Kamera verschoben wird, kann sie kollidieren. Oder besser andersherum formuliert: Die Kamera darf nur verschoben werden, wenn auf dem Weg der Verschiebung keine Kollision stattfindet. Und das werden wir wie folgt umsetzen.
Bei Start des Programms gehen wir davon aus, dass die Kamera an einer gültigen Stelle in dem Level steht. Der Leveldesigner sollte schlau genug gewesen sein, den Startpunkt im Level so geschickt gesetzt zu haben. Immer wenn die Kamera verschoben werden soll, dann nehmen wir die gewünschte Position, an die die Kamera gehen möchte, und legen noch eine Bounding Sphere um diesen Punkt herum. Damit haben wir auch schon zwei Punkte zwischen denen wir das zu testende Liniensegment ziehen. Startpunkt der Linie ist die gewünschte Position der Kamera, Endpunkt ist die gewünschte Position plus den Radius der Bounding Sphere mal den normalisierten Bewegunsgvektor der Kamera:

```
   vEndpunkt = vStartpunkt + fRadius * vDir;
```

Auf diese Art und Weise testen wir nicht nur den Punkt an dem die Kamera stehen möchte, sondern auch noch ein Stück voraus. Schliesslich nimmt die Kamera auch selbst ein Stück Platz im Raum ein und ist kein unendlich kleiner Punkt. Im Code sieht das wie folgt aus:

```
void xKamera_verschieben(float fX, float fY, float fZ) {
   D3DVECTOR vPos_Neu, vHand, vRichtung;

   vPos_Neu = xKam.vPos;

   // Entlang der x Achse (rechts vs links)
   vPos_Neu.x += xKam.vRight.x * fX;
   vPos_Neu.y += xKam.vRight.y * fX;
   vPos_Neu.z += xKam.vRight.z * fX;

   // Entlang der y Achse (hoch vs runter)
   vPos_Neu.x += xKam.vUp.x * fY;
   vPos_Neu.y += xKam.vUp.y * fY;
   vPos_Neu.z += xKam.vUp.z * fY;
```

```
   // Entlang der z Achse (vor vs zurück)
   vPos_Neu.x += xKam.vDir.x * fZ;
   vPos_Neu.y += xKam.vDir.y * fZ;
   vPos_Neu.z += xKam.vDir.z * fZ;

   // Vektor der Bewegungsrichtung
   vRichtung.x = vPos_Neu.x - xKam.vPos.x;
   vRichtung.y = vPos_Neu.y - xKam.vPos.y;
   vRichtung.z = vPos_Neu.z - xKam.vPos.z;

   // Vektor von 1 Meter Länge
   vRichtung = xUtil_Normalisieren(&vRichtung);

   // Ausgestreckte Hand 0,5m in Bewegungsrichtung
   vHand.x = vPos_Neu.x + 0.5f*vRichtung.x;
   vHand.y = vPos_Neu.y + 0.5f*vRichtung.y;
   vHand.z = vPos_Neu.z + 0.5f*vRichtung.z;

   // Nur wenn Kameraposition inklusive Handreichweite komplett
   // ausserhalb soliden Raums liegen darf die Kam sich bewegen
   if (BSP_Baum.Sichtlinie(&BSP_Baum, vPos_Neu, vHand, NULL, NULL) == TRUE) {
      xKam.vPos = vPos_Neu;
      }
   } // xKamera_verschieben
/*----------------------------------------------------------------*/
```

Anstatt die gewünschte Positionsänderungen gleich in der Kamerastruktur zu speichern verwenden wir die temporäre Variable `vPos_Neu`. Der Vektor `vHand` bezeichnet die Position im Raum, wo die Kamera quasi ihre virtuelle Hand hätte, wenn sie ihren imaginären Arm in Bewegungsrichtung ausstrecken würde. Der Vektor ist damit das Ende des Liniensegmentes auf der Bounding Sphere um die Kamera herum. Zu guter Letzt wird die zwischengespeicherte Position nur für die Kamera zugelassen, wenn die BSP Baum Funktion keine Kollision, also eine freie Sichtlinie, gemeldet hat.

So, damit ist die Kollisionsabfrage in unseren BSP Baum integriert. Zusätzlich haben wir noch eine Funktion mit der wir bequem zwei beliebige Punkte an willkürlichen Positionen in der 3D Welt (*auch im soliden Raum*) daraufhin prüfen können, ob sie einander innerhalb des BSP Baums sehen können. Das reicht wohl für ein Kapitel **:-)**

Aber ich kann mir auch jetzt schon die Fragen vorstellen die kommen werden, also hänge ich mal wieder einen kleinen Theorieteil an dieses Kapitel an um ein paar Ideen zu geben, wie es weitergehen könnte und was zu beachten ist.

Sehen was los ist...

Tja, nun haben wir schon viel viel an unserem BSP herumgeschraubt, da stellt sich mit unter auch die berechtigte Frage danach ein, ob man nicht vielleicht doch mal etwas davon sehen könnte. Wir sehen zwar unsere Levelgeometrie und spüren nun auch die Polygone durch die wir nicht durch können, aber so richtig was sehen tun wir von dem BSP Baum nicht. Ich habe daher in den Code die Variable `g_blnDebug` eingeführt. Wird diese auf `FALSE` gesetzt, so erhalten wir die normale Programmausgabe, also unser 3D Level. Setzen wir sie aber gleich zu Beginn auf `TRUE` (*vor der BSP Baum Erstellung*), dann wird lediglich eine einzige Textur, Index 0 aus dem Array, für alle Polygone verwendet. Dafür aber erhält jedes Leaf eine eigene, zufällige Farbe. Zusätzlich werden die Normalenvektoren aller Polygone angezeigt. SO kann man also genau sehen, welche Polygone zusammen ein Leaf bilden und wo deren Normalenvektoren liegen. Durch die Taste 'n' kann man auch während des laufenden Programms die Variable umschalten, jedoch werden die Farben der Leaves dann nicht mehr geändert. Dies liesse sich auch realisieren, aber das ist ja nicht Sinn der Sache gewesen **:-)**

Levelbau generell und mit AC3D

Modellierungssoftware
Zu aller erst erreicht mich immer die Frage, wie man eigene Level für meine Implementierung bauen kann. Hier kann ich nur auf das Kapitel neun verweisen. Dort habe ich genau angeben, wie das Dateiformat für Polygonmengen eines Levels aufgebaut sein müssen. Ebenfalls im neunten Kapitel haben wir auch einen Loader implementiert, der solche Dateien laden kann. Ich persönlich habe AC3D verwendet, um das Demolevel zu bauen. Allerdings eignet sich jeder beliebige 3D Modeller dazu, so lange man eigene Exporter dafür schreiben kann. AC3D hat beim Levelbau allerdings einen entscheidenden Nachteil. Man kann die Beleuchtung der 3D Ansicht in AC3D nicht deaktivieren. Dadurch erscheinen die Farben der Polygone immer verschieden und man kann das Setzen einer sinnvollen Farbe (*die später die Beleuchtungsintensität in unserer Engine ist*) kaum durchführen. Daher wirkt der Demolevel auch schlecht beleuchtet. Andere 3D Modeller haben diesen Nachteil unter Umständen nicht. Über kurz oder lang wird man aber auch nicht drum herum kommen, einen eigenen Leveleditor zu programmieren.

Illegale Geometrie
Es gibt die sogenannte illegale Geometrie. Dieser begriff bezeichnet ein Phänomen, das man bei einem BSP Baum tunlichst vermeiden sollte. Jedes Polygon in unserem Level hat eine Vorderseite und eine unsichtbare Rückseite. Die Vorderseite wird durch den Normalenvektor der Ebene des Polygons angezeigt. Illegale Geometrie haben wir immer dann vorliegen, wenn eine unsichtbare Rückseite eines Polygons eine sichtbare Vorderseite eines Polygons sehen kann, ohne dass ein *abgeschlossener* Bereich soliden Raums dazwischen liegt. Das beste Beispiel dafür sind die Säulen in dem grossen Raum des Demolevels. Wir hätten es mit illegaler Geometrie zu tun, wenn der Raum eine durchgängige Decke und einen durchgängigen Boden hätte. Die unsichtbaren Rückseiten innen an den Säulen könnten dann nämlich illegalerweise die Frontseiten von Boden und Decke sehen auf denen sie direkt aufliegen.
Derartige Konstruktionen sollten vermieden werden. Statt dessen muss man Boden- und Deckenpolygon in mehrere konvexe Polygone zerteilen, so dass die Aufsetzpunkte der Säulen aus den Polygonen ausgespart werden. Als Beispiel kann man sich den Demolevel in AC3D ansehen. Diese illegale Geometrie führt in dieser Implementrierung lediglich zu dem Problem, dass die Kollisionsabfrage hier nicht greift, weil die Polygone nicht eindeutig zu Front und Back geordnet werden können.

Detaillierte Geometrie
Und noch ein Warnhinweis zum Levelbau. Die Geometrie im BSP Baum sollte möglichst nur die Wände, Decken und Böden des Levels enthalten, sonst aber nichts. Detaillierte Objekte wie Einrichtungsgegenstände (*Tische, Regale, Stühle, Fernseher, Betten, usw.*) sind immer getrennt zu behandeln. Hier kann man jedem Leaf im BSP Baum eine Liste von immobilen und inaktiven Objekte zufügen, die dann in den Baum einsortiert und in diesen Listen gespeichert werden. Idealerweise erhält jedes Leaf nur eine Indexliste in ein separates, globales Objektarray (*meinetwegen auch als Attribut der BSP Klasse* **;-)**). Erstreckt sich ein solches Objekt nämlich über mehrere Leaves, so wird sein Index mehrfach gespeichert. Wenn man ein Leaf rendert, dann aktiviert man in einem Array über die Anzahl dieser Objekte einfach alle Objekte, die in diesem Leaf lagen. Sind alle Leafs gerendert, dann haben wir mit diesem Array eine Liste von Indices in das globale Objektarray. Und genau diese Objekte rendern wir dann auch. Man beachte, dass man hier auch eine Kollisionsfunktion separat integrieren muss.

Animierte Objekte und Monster
Ganz analog gehören animierte Objekte (bewegliche Türen, einstürzende Wände, usw.) nicht in den BSP Baum. Monster noch viel weniger. Analog zu den immobilen und inaktiven Objekten behandeln wir die animierten beziehungsweise diejenigen, die der Spieler aktivieren kann, um eine Reaktion auszulösen (*Schalter, Knöpfe, usw.*). Beim bauen der Levelgeometrie werden Türen usw. also erst mal weg gelassen und später separat integriert. Es sei denn man programmiert sich auch gleich noch seinen eigenen Leveleditor, der diese Unterscheidungen bereits berücksichtigt.

Portale in eine neue Dimension des BSP Baums

Auf in den Krieg! Die Schlacht um den heiligen 3D Indoor Algorithmus tobt schon seit ein paar Jahren. An jeder Ecke liest man (mehr oder minder qualifizierte) Äusserungen, welcher Algorithmus mit welcher Datenstruktur ideal sei für 3D Indoor Visualisierung. Hier gibt es zwei grosse Lager, BSP Bäume und Portal Engines. Es ist ja schon lange kein Geheimnis mehr, dass John Carmack den, bereits lange bekannten, BSP Baum dafür verwendet hat,

mit **Doom** eine neue Generation von Spielen ins Leben zu rufen. Seither wurde diese Technik immer weiter verfeinert und fand in jeder neuen Engine von *id Soft* Anwendung. Das andere Lager scharrt sich um Seth Tellers Ph.D. Thesis. In dieser wurde die Portal Engine Technologie untersucht und beschrieben, mit der man 3D Indoor Architekturen ebenfalls gut darstellen kann.

Und nun ziehe ich meinen Schlachtruf wieder zurück, denn an solchen brotlosen Diskussionen wollen wir uns nicht beteiligen. Welcher Algorithmus besser geeignet ist und welcher nicht, das ist immer zum Teil abhängig von konkreten Projekt und zum anderen gibt es keine weltbewegenden Vorteile des einen die die Vorteile des anderen übertreffen würden.

Nachdem wir nun ein paar Grundbegriffe erklärt haben, wissen wir auch damit umzugehen. Und wenn ich im folgenden etwas über Portale erzähle dann wird das niemand mehr mit einer Portalengine verwechseln. Denn auch wenn wir eine BSP Baum Engine haben, dann können wir trotzdem Portale sinnvoll verwenden. Und daher lege ich hier mal schnell theoretisch dar, was Portale eigentlich sind und wie man diese Portale erzeugen kann. Danach überlegen wir uns ein paar Anwendungen, die wir mit diesen Portalen erzeugen können.
Unser BSP Baum enthält nun alle Ebenen der Polygone an den Nodes, sowie alle Polygone des Levels an den Leaves. Schauen wir uns diese Leaves noch einmal genauer an. Sie enthalten also die Polygone. Damit grenzen sie sich gegen den soliden Raum ab. Aber es gibt durchaus Stellen am Rand des Leafs, wo es keine Polygone und keinen soliden Raum gibt. Das sind nämlich die Übergängen zu einem direkt betretbaren, angrenzenden Leaf. Schauen wir einmal auf die **Abbildung 6** die uns das grafisch zeigt.

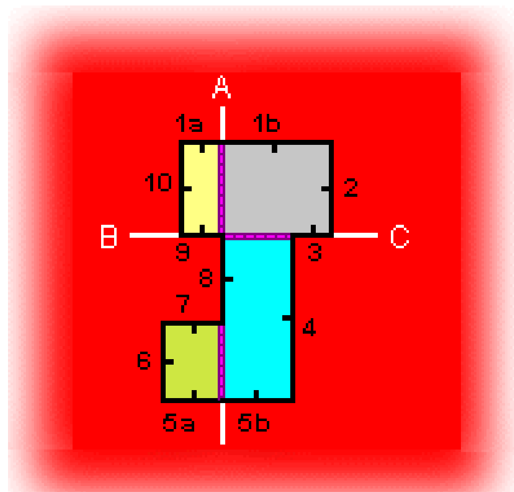


Abbildung 6: Leaves des Demolevels

Die vier Leaves des Levels sind gelb, grau, blau und grün gefärbt. Die Polygone (*schwarze Linien*) begrenzen die Leaves an den Bereichen zum solidem Raum. Betrachten wir uns aber mal die Splitterebenen A, B und C. Diese ziehen nämlich Grenzlinien (*lila-schwarz gestreift*) dort, wo sich zwei Leaves berühren. Wie man sieht ist das nicht zwangsweise dort der Fall, wo enge Durchgänge oder Türen sind, es kann auch mitten in einem Raum sein. Betrachten wir das ganze dreidimensional, dann haben wir keine Grenzlinien mehr. Vielmehr benötigen wir nun Polygone, um die Auftreffflächen zweier Leaves komplett zu füllen. Hätten wir zwei Leaves die sich an einem Durchgang berühren, so wäre das Grenzpolygon genau das, welches den Durchgang wie eine Tür haargenau verschliesst, egal welche Form der Durchgang hat. Und nun Trommelwirbel und Tusch bitte. Dieses Grenzpolygon nennt man ein Portal. Portale sind also einfach Polygone, die alle Auftreffflächen zwischen allen Leaves in unserem Level versiegeln.

Aha...aber warum sollten wir diese Durchgänge versiegeln? Schliesslich sind diese Durchgänge eine valide Möglichkeit für die Objekte in dem Level, um von einem Leaf in ein angrenzendes zu gelangen! Nun, ein Portal ist zwar nichts anderes als ein Polygon. Aber es ist ein unsichtbares Polygon welches wir einfach nicht rendern. Und mit Hilfe dieser Portale können wir beispielsweise berechnen, welche anderen Leaves von einem Portal aus gesehen werden können und das rekursiv durch alle anderen Portale die dieses Portal sehen kann. Damit erhalten wir eine Szene in der wir null Overdraw haben wenn wir die gesamte Geometrie an den Portalen auch noch clippen. Zudem können wir daraus das PVS berechnen. Was das ist, dazu hab ich ja schon in Kapitel neun etwas gesagt.

Erstellung der Portale und PVS Theorie

Und spätestens bei dem Wort PVS bekommen einige jetzt ganz grosse Ohren und verschwitzte Handflächen. Also erklär ich mal kurz die Theorie, wie man die Portale im soliden BSP Baum erzeugt.

Aufgrund der Struktur des soliden BSP Baumes ist es auf alle Fälle so, dass ein Portal immer auf einer Spitter Ebene (*normal oder erzwungen*) liegt. Potentiell kann es also für jede Splitterebene ein Portal geben, das bedeutet für jede Ebene in unserer Levelgeometrie. Wir beginnen die Erstellung der Portale also damit, dass wir eine Liste mit *potentiellen Portalen* erzeugen, und zwar für jeden Node eines. An jedem Node unseres Baumes haben wir ja schliesslich genau eine Ebene der Levelgeometrie.
Ich hatte ja eben schon erwähnt, dass ein Portal einfach ein normales Polygon ist. Doch welche Form und Grösse hat das Portalpolygon? Seine Ausrichtung ist ja die der Ebene, aber welche Abmessungen muss es haben?
Nun, das Portal kann ja maximal so gross werden, wie die Geometrie die unter dem entsprechenden Node noch vorhanden ist. Und welche Abmessung diese Geometrie maximal hat, das verrät uns die Bounding Box die an dem Node gespeichert ist. Wir beginnen also damit, ein Initialportal für jeden Node zu konstruieren, welches entlang der Splitterebene des Nodes ausgerichtet ist und sich so weit in den Raum erstreckt bis es die Ränder der Bounding Box erreicht (*Mittelpunkt der Bounding Box, verschoben entlang des Normalenvektors um die Entfernung des Mittelpunktes zur Ebene ergibt den Mittelpunkt des Polygons, von da aus rechtwinklig zum Normalenvektor Halbe Breite und halbe Höhe der Bounding Box addieren und subtrahieren*). Damit erhält man ein Viereck als Initialportal, welches sich über die gesamte Subgeometrie erstreckt.

Okay, wir haben jetzt eine lange Liste mit vielen Initialportalen. Und zwar genau so viele, wie wir Polygone in unserem BSP Baum haben. Doch diese Initialportale haben wenig Nutzen, da sie ja noch nicht genau die Auftreffflächen zweier Leaves füllen, sondern zum einen riesengross sind und zum anderen auch an Stellen liegen, wo es gar keine Portale (*keine Aufftrefffflächen*) gibt. Wir nehmen nun also *jedes* Portal aus der Liste und werfen es wieder beginnend bei der Wurzel in den BSP Baum. An jedem Node wird das potentielle Portal klassifiziert (*wie ein normales Polygon, das haben wir ja schon mal gemacht*).

Liegt es komplett im Backbereich der Splitterebene, dann schicken wir es in den Backnode dieses Nodes. Sollte der `NULL` sein, dann liegt das potentielle Portal...genau, im soliden Raum und ist damit kein Portal und wird aus der Liste gelöscht.

Liegt es komplett im Frontbereich der Splittereben, dann schicken wir es in den Frontnode dieses Nodes. Sollte dieser ein Leaf sein, dann ist das potentielle Portal ein gültiges Portal und bleibt in der Liste erhalten. Zudem speichern wir eine ID in ihm, die eindeutig das Leaf angibt zu dem dieses Portal gehört.

Und nun zu den beiden interessanten Fällen. Spannt sich das potentielle Portal quer über die Ebene (*zum Teil Front und zum Teil Back*), dann splitten wir das potentielle Portalpolygon in zwei potentielle Portalpolygone, ein Frontpolygon und ein Backpolygon. Ganz analog wie wir es bei der BSP Baumerstellung auch schon gemacht haben. Die entsprechenden beiden Teilpolygone schicken wir dann in den jeweiligen Node, Front beziehungsweise Back.

Etwas verwirrend erscheint zunächst der Fall wenn das potentielle Portal genau auf der Ebene liegt. Dann schicken wir das potentielle Portal in einen beliebigen Node, Front oder Back, und lassen es dort beschneiden, klassifizieren und splitten. Dadurch erhalten wir eine ganze Liste mit potentiellen Portalen. Jedes potentielle Portal dieser Liste schicken wir dann in den Node, den wir oben nicht genommen haben. Alle Fragmente des ursprünglichen potentiellen Portals, welche auch diese Tortour überleben, sind gültige Portal. Die anderen Fragmente werden aus der Liste gelöscht und vergessen **:-)**

Okay...das war's. Damit haben wir nach einer ganzen Reihen von Durchläufen nachher eine Liste mit gültigen Portalen erhalten. Halt, so ganz stimmt das noch nicht. Ein Portal muss noch eine weitere Eigenschaft erfüllen, damit es wirklich ein gültiges Portal ist. Wenn zwei Leaves an einer Stelle im gültigen Raum ohne Wand dazwischen aufeinandertreffen, so gibt es an dieser Stelle ein Portal, welches die IDs von genau zwei Leaves gespeichert haben muss. Ein Portal gehört immer zu genau zwei Leaves. Wenn ein potentielles Portal genau auf einer Ebene liegt, dann wird es ja sowohl in den Frontnode als auch den Backnode geschickt. Dabei kann es dann eben zwei IDs von zwei verschiedenen Leaves erhalten und damit zu einem gültigen Portal werden. Gültige Portale liegen nämlich genau auf (*regulären, nicht erzwungenen*) Splitterebenen und sind damit sowohl in deren Frontnodes als auch deren Backnodes gültig.

Puh...ist das alles einfach heute **:-)** Nun noch auf ein Wort zur Berechnung des PVS. Hier nimmt man einfach jedes Leaf aus dem BSP Baum und die dazu gehörenden Portale. Jetzt kann man durch Frustrum Culling durch die einzelnen Portale bestimmen, was man durch dieses Portal sehen kann. Hierzu sollte man mal die Ph.D. Thesis von Seth Teller im Internet suchen. Das Portal Clipping wird darin auch behandelt.

Jetzt ist es vorbei

Alles hat ein Ende nur die Wurst hat zwei. Schön, dass das Radio letztes Jahr endlich mit der Tradition gebrochen hat, dieses Lied an Sylvester bis zum Erbrechen zu spielen. Das hab ich schon immer wie die Pest gehasst. Aber dennoch ist es die traurige Wahrheit, denn auch dieses Kapitel ist nun zu Ende. Ich hoffe, ich habe es wieder geschafft, ein wenig Licht in das Dunkel um die Spieleprogrammierung zu bringen. Und ich verweise auch gerne noch mal auf das Projekt Pandoras Box welches als Beispiel für den dritten Band implementiert wird **:-)** Aber bis dahin ziehen noch einige Monate ins Land...

Damit haben wir nun auch das endgültige Ende dieses Tutorials erreicht. Für ein Online Tutorial ist es um einiges umfangreicher geworden, als ich es zunächst geplant hatte. Aber es macht einfach unheimlich viel Spass, wenn so viel Feedback zur eigenen Arbeit kommt. Ich denke ich habe Euch hier genug Material an die Hand gegeben, um im Bereich 3D Spieleprogrammierung von Indoor Games auf eigenen Füssen stehen zu können. Am Anfang zwar noch etwas wacklig, aber mit genug Potential schnell das Gehen lernen zu können...

Stefan Zerbst, Braunschweig 9. Januar 2002

Und hier gibt es den Code des ***gesamten*** Tutorials zum Download: Projektdateien

ACHTUNG: Leider gab es bei AC3D ein paar Probleme mit der Beleuchtung, daher mag der Level nicht sehr schön beleuchtet erscheinen. In AC3D wird eine Punktlichtquelle verwendet die alle Geometrie abhängig von der Rotation der Objekte in der Welt berechnet. Daher kann man den korrekten Farbwert der Polygone nie so erkennen wie er später in diesem Code hier erscheinen wird.

It's over

Woah...jetzt ist auch dieses Megatutorial an seinem Ende. Und sagt schön Danke! Nein, Quatsch. Das ist natürlich keine Voraussetzung, aber ich freue mich immer darüber wenn man mir per eMail Anregungen, Kommentare, Kritik und natürlich auch Lob zukommen lässt. Mich interessiert natürlich schon ob das hier überhaupt jemand liest und ob es für Euch von Interesse und hilfreich war, schliesslich habe ich allein in die beiden BSP Kapitel vier Wochen Arbeit gesteckt und drei Monate für das ganze Tutorial, wenn auch mit den letzten beiden Kapiteln über einen Zeitraum von einem Jahr verteilt. Für alle die es noch nicht wissen: Meine eMail Adresse ist ceo@stefanzerbst.de

She clings to me like cellophane
fake plastic submarine
So what if the sex was great
Just a temporary escape
Another thing I grew to hate
But now that's over
Why you always kick me when I'm high
Knock me down till we see eye to eye
I used to hang on every word
each lie was more absurd
Kept me so insecure
She taught me how to trust
And to believe in us
And then she taught me how to cuss
...that bitch
It's over!!!
SR-71, Right Now

Dies ist das bittere Ende!!! ....*vorerst???*