Entwicklung einer interaktiven Beispielapplikation auf Basis von DirectX

Studienarbeit

Vorgelegt von

Irene Schindler
Mat.-Nr.: 200210216

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer und Prüfer:
Prof. Dr.-Ing. Stefan Müller

Februar 2005

Inhaltsverzeichnis

1 Einleitung **1**

2 DirectX 9 **2**

2.1 DirectX-Komponenten 3
2.2 Das COM – Grundlage von DirectX 4
2.3 Direct3D - Grundlagen 4
2.3.1 Transformationspipeline 5
2.3.2 Rasterizer 5
2.3.3 Schnittstellen 6
2.3.4 Bibliotheksdateien 6
2.4 Initialisierung von Direct3D 6
2.5 DirectSound - Grundlagen 8
2.5.1 Schnittstellen 9
2.5.2 Bibliotheksdateien 9
2.6 Initialisierung von DirectSound 9

3 Maya 5.0 **10**

3.1 Maya Complete 11
3.2 Maya Unlimited 12
3.3 Grundlagen der Modellierung 13
3.4 Export von Modelldateien 14

4 Konzept-Entwicklung **15**

4.1 Das 3D-Modell 17

5 Implementierung der Grafikanwendung **18**

5.1 Initialisierung der Szene 19
5.1.1 Render-States 19
5.1.2 Vertexformat 20
5.1.3 Sampler-States 20
5.1.4 Texturen laden 21
5.1.5 Umgebungstexturen laden 21
5.1.6 Volumentexturen laden 22
5.1.7 Modell aus einer X-Datei laden 22
5.2 Move-Funktion 23
5.3 Render-Funktion 25
5.3.1 Sichtmatrix und Projektionsmatrix setzen 26
5.3.2 Sky-Box zeichnen 27
5.3.3 Beleuchtung 28
5.3.4 3D-Modell zeichnen 29
5.3.5 Alpha-Blending 30
5.3.6 Multi-Texturing 31

5.3.7 Volumentexturen zeichnen 33
5.3.8 Nebel 34
5.4 Sound 34
5.5 Ergebnisbilder 35

6 Ausblick 37

1 Einleitung

Die Grafikprogrammierung kommt heutzutage in vielen Bereichen zur Anwendung, wie z.B. im Maschinenbau, in der Architektur und natürlich auch in Computerspielen.

Beim Programmieren von Grafikanwendungen, in denen Daten von 3D-Objekten mittels Hardware verarbeitet werden, muss man sich für eine API (Application Programming Interface), eine Schnittstelle für die Erstellung von Anwendungen, entscheiden.

Die beiden am meisten verbreiteten APIs für die Grafikprogrammierung sind OpenGL und DirectX. OpenGL ist für viele Betriebssysteme, wie MacOS und Linux, vorhanden. Aber die meisten Spiele werden immer noch für die Windows-Plattform entwickelt. Und in diesem Bereich hat sich DirectX eindeutig als die wichtigste Schnittstelle etabliert.

Der Vorteil von DirectX liegt darin, dass viele Techniken bereits in der API als Funktionen und Datenstrukturen vorhanden sind und nicht erst selbst implementiert werden müssen.

DirectX besteht aus verschiedenen Komponenten, wie Direct3D, DirectInput, DirectSound, DirectPlay u.a.. Jede Komponente behandelt unterschiedliche Aspekte der Spiele-Programmierung und richtet sich dadurch auch an unterschiedliche Hardware. Weiterhin gibt es diverse File-Loader, um 3D-Modelle, die mit einem 3D-Modeling Tool erstellt wurden, mit der Anwendung lesen und anzeigen zu können. Die meisten der heute auf dem Markt erscheinenden PC-Spiele werden in C++ geschrieben und benutzen DirectX.

OpenGL ist in C geschrieben und bietet daher kein komfortables Klassensystem, um koplexe Szenen zu handhaben. Vom Umfang her ist es auch sehr viel kleiner als die äquivalente DirectX-Komponente Direct3D, die für die 3D-Grafik zuständig ist. Deshalb wird OpenGL nur teilweise bei der Spieleentwicklung benutzt. Hauptsächlich wird es in professionellen Anwendungen wie CAD oder in der 3D-Datenvisualisierung verwendet, da es die Möglichkeiten der Hardware zur Performanceoptimierung besser ausnutzt.

3D-Modelle für Grafikanwendungen können mit sogenannten ”modeling packages” erstellt werden. Maya von AliasWavefront zählt hier zu den wohl umfangreichsten und vollständigsten Programmen. Ein weiterer Vorteil ist, dass es neben der Kaufversion von Maya, eine ”Maya Personal Learning Edition” gibt, die man frei downloaden kann. Eine Alternative zu Maya ist z.B. das Programm 3DStudio Max, welches einen ähnlich großen Funktionsumfang liefert. Allerdings gibt es für 3DStudio Max keine Version zum freien Download.

Für diese Studienarbeit fiel die Wahl der API auf DirectX und im speziellen auf die Komponenten Direct3D, eine umfangreiche 3D-Engine, und auf DirectSound, das den Zugriff auf alle Soundfunktionen ermöglicht. Als 3D-Modeling

Tool wurde Maya gewählt.

Am Anfang der Studienarbeit stand die Einarbeitung in Maya sowie in DirectX und seine Komponenten. Anschließend wurde ein Konzept für die Beispielapplikation entwickelt und die 3D-Modelle in Maya erstellt. Mittels eines File-Loaders wurden die Modelle in DirectX geladen, wo schließlich die endgültige Grafikanwendung entwickelt wurde.

2 DirectX 9

DirectX besteht aus vielen einzelnen unabhängigen Komponenten, von denen jede für sich einen gewissen Funktionsbereich abdeckt (Grafik, Sound, Eingabe, Netzwerk, ...).

Hauptsächlich wurde DirectX für die Programmierung von anspruchsvollen Spielen entworfen, und es bietet den heutigen Spieleprogrammierern alles, was sie wollen:

- schnelle 3D-Grafik
- klaren 3D-Sound
- 3D-Musik
- Ansteuerung aller Arten von Eingabegeräten: Tastatur, Maus, Joystick, Game-Pad, Lenkrad, Gaspedal, usw.
- Netzwerkunterstützung
- Medienwiedergabe: MP3-Dateien, Videos uvm.

Die einzelnen Komponenten von DirectX sind:

1. DirectX Graphics
2. DirectX Audio
3. DirectInput
4. DirectPlay
5. DirectShow
6. DirectSetup

Um mit DirectX überhaupt erst arbeiten zu können, stellt Microsoft das **DirectX-SDK** zur Verfügung, das kostenlos von der Microsoft-Website heruntergeladen werden kann. Das SDK (Software Development Kit) ist sozusagen der Vermittler zwischen der jeweiligen Programmiersprache und DirectX, indem es, neben den vielen Beispielprogrammen und einer umfangreichen Dokumentation, die nötigen Header- und Bibliotheksdateien von DirectX liefert.

2.1 DirectX-Komponenten

DirectX Graphics

Früher gab es zwei Komponenten:

- **Direct3D** für 3D Grafiken und
- **DirectDraw** für 2D Grafiken

Ab DirectX 8 wurden die beiden Komponenten zu **DirectX Graphics** vereint.

Weiterhin gibt es eine hilfreiche Zusatzbibliothek, die viele nützliche Funktionen zur Arbeit mit **Direct3D** bietet, wie z.B. mathematische Hilfestellungen u.v.m.. Der Name der Zusatzbibliothek ist **D3DX**. Ihr Nachteil ist allerdings, dass sie die ausführbare EXE-Datei vergrößert, wenn sie verwendet wird.

DirectX Audio

DirectX Audio ist für den Sound zuständig und besteht aus:

- **DirectSound** und
- **DirectMusic**

DirectSound bietet sowohl Soundeffekte wie z.B. eine Explosion, als auch interaktive Hintergrundmusik, die sich je nach Spielsituation in Laufzeit verändern lässt.

Ein weiteres Feature ist unter anderem der 3D-Sound, bei dem z.B. ein virtueller Krankenwagen auf den User zufährt und das Sirenengeräusch daraufhin so verzerrt wird, wie das auch in Wirklichkeit der Fall wäre (Dopplereffekt).

Wenn man die richtige Ausrüstung besitzt, können Sounds sogar räumlich abgespielt werden (vorne, hinten, rechts, links).

Auch das Aufnehmen von Sounds per Mikrofon zählt zu den Funktionen von **DirectX Audio**. Damit lassen sich z.B. richtige Unterhaltungen während eines Netzwerkspiels führen oder auch Sprachkommandos bewerkstelligen.

DirectInput

DirectInput ist für die Eingabe zuständig. Zur Liste der Fähigkeiten zählt sogar die Unterstützung von Force-Feedback. Dadurch kann der Joystick eigene Bewegungen durchführen wie z.B. Rütteln oder Gegenlenkung, wenn er es unterstützt.

DirectPlay

DirectPlay wurde entworfen, damit Spiele auch im Internet, im Netzwerk, per Kabelverbindung oder per Telefonverbindung gespielt werden können, denn es macht es relativ einfach, eine Multi-Player-Funktion in ein Spiel einzubauen.

Ein großer Vorteil ist, dass dabei für alle Möglichkeiten genau dieselbe Funktion benutzt werden kann, da **DirectPlay** keinen Unterschied zwischen einem Internetspiel und einem Netzwerkspiel kennt.

DirectShow

Durch diese Komponente ist man in der Lage, alle möglichen Typen von Multimedia-Dateien abzuspielen. **DirectShow** ist auch für die Wiedergabe von DVDs und für die Steuerung von Aufnahmegeräten (Videokameras) zuständig.

DirectSetup

Mit Hilfe von **DirectSetup** kann ein Installationsprogramm für DirectX in ein Spiel eingebaut werden. Ein Spiel kann dann testen, ob die geforderte Version von DirectX auf dem Computer vorhanden ist, und bei Bedarf die aktuelle Version aufspielen.

2.2 Das COM – Grundlage von DirectX

Das COM (Component Object Model = Komponenten-Objektmodell) ist ein Modell, das den Umgang und die Verwaltung von Objekten, die Teil einer großen Funktionssammlung – wie DirectX – sind, durch eine übersichtliche Organisation vereinfacht. DirectX baut auf dem COM auf, das der Objektorientiertheit von C++ entgegenkommt.

Das COM fordert die Auteilung eines Programms in verschiedene Schnittstellen (Interfaces), so dass jede Schnittstelle jeweils einen Aufgabenbereich abdeckt. So gibt es z.B. eine Schnittstelle für die 3D-Grafik und eine andere Schnittstelle für den Sound. Wurde eine Schnittstelle für einen bestimmten Aufgabenbereich erstellt, stellt sie wiederum Funktionen bereit, die es ermgöglichen in diesem Aufgabenbereich zu arbeiten.

Eine Schnittstelle muss durch eine bestimmte Funktion erzeugt werden, die für jeden Schnittstellentyp anders lautet. Die Namen der Schnittstellen tragen immer das Präfix ”I” für ”Interface”. Der eigentliche Name der Schnittstelle folgt direkt danach, wie z.B. bei *IDirect*3*D*9 – Schnittstelle für die Verwaltung von **Direct3D**.

2.3 Direct3D - Grundlagen

Direct3D ist die DirectX-Komponente, die sich ausschließlich mit der Darstellung von 3D-Grafik beschäftigt und hierzu auch eine standardisierte Befehlsbibliothek liefert, die bei jeder Hardware das Gleiche bewirkt.

Direct3D ist sozusagen der Vermittler zwischen dem Programm und dem Treiber der Grafikkarte. Es nimmt die Befehle an, verarbeitet sie und leitet Teile davon weiter. Vereinfacht gesagt, gibt man **Direct3D** ein paar Vektoren, Farben, Matrizen, Texturen, Lichter und Ähnliches, und das alles wird dann so zu einem Ganzen

zusammengesetzt, dass ein Bild dabei herauskommt.

2.3.1 Transformationspipeline

Die Transformation von 3D-Objekten und Vektoren übernimmt **Direct3D**. Dafür gibt es drei frei definierbare Transformationsmatrizen, die zu Beginn alle auf die Identitätsmatrix gesetzt sind und dann beim Zeichnen (Rendern) hintereinander auf alle Vektoren angewandt werden.

Die Weltmatrix führt die Welttransformation durch, die die erste Station in der Transformationspipeline ist. Hier werden Verschiebungen, Rotationen und Skalierungen der 3D-Objekte vorgenommen.

Die Sichtmatrix führt die Sichttransformation durch. An dieser Stelle kommt die Kamera ins Spiel, so dass die Vektoren, die durch diesen Schritt in der Transformationspipeline erzeugt werden, bereits relativ zum Beobachter sind. Die Sichttransformation verwandelt also absolute Positionsangaben in Koordinaten relativ zu einer virtuellen Kamera. Wenn keine Kamera benötigt wird, kann hier auch die Identitätsmatrix gesetzt werden.

Die Projektionsmatrix sorgt für den 3D-Effekt. Will man aber z.B. 2D-Grafik erzeugen, dann setzt man hier einfach die Identiätsmatrix. Die durch die Projektion erzeugten Vektoren stellen die endgültige Position auf dem Bildschirm dar, wobei der Punkt (0, 0) der Bildschirmmittelpunkt ist.

Nach der Projektion werden alle Dreiecke geclippt, also abgeschnitten, wenn sie ganz oder teilweise außerhalb des Sichtbereichs des Beobachters liegen.

2.3.2 Rasterizer

Der Rasterizer stellt den Kern von **Direct3D** dar, denn letztendlich sorgt er dafür, dass man 3D-Grafik auf dem Bildschirm auch wirklich sehen kann. Er ist meistens in der Hardware (also in der 3D-Grafikkarte) implementiert.

Der Rasterizer teilt die Vektoren der zu zeichnenden Punkte, Linien und Dreiecke in das Bildschirmraster ein. Jedem Vektor wird so eine 2D-Pixelkoordinate zugeordnet. Je höher die Auflösung ist, desto exakter können die Vektoren in das Raster eingeteilt werden. Nachdem die endgültigen Bildschirmkoordinaten festgelegt sind, übernimmt der Rasterizer das Zeichnen der Primitiven, das Rendern.

Diese Hardwarebeschleunigung (Hardware Acceleration) ermöglicht es, dass ein Programm schon einmal die Berechnungen (Objekte traslatieren, Kollisionen berechnen usw.) für das nächste Bild durchführen kann, während die Hardware im Hintergrund am aktuellen Bild arbeitet (eine gewisse Parallelität, wobei die Aufgaben auf Hardware und Software aufgeteilt werden).

Die Art und Weise, wie **Direct3D** Primitive rendern soll, wird mit den Render-States festgelegt. Sie steuern den Raserizer und kontrollieren so verschiedene Ef-

fekte und Einstellungen, wie z.B. Nebel, Beleuchtung, Alpha-Blending oder Z-Buffering.

Jedem Render-State kann ein bestimmter Wert zugewiesen werden (meistens ist es einfach nur ”true” oder ”false” - je nachdem, ob das gewünschte Feature aktiviert sein soll oder nicht). Im Prinzip sind alle Render-States nichts anderes als eine einfache Konstante und sie beginnen alle mit ”D3DRS_”.

2.3.3 Schnittstellen

Die beiden wichtigsten Schnittstellen von **Direct3D** sind

- *IDirect3D9* und
- *IDirect3DDevice9*.

IDirect3D9 ist eine COM-Schnittstelle und für die Verwaltung von **Direct3D** zuständig. *IDirect3D9* dient der Auflistung aller auf dem System verfügbarer Adapter (Grafikkarten oder Grafikchips). Außerdem liefert sie Informationen über die Fähigkeiten der Adapter, wie z.B. die unterstützten Videomodi, die Fähigkeiten und den Namen des Geräts, die Treiberversion usw..

Eine Methode der *IDirect3D9*-Schnittstelle erzeugt die Geräteschnittstelle *IDirect3DDevice9*, die einen speziellen Adapter repräsentiert. Die Befehle zum Zeichnen von 3D-Grafiken werden von ihr angenommen.

2.3.4 Bibliotheksdateien

Um auf die Funktionen und Schnittstellen von **Direct3D** zugreifen zu können, muss man mindestens zwei Dateien zum Programm hinzufügen: Die Datei **D3D9.h** wird per *#include* eingebunden, und die beiden Dateien **D3D9.lib** und **DXErr9.lib** werden in die Liste der Bibliotheksdateien eingetragen. Ab da sind dem Compiler alle **Direct3D**-Symbole bekannt, so dass man sie im Programm verwenden kann.

2.4 Initialisierung von Direct3D

Am Anfang des Programms wird **Direct3D** initialisiert und am Ende wieder heruntergefahren.

Der erste Schritt bei der Initialisierung ist das Anlegen der *IDirect3D9*-Schnittstelle, der Verwaltungsschnittstelle von **Direct3D**, die Informationen über die auf dem System verfügbaren Adapter liefert. Hierzu wird die Funktion *Direct3DCreate* aufgerufen, die als einzigen Parameter die Versionsnummer der zu erstellenden Schnittstelle erwartet. An dieser Stelle wird immer das Makro D3D_SDK_VERSION angegeben. Es ist in der Direct3D-Headerdatei deklariert und hält immer die Version des installierten DirectX-SDK bereit.

Beispiel:

```
//globale Variable für die IDirect3D9-Schnittstelle
PDIRECT3D9 g_pD3D = NULL;

//IDirect3D9-Schnittstelle erzeugen
g_pD3D = Direct3DCreate9(D3D_SDK_VERSION);

if(g_pD3D == NULL)
{
    //beim Erstellen der Schnittstelle ist ein Fehler aufgetreten!
}

//Schnittstelle wieder freigeben
g_pD3D->Release();
```

Anmerkung: PDIRECT3D9 ist die Zeigervariante der Schnittstelle *IDirect3D9*

Als nächstes muss ein Fenster erzeugt werden, in dem sich alles abspielen wird. Dies muss auf jeden Fall geschehen, auch wenn das Programm den Vollbildmodus verwendet. Im Fenstermodus sollte die Fenstergröße der Größe des Bildpuffers entsprechen, um eventuelle Verzerrungen zu vermeiden. Die Fenster-Generierung sollte in einer Funktion erfolgen, die eine neue Fensterklasse erstellt, sie registriert und dann ein Fenster dieser Klasse erzeugt. Zusätzlich benötigt man eine Nachrichtenfunktion des Fensters. Das Fenster-Handle wird in einer globalen Variablen gespeichert.

Der nächste Schritt ist die Erzeugung der *IDirect3DDevice9*-Schnittstelle. Dazu muss die Methode *IDirect3D9 :: CreateDevice* aufgerufen werden, die sechs Parameter erwartet. Diese bestimmen u.a. die Grafikkarte, für die die Schnittstelle erzeugt werden soll, den Modus, in dem die Grafikkarte laufen soll (Hardware- oder Softwaremodus) oder das Handle des Fensters, in dem die 3D-Grafik dargestellt werden soll.

An die erstellte *IDirect3DDevice9*-Schnittstelle können nun sämtliche Befehle zum Zeichnen von Grafiken im Anwendungsfenster geschickt werden.

Eine gute Überprüfung welche Fähigkeiten eine Grafikkarte unterstützt ist sehr wichtig für jede **Direct3D**-Initialisierungsfunktion. Denn das Spiel bzw. das Programm muss mit der Grafikkarte zurechtkommen und nicht umgekehrt. Deshalb sollte die Erstellung der Geräteschnittstelle *IDirect3DDevice9* nicht ohne eine vorherige Auswahl aller wichtigen Parameter erfolgen. Aus diesem Grund enthält das DirectX-SDK das Tool "DirectX Caps Viewer". Mit Hilfe dieses Tools kann man die Fähigkeiten (Caps) jeder DirectX-Komponente und jedes auf dem System verfügbaren Gerätes einsehen.

Am Ende des Programms muss **Direct3D** wieder heruntergefahren werden.

Die beiden Schnittstellen *IDirect3D9* und *IDirect3DDevice9* werden in zwei globalen Variablen gespeichert. Zum Herunterfahren wird einfach auf beiden Schnittstellen die *Release*-Methode aufgerufen, wobei die Reihenfolge dabei egal ist. Das Freigeben einer Schnittstelle durch die *Release*-Methode ist sehr wichtig, denn eine nicht freigegebene Schnittstelle wird mit großer Wahrscheinlichkeit in einem Speicherleck enden.

2.5 DirectSound - Grundlagen

DirectSound unterstützt folgende Features:

- Mischen einer unbegrenzten Anzahl von Sounds, und das recht schnell.
- Wenn beschleunigende Hardware vorhanden ist, wird diese automatisch genutzt (Sounds können eventuell sogar direkt in der Soundkarte abgelegt werden oder die Hardware mischt die Sounds).
- 3D-Sound: Sounds können frei im Raum positioniert werden, und **DirectSound** berechnet automatisch, wie sie den Hörer erreichen. Dabei kann auch der Dopplereffekt (Änderung der Höhe eines Sounds bei sich bewegender Schallquelle) simuliert werden.
- Man kann verschiedene Effekte, deren Parameter sogar in Echtzeit verändert werden können, auf Sounds anwenden.
- Sounds können sogar zur gleichen Zeit aufgenommen und auch abgespielt werden.

DirectSound organisiert alle Sounds in Soundpuffern, die man sich wie fast beliebig große Speicherbereiche vorstellen kann, die den Sound, neben Formatbeschreibungen, in Wellenform beinhalten.

Allerdings ist der primäre Soundpuffer der einzige Soundpuffer, der wirklich abgespielt werden kann. Er wird durch die Soundkarte und die Lautsprecher wiedergegeben.

In den sekundären Soundpuffern werden die eigentlichen Sounds gespeichert. Wann immer ein sekundärer Soundpuffer abgespielt wird, schickt er seine Daten zum Mixer. Dieser mischt alle zurzeit spielenden sekundären Soundpuffer, indem er aus den vielen Sounds einen einzigen macht und anschließend schreibt er das Ergebnis in den primären Soundpuffer, der dann schließlich abgespielt wird. Eigentlich ist also der Mixer dafür verantwortlich, dass man mehrere verschiedene Sounds gleichzeitig hören kann.

Alle Stationen, die ein Sound durchläuft (Mixer, 3D-Sound, Effekte), können entweder per Software oder per Hardware durchgeführt werden - ähnlich wie bei **Direct3D**.

Auch kann ein sekundärer Soundpuffer entweder im Systemspeicher oder im Hardwarespeicher abgelegt werden. Wenn die Hardware es unterstützt, dann sollte hierzu der Hardwarespeicher gewählt werden. Dieser kann nämlich viel schneller abgespielt werden, da sich der Hauptprozessor nicht darum kümmern muss.

2.5.1 Schnittstellen

Im Wesentlichen benötigt man beim Sound nur zwei Schnittstellen:

- $IDirectSound8$ und
- $IDirectSoundBuffer8$.

Das Ausgabegerät wird hier durch die $IDirectSound8$-Schnittstelle repräsentiert, da es in **DirectSound** keine wirkliche Geräteschnittstelle wie bei **Direct3D** gibt. Durch eine Methode dieser Schnittstelle werden dann alle Soundpuffer erstellt.

Die Schnittstelle für alle Soundpuffer ist $IDirectSoundBuffer8$. Sie bietet Methoden zum

- Abspielen
- Stoppen
- und für verschiedene Effekte.

2.5.2 Bibliotheksdateien

Zuerst ist es notwendig die Header-Datei **DSound.h** in das Programm einzubinden. Anschließend müssen noch die Bibliotheksdateien **DSound.lib** und **DXGUID.lib** zum Projekt hinzugefügt werden.

2.6 Initialisierung von DirectSound

Eine $IDirectSound8$-Schnittstelle wird durch die Funktion $DirectSoundCreate8()$ erzeugt, die drei Parameter erwartet. Der erste Parameter bestimmt das Gerät, auf dem der Sound wiedergegeben werden soll. Wenn das Standardwiedergabegerät von Windows verwednet werden soll, wird für den ersten Parameter einfach NULL angegeben. Der zweite Parameter gibt an, welcher Variable die $IDirectSound8$-Schnittstelle zugewiesen werden soll. Der dritte Parameter wird nicht verwendet und ist deshalb immer NULL. Nach der Erzeugung der $IDirectSound8$-Schnittstelle ist es nun möglich Soundpuffer herzustellen.

Vorher muss allerdings mit $IDirectSound8 :: SetCooperativeLevel()$ die Kooperationsebene von **DirectSound** gesetzt werden. Die Kooperationsebene bestimmt die Art und Weise, wie eine Anwendung mit anderen Anwendungen zusammenarbeitet. Ob sie z.B. den Zugriff auf ein Gerät für sich allein beansprucht

oder nicht. Im Fall von **DirectSound** entscheidet die gesetzte Kooperationsebene, ob das Programm, das **DirectSound** verwendet, das Audioformat des primären Soundpuffers setzen darf oder nicht. Dieses Format bestimmt nämlich die Soundqualität.

Der erste Parameter von *SetCooperativeLevel()* ist ein Fenster-Handle, das Fenster der Anwendung. Der zweite Parameter gibt die Kooperationsebene an. Hierfür gibt es zwei Möglichkeiten:

1. DSSCL_PRIORITY: Die Anwendung darf das Format des primären Soundpuffers bestimmen und hat höchste Kontrolle über die Hardware und ihre Ressourcen. Dies ist für Spiele geeignet.

2. DSSCL_NORMAL: Normaler Modus. Das Format des primären Soundpuffers ist festgelegt (8 Bits). Hier wird keine Anwendung bevorzugt.

Nach dem Setzen der Kooperationsebene können sowohl der primäre Soundpuffer, als auch die sekundären Soundpuffer mit der Methode *IDirectSound8 :: CreateSoundBuffer()* erstellt werden. Die einzelnen Parameter dieser Methode legen die verschiedenen Eigenschaften und Fähigkeiten des zu erzeugenden Soundpuffers fest. Als Ergebnis liefert die Methode eine *IDirectSoundBuffer*-Schnittstelle, die eine frühere Schnittstellenversion ist und dementsprechend noch nicht so viele Fähigkeiten besitzt. Sie ist aber die Schnittstelle für den primären Soundpuffer.

Um die neuste Schnittstellenversion anzufordern, die für die sekundären Soundpuffer verwendet werden kann, muss man zusätzlich die COM-Methode *QueryInterface* aufrufen und ihr den Schnittstellenbezeichner *IID_IDirectSoundBuffer8* übergeben. Sie liefert dann endlich die *IDirectSoundBuffer8*-Schnittstelle, die die neuste Version ist. Es gibt bei DirectX 9 keine *IDirectSoundBuffer9*-Schnittstelle, da es seit DirectX 8 keine nennenswerten Änderungen in diesem Bereich gegeben hat. Die alte Schnittstelle *IDirectSoundBuffer* muss man noch mit *Release()* wieder freigeben, da sie nun nicht mehr benötigt wird.

Die erstellten Soundpuffer können nun theoretisch mit den gewünschten Sounds gefüllt und abgespielt werden.

3 Maya 5.0

In erster Linie ist Maya ein Programm zur Erzeugung dreidimensionaler Grafiken in Bewegung. Es wurde auch bereits für viele bekannte Kinohits, wie z.B. "Star Wars" oder "Das große Krabbeln" eingesetzt.

Maya ist in zwei Versionen unterteilt:

- **Maya Complete** und
- **Maya Unlimited**.

3.1 Maya Complete

Maya Complete ist die Basis-Software mit allen grundlegenden Werkzeugen und besteht aus den Modulen

- Modeling
- Animation
- Artisan
- Dynamics
- Rendering und
- Paint Effects.

Basis-Modul

Im Basis-Modul befindet sich die Oberfläche des Programms mit seinen Bearbeitungsfenstern sowie eine Schnittstelle zur Integration von Plug-Ins für weitere Module des Herstellers oder Drittanbieter.

Modeling-Modul

Im Modeling-Modul können Polygon- und NURBS-Objekte modelliert werden. Es steht dabei auch eine Vielzahl von Kurven- und Flächenfunktionen zur Verfügung.

Animationsmodul

Im Animationsmodul sind Funktionen für grundlegende Animationstechniken, wie z.B. die Keyframe-Animation oder die Motion-Path-Animation, enthalten. Es werden außerdem Deformations- und Sonderwerkzeuge für die Charakteranimation sowie für die Entwicklung von Expressions einbezogen.

Artisan

Artisan ist eine Schnittstelle, die eine intuitive Arbeitsweise ermöglicht. Mit dem Artisan-Pinselsystem ist es möglich auf Oberflächen zu malen oder Körper zu verformen.

Dynamics-Modul

Möchte man physikalische Vorgänge, wie z.B. Schwerkraft oder Windkraft, simulieren, findet man im Dynamics-Modul die Befehle zur Berechnung solch komplexer Bewegungsabläufe. Diese können sowohl auf starre, als auch auf nachgiebige Körper eingehen. Außerdem gibt es im Dynamics-Modul die Option, Partikelsysteme herzustellen.

Rendering-Modul

Das Rendering-Modul setzt die Arbeit in fotorealistische Bilder um. Dabei stehen Editoren zur Verfügung, die bis ins Detail auf den Rechenprozess Einfluss nehmen können. Mit der Version 5.0 in Maya hat man sogar die Möglichkeit, einen Vektor-Renderer für Flash oder comicähnliche Effekte einzusetzen. Der externe Renderer Mental Ray ist ebenfalls verfügbar und kann verwendet werden.

Paint Effects-Modul

Das Modul Paint Effect ist neben Artisan ein Werkzeug für die visuelle Gestaltung von komplexen Formen wie Pflanzen oder Haaren. Mit einer unbegrenzten Vielzahl von Pinseln können zweidimensionale Bilder erzeugt werden, die anschließend wie dreidimensionale Objekte im Raum behandelt werden. Diese ”Pseudo-3D-Objekte” können dann sowohl animiert, als auch Schwerkräften unterworfen werden. Seit Maya 5.0 können die Paint-Effects-Formen in Polygonmodelle umgewandelt und dadurch über Renderer, wie z.B. Mental Ray, berechnet werden.

MEL

Maya Complete verfügt über eine interne Skriptsprache, MEL, mit deren Hilfe dann eigene Makros oder Skripte erstellt werden können.

3.2 Maya Unlimited

Maya Unlimited bietet nochmals individuellere Module für die Erledigung spezieller Aufgaben während eines Animationsprojektes.

Maya Cloth-Modul

Maya Cloth steht für die Kreation und Animation realistisch wirkender Kleidung. An Hand von zweidimensionalen Kurven kann ein Schnitt erstellt werden – ähnlich dem Schnittbogen eines realen Kleidungsstücken. Daraus wird anschließend die Kleidung für den Charakter erzeugt. Auf die entworfene Kleidung kann man sogar Kraftfelder wirken lassen, so dass sich das Kleidungsstück bewegt, während die animierte Figur eine Handlung durchführt. Es ist sogar möglich über Parameter zu definieren, aus welchem Stoff das jeweilige Kleidungsstück bestehen soll.

Fur-Modul

Das Fur-Modul dient zur Erzeugung kurzer Haare über mehrere Flächenverbände hinweg. Mit dem interaktiven Artisan-Pinselsystem können die Haare in ihrer Länge und Form verändert werden. Auch Gräser oder fellartige Stoffe können mit Maya-Fur erstellt werden. Das Haarsystem reagiert ebenfalls auf die Animation.

Maya Live-Modul

Maya Live ist ein Motiontracking-Modul, das benutzt wird, um computergenerierte Objekte in Realfilmszenen einzupassen. Aufgrund von Anhaltspunkten in einer gefilmten Bildsequenz wird ein Kameraflug für die Animationsszene errechnet. Anschließend kann ein computergeneriertes Objekt ebenfalls errechnet und über Composit-Software in den Realfilm eingebunden werden.

Maya Fluids-Modul

Mit Fluid Effects werden realistische Simulationen von flüssigen Stoffen erstellt. So kann man aber auch z.B. Flammen, Explosionen usw. schnell und fotorealistisch generieren. Die Möglichkeiten reichen sogar bis dahin, dass man ein Objekt auf den Wellen wippen lässt oder dass Schiffe eine Bugwelle erzeugen könnten.

3.3 Grundlagen der Modellierung

Um eine computergenerierte Szene zu erschaffen, müssen Formen im leeren Raum erstellt werden. Dazu werden einfache Primitive erzeugt und mithilfe verschiedener Werkzeuge verformt oder zu immer komplexeren Konstruktionen zusammengesetzt. Diese Formen und Körper werden dann als Modell bezeichnet. Anschließend kann das Modell animiert, texturiert und gerendert werden.

Für architektonische Darstellungen werden meist Polygone zur Modellierung verwendet, da eine lineare Punkt-zu-Punkt-Verbindung, für streng geometrische Formen am besten geeignet ist. Skulpturen sollten dagegen aus NURBS- oder Subdivision-Körpern erzeugt werden, da die NURBS- und Subdivisionmodellierung organische und geschwungene Formen unterstützt.

Die Weltachse ist die Grundvoraussetzung für die Definition eines virtuellen Raumes. Außerdem besitzt jedes Objekt einen Bezugspunkt, der Pivot genannt wird und ein Repräsentant des lokalen Achsensystems eines Objektes ist. Der Pivot-Punkt definiert auch den Abstand des Objektes vom Weltursprung und somit seine Position im Raum. Zusätzlich besitzt jede Fläche ein zweidimensionales Koordinatensystem, das mit seinen Achsen u und v, die Oberfläche beschreibt.

Um strukturiert arbeiten zu können, hat man in Maya die Möglichkeit, Gruppen aus mehreren Objekten zu bilden und deren Bezugspunkt, den Pivot, zu beieinflussen. Damit kann man eigene Hierarchien erzeugen. Eine schematische Übersicht darüber, was in der Szene dann alles existiert, hat man im *Hypergraph*-Fenster.

Wenn man in Maya Primitive erzeugt, dann sind diese automatisch mit vordefinierten Texturkoordinaten belegt. Wenn man aber mit Polygonen ein Modell aufbaut und bearbeitet, gehen in der Regel die Texturkoordinaten verloren oder müssen nach eigenen Vorstellungen neu erzeugt werden. Komplexere Polygonmodelle benötigen also eine explizite Zuweisung der uv-Texturkoordinaten.

Um Texturkoordinaten auf Polygonobjekte anzubringen, gibt es in Maya spezielle Projektionswerkzeuge, die Texturzuweisungen erstellen (Texture Mapping). Dabei gibt es die Möglichkeiten

- *Planar Mapping*:
  für ebene Flächen
- *Cylindrical Mapping*:
  für Formen, die einer Röhre gleichen
- *Spherical Mapping*:
  für Körper, die Rundungen ähnlich einer Kugel haben
- und *Automatic Mapping*.

Es werden sogenannte uv-Maps generiert, die die Oberfläche eines Polygonmodells in einem uv-Koordinatensystem beschreiben. Polygonmodellen können mehrere uv-Maps zugewiesen werden, zwischen denen man dann umschalten kann.

Im *UV-Texture-Editor* kann eine Abwicklung der Texturkoordinaten dargestellt werden. Dort werden die Flächen eines Objektes über ein Bild plaziert, so dass der Verlauf einer Textur über mehrere Flächen besser beurteilt und bearbeitet werden kann. Diese Abwicklung der Texturkoordinaten endet in einer zweidimensionalen Darstellung des dreidimensionalen Objektes.

3.4 Export von Modelldateien

Wenn man in einem Programm komplexe Modelle darstellen möchte, die man nicht ohne weiteres einfach so per Hand zusammensetzen könnte, indem man mühsam Polygon für Polygon erzeugt, hilft man sich einfach mit einem 3D-Modeling-Tool weiter, wie z.B. Maya oder 3DStudio Max. Mit diesem Modellierungsprogramm erstellt man eine Modelldatei, die anschließend mit einem Exporter in ein Format umgewandelt wird, das die Anwendung, in der es verwendet werden soll, lesen kann. Dann lädt man die exportierte Modelldatei in das Programm und kann dort so das 3D-Modell darstellen.

In einer Modelldatei sind folgende Daten enthalten:

- Informationen über Vertizes, aus denen das Modell besteht, mit frei definierbarem, also nicht festgelegtem, Vertexformat
- Indizes, um doppelte Vertizes und damit verschwendeten Speicher zu vermeiden
- Materialeinstellungen, Texturen, Render-States, usw.
- evtl. Animationsdaten oder Beleuchtung

In DirectX werden X-Dateien als 3D-Modelldateien verwendet. Modelldateien, die mit Maya erstellt wurden, liegen im .mb-Format vor. Standardmäßig gibt es in Maya aber leider keinen Exporter, der das .mb-Format in ein .x-Format umwandelt. Deshalb ist es notwendig sich einen entsprechenden Exporter zu beschaffen (oder sich einen selbst zu schreiben, wenn man Lust hat) und diesen als Plug-In in Maya einzubinden, um X-Dateien für DirectX erstellen zu können. Liegen diese schließlich vor, dann sind die D3DX-Hilfsfunktionen

- *D3DXLoadMeshFromX* und
- *D3DXLoadMeshHierarchyFromX*

von **Direct3D** in der Lage das Modell aus einer X-Datei in das Programm zu laden.

Das .x-Format speichert alles hierarchisch ab, so dass die Hierarchie eines Modells in Maya auch in der X-Datei erhalten bleibt. Diese kann dann auch im Programm genutzt werden, wenn die X-Datei mit der Funktion *D3DXLoadMeshHierarchyFromX* in die Anwendung geladen wird.

Der in dieser Studienarbeit verwendete Maya-Exporter für X-Dateien ist nicht in der Lage NURBS-Primitve richtig zu exportieren. Deshalb mussten alle Modelle ausschließlich mit Polygon-Primitven modelliert werden. Aber selbst dann hat der Exporter an manchen Stellen Probleme mit der richtigen Konvertierung. Je nachdem wie komplex ein Polygon-Objekt ist, ist es sinnvoll es in Maya "vorzutriangulieren", so dass diese Triangulierung von DirectX übernommen wird und nicht selbst vorgenommen werden muss. Zusätzlich müssen für die richtige Darstellung der Texturen in der X-Datei die u- und v-Texturkoordinaten beim Export "geflippt" werden. Sonst erscheinen sie in der Grafikanwendung auf dem Kopf stehend. Wenn man alle diese Aspekte beachtet, sollte DirectX in der Lage sein, die 3D-Modelldaten korrekt auszuwerten und darzustellen.

4 Konzept-Entwicklung

Nach der Einarbeitung in DirectX stand die Entscheidung an, auf welche Komponenten und weiter auf welche Möglichkeiten dieser man sich bei der Entwicklung der Grafikanwendung konzentrieren will. Die Wahl fiel logischerweise einmal auf **Direct3D**, da es ja schließlich eine Grafikapplikation sein sollte, und zusätzlich sollte auch **DirectSound** zur Anwendung kommen.

Im Bereich von **Direct3D** interessierten mich persönlich folgende Techniken:

- *Alpha – Blending* :
  Zeichnen von teilweise transparenten oder durchsichtigen Objekten

  Beispielmöglichkeiten:

  - Flamme einer Fackel

  - Schutzschild eines Raumschiffs
  - Korona der Sonne
  - Geistergestalt

- $Multi-Texturing$:
  mehrere Texturen werden übereinander auf ein und dasselbe Polygondreieck gelegt und unterschiedlich bewegt, so dass grafische Effekte mit Speichereinsparungen möglich sind (**Direct3D** ermöglicht bis zu 16 Texturen gleichzeitig übereinander)

  Beispielmöglichkeiten:

  - Wasser
  - Wolken
  - Feuer
  - Energiefeld

- $Volume-Texturing$:
  eine flache Textur wird zu einem Quader erweitert, so dass dreidimensionale Texturen mit u-, v- und w-Achse entstehen

  Beispielmöglichkeiten:

  - Landschaften mit unterschiedlichen Höhenbereichen
  - Animationen (z.B. eine Explosion, wobei die w-Achse als Zeitachse benutzt wird)
  - Licht und Schatten (als Alternative zu Light-Maps)

- $Environment-Mapping$:
  sechs Einzeltexturen werden zu einer würfelförmigen Textur zusammengesetzt

  Beispielmöglichkeiten:

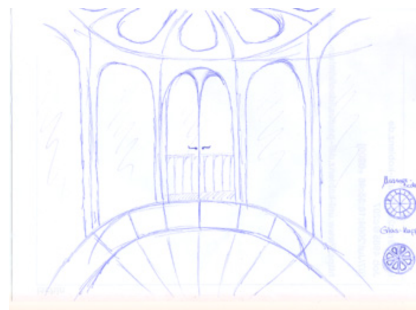
  - Sky-Box
  - Simulation spiegelnder Oberflächen in Echtzeitgrafik

Daraufhin wurden Ideen für ein Szenario entwickelt, in welchem diese Techniken ansprechend umgesetzt werden könnten. Das Ergebnis ist ein 3D-Modell einer kleinen Fantasiewelt, mit einem atmosphärischen Raum als Mittelpunkt und einer futuristischen, stadtähnlichen Umgebung außerhalb dieses Raumes. Innerhalb der Welt kann man sich frei bewegen und hat so die Möglichkeit, die verschiedenen Grafik-Effekte in der soundunterlegten Umgebung zu entdecken.

4.1 Das 3D-Modell

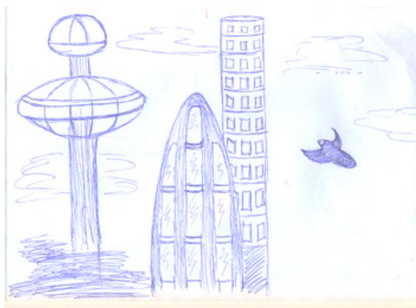
Zuerst wurden grobe Skizzen der 3D-Welt angefertigt, anhand derer dann die 3D-Modelle in Maya modelliert wurden.

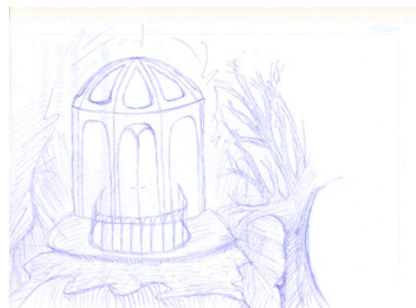
Einmal sollte es einen runden Raum mit einer großen Fensterfront geben, der nach oben mit einer Kuppel abgeschlossen ist, die ebenfalls Fensteröffnungen in Tropfenform besitzt. Innerhalb des Raumes sollten sich diverse Möbel, wie Stühle und ein Tisch, eine Tür, Bücherregale und ein Kamin befinden. An den Raumwänden sollten Bilder hängen.



Skizzen vom Raum

Außerhalb des Raumes sollten verschiedene Gebäude zu sehen sein, die rundherum von Wasser umgeben sind.

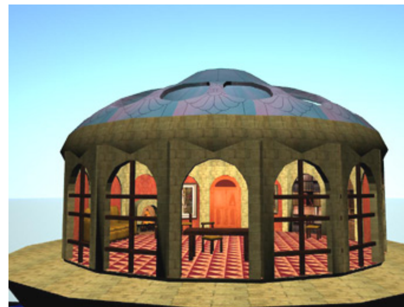




Skizze der Stadt und des Raumes von außen

Anhand dieser Skizzen wurde folgende Welt in Maya modelliert.

zwei Ansichten des modellierten Raumes



Stadtansicht und der Raum von außen

Auf der Basis dieses Modells wurde in DirectX die Fantasiewelt schließlich vervollständigt. Folgende Details kamen hinzu:

- Kamera-Steuerung,
- Sky-Box,
- Beleuchtung,
- Nebel,
- Wasser,
- Wolken,
- Feuer im Kamin,
- Spiegelung in den Fenstern,
- Hintergrundmusik,
- Wellengeräusche,
- Sound für das Feuerknistern und schließlich
- Schrittgeräusche, wenn man sich in der Welt bewegt.

5 Implementierung der Grafikanwendung

In der Hauptfunktion *WinMain*() wird zuerst ein Fenster für die Anwendung erstellt. Anschließend wird **Direct3D** und **DirectSound**, wie oben beschrieben, initialisiert. Die globale Variable *g_pD3DDevice* repräsentiert dabei die Geräteschnittstelle für **Direct3D** und *g_pDSound* die für **DirectSound**. Danach wird die Funktion für die Initialisierung der Szene aufgerufen. In ihr werden:

- die Render-States gesetzt, die das Verhalten des Rasterizers kontrollieren und festlegen, wie er seine Dreiecke zu zeichnen hat,
- das Vertex-Format gesetzt, das angibt, welche Daten ein Vertex enthält,
- die Sampler-States gesetzt, die festlegen, wie die Grafikkarte beim Zeichnen eines Pixels vorgeht, um aus den über die Vertizes hinweg interpolierten Texturkoordinaten und der Textur selbst eine Farbe zu erhalten,
- die verschiedenen Texturen geladen, die in der Grafikapplikation verwendet werden und schließlich
- die 3D-Modelle aus den X-Dateien geladen.

Nach der Initialisierung der Szene wird die Nachrichtenschleife der Anwendung betreten, die ihrerseits die *Render()*-Funktion, in der alle Zeichenbefehle stehen, und die *Move()*-Funktion, die für die Bewegung innerhalb der Szene einerseits und für den flüssigen Programmablauf andererseits, zuständig ist, aufruft. Wenn die Nachrichtenschleife verlassen wird, dann wird **DirectSound**, **Direct3D**, die Szene und schließlich das Anwendungsfenster heruntergefahren und alle Schnittstellen wieder freigegeben, um eventuelle Speicherlecks zu vermeiden.

5.1 Initialisierung der Szene

5.1.1 Render-States

Wurde ein Render-State einmal gesetzt, gelten seine Einstellungen so lange, bis diesem wieder ein neuer Wert zugewiesen wird. Mit der Methode *SetRenderState()* werden die Render-States gesetzt und mit *GetRenderState()* können sie abgefragt werden.

Quellcode:

```
//Beleuchtung einschalten
g_pD3DDevice->SetRenderState (D3DRS_LIGHTING, TRUE);
//Culling ausschalten
g_pD3DDevice->SetRenderState (D3DRS_CULLMODE, D3DCULL_NONE);
//Dithering aktivieren
g_pD3DDevice->SetRenderState (D3DRS_DITHERENABLE, TRUE);
//Z-Buffer-Test für die zu zeichnenden Pixel ausschalten
g_pD3DDevice->SetRenderState(D3DRS_ZENABLE, FALSE);
//das Schreiben in den Z-Buffer ist nicht erlaubt
g_pD3DDevice->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);
```

5.1.2 Vertexformat

Ein Vertex kann viele verschiedene Informationen, wie z.B. die Positionsangabe oder die Farbangabe, beinhalten. Diese Daten werden zwischen den Vertizes interpoliert. **Direct3D** erlaubt es, den Inhalt des Vertexformats, frei zu wählen, aber nicht die Anordnung der Daten darin. Die Farben werden z.B. immer nach der Position angegeben.

Wenn ein Vertexformat mit der Methode *SetFVF()* gesetzt wurde, erwartet **Direct3D**, dass ab diesem Zeitpunkt alle zu zeichnenden Primitiven aus Vertizes diesen Formats bestehen, bis ein neues Format festgelegt wird. Das Vertexformat selbst wird in einer Struktur definiert, die mit dem *struct*-Befehlt erstellt wird.

Quellcode:

```
//*********************************************************
//Struktur für einen Vertex festlegen
//*********************************************************
//Vertex soll Positionsinformationen und Farbinformationen enthalten
struct SVertex
  {
    //Position des Vertex
    //Struktur D3DXVECTOR3 definiert einen Vektor,
    //der aus drei float-Werten besteht: x, y, z
    D3DXVECTOR3 vxPosition;

    //Farbe des Vertex
    //Struktur D3DCOLOR definiert den elementare Direct3D-Farbtyp
    D3DCOLOR vxColor;

    //Texturkoordinaten des Vertex
    //Struktur D3DXVECTOR2 definiert zwei float-Werte: x, y
    D3DXVECTOR2 vxTextur;
  };
```

```
//*******************************************************************
//Vertexformat setzen
//*******************************************************************
//mit Positionsangabe, Farbangabe und einem Paar Texturkoordinaten
if (FAILED (hResult =
   g_pD3DDevice->SetFVF (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1)))
{
   //Fehler beim Setzen des Vertexformats
   WriteToLog ("&#8226; Fehler beim Setzen des Vertexformats!<br>");

   return 1;
}
```

5.1.3 Sampler-States

Sampler-States sind mit den Render-States vergleichbar. Sie werden für jede Texturschicht einzeln gesetzt. Wie schon erwähnt, kontrollieren sie dann für jede Texturschicht, wie die Grafikkarte vorgehen soll, um die Farbe an einer bestimmten Stelle in der Textur herauszufinden.

Die Sampler-States bestimmen unter anderem das Verhalten bei Texturkoordinaten, die außerhalb von [0; 1] liegen und auch die Texturfilter, die die Qualität der Texturen auf dem fertigen Bild verbessern. Man setzt sie mit der Methode *SetSamplerState()* und fragt sie mit *GetSamplerState()* wieder ab.

- Das Sampler-State *D3DSAMP_MAGFILTER* legt fest, welcher Vergrößerungsfilter verwendet werden soll, wenn eine Textur auf dem Bildschirm größer erscheint, als sie tatsächlich ist. Wenn sie also gestreckt werden muss.

- *D3DSAMP_MINFILTER* bestimmt dagegen, welcher Verkleinerungsfilter angewendet werden soll, wenn eine Textur auf dem Bildschirm kleiner erscheint.

- Mit *D3DSAMP_MIPFILTER* gibt man an, wie entschieden wird, welche MIP-Map verwendet werden soll. Von ein und derselben Textur gibt es nämlich mehrere Versionen (MIP-Maps), von denen jede jeweils vier mal so klein ist, wie die vorhergehende.

Quellcode:

```
// Bilineare Texturfilter mit linearem MIP-Mapping
// für die ersten drei Texturschichten
g_pD3DDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(1, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(1, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(1, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(2, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(2, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
g_pD3DDevice->SetSamplerState(2, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
```

5.1.4 Texturen laden

In **Direct3D** wird jede Textur durch die Schnittstelle *IDirect3DTexture9* repräsentiert. Für jede Textur, die die Anwendung verwenden möchte, wird eine solche Schnittstelle durch die Funktion *D3DXCreateTextureFromFileEX()*angelegt.

In der Grafikapplikation soll das Wasser durch zwei verschiedene Texturen, die übereinander gelegt und unterschiedlich transformiert werden, simuliert werden. Für die Realisierung des Multi-Texturing werden deshalb erstmal zwei Wassertexturen in das Programm geladen.

Quellcode:

```
//zwei Texturen für das Wasser
VAR PDIRECT3DTEXTURE9 g_apWaterTexture [2];
```

```
//*************************************************************
//Laden und aktivieren der Texturen für das Wasser
//*************************************************************
char acFilename [256];
//die Textur laden
for (int iTexture = 0; iTexture < 2; iTexture++)
{
   sprintf (acFilename, "waterScene%d.jpg", iTexture + 1);
```

```
    if (FAILED (hResult = D3DXCreateTextureFromFileEx (g_pD3DDevice,
                              acFilename,
                              D3DX_DEFAULT,
                              D3DX_DEFAULT,
                              D3DX_DEFAULT,
                              0,
                              D3DFMT_UNKNOWN,
                              D3DPOOL_MANAGED,
                              D3DX_FILTER_NONE,
                              D3DX_DEFAULT,
                              0,
                              NULL,
                              NULL,
                              &g_apWaterTexture [iTexture])))
    {
        WriteToLog ("&#8226; D3DXCreateTextureFromFileEx() failed!<br>");
        return 1;
    }
}//for
```

5.1.5 Umgebungstexturen laden

Für eine Umgebungstextur wird aus sechs einzelnen Texturen ein 360-Panorama in alle Richtungen erzeugt, indem diese sechs Einzeltexturen in einem Würfel angeordnet werden. Diese sogenannten kubischen Umgebungstexturen oder auch Sky-Boxes genannt, werden aus DDS-Dateien geladen und benötigen außerdem dreidimensionale Texturkoordinaten. Diese Texturkoordinaten sind als Richtungsvektoren zu verstehen.

Für kubische Umgebungstexturen gibt es die spezielle Schnittstelle *IDirect3DCubeTexture9*. Das Laden einer Umgebungstextur übernehmen die beiden D3DX-Funktionen *D3DXCreateCubeTextureFromFile()* und *D3DXCreateCubeTextureFromFileEx()*. Wobei es mit der *Ex*-Version der Funktion möglich ist den Ladevorgang genauer zu kontrollieren.

Quellcode:

```
//******************************************************
//Struktur für einen Vertex der Sky-Box
//******************************************************
struct SSkyBoxVertex
{
    D3DXVECTOR3 vPosition;     // Position des Vertex
    D3DXVECTOR3 vTexture;      // 3D-Texturkoordinaten
};
```

```
// Die Umgebungstextur
VAR PDIRECT3DCUBETEXTURE9 g_pEnvMap;
// Vertizes der Sky-Box
VAR SSkyBoxVertex g_aSkyBoxVertex[8];
// Indizes der Sky-Box
VAR unsigned short g_ausSkyBoxIndex[36];


//*****************************************************************
//Laden der Umgebungstextur
//*****************************************************************
// Die Umgebungstextur ohne MIP-Maps laden
if(FAILED(hResult = D3DXCreateCubeTextureFromFileEx(g_pD3DDevice, // Device
            "SB1.dds",                  // Dateiname
            D3DX_DEFAULT,               // Größe
            1,                          // MIP-Maps
            0,                          // Verwendungszweck
            D3DFMT_UNKNOWN,             // Format
            D3DPOOL_MANAGED,            // Speicherklasse
            D3DX_FILTER_NONE,           // Filter
            D3DX_DEFAULT,               // MIP-Map-Filter
            0,                          // Color-Key
            NULL,                       // Unwichtig
            NULL,                       // Unwichtig
            &g_pEnvMap)))               // Die Textur
{
    // Fehler!
    WriteToLog ("&#8226; D3DXCreateCubeTextureFromFileEx failed!<br>");
}



// Die Vertizes der Sky-Box erstellen
g_aSkyBoxVertex[0].vPosition = D3DXVECTOR3 (-75.0f,  85.0f, 100.0f);
g_aSkyBoxVertex[1].vPosition = D3DXVECTOR3 ( 75.0f,  85.0f,  100.0f);
g_aSkyBoxVertex[2].vPosition = D3DXVECTOR3 ( 75.0f,  85.0f, -50.0f);
g_aSkyBoxVertex[3].vPosition = D3DXVECTOR3 (-75.0f,  85.0f, -50.0f);
g_aSkyBoxVertex[4].vPosition = D3DXVECTOR3 (-75.0f, -15.0f,  100.0f);
g_aSkyBoxVertex[5].vPosition = D3DXVECTOR3 ( 75.0f, -15.0f,  100.0f);
g_aSkyBoxVertex[6].vPosition = D3DXVECTOR3 ( 75.0f, -15.0f, -50.0f);
g_aSkyBoxVertex[7].vPosition = D3DXVECTOR3 (-75.0f, -15.0f, -50.0f);

// Die Texturkoordinaten brauchen man nicht per Hand einzutragen:
// sie entsprechen den Positionsangaben.
for(int iVertex = 0; iVertex < 8; iVertex++)
{
    g_aSkyBoxVertex[iVertex].vTexture = g_aSkyBoxVertex[iVertex].vPosition;
}
```

```
// Nun werden die Indizes eingetragen
unsigned short ausIndex[36] = {7, 3, 0,   4, 7, 0,  // Vorderseite
                               5, 1, 2,   6, 5, 2, // Hinterseite
                               4, 0, 1,   5, 4, 1, // Linke Seite
                               6, 2, 3,   7, 6, 3, // Rechte Seite
                               2, 1, 0,   3, 2, 0, // Oberseite
                               4, 5, 6,   7, 4, 6};    // Unterseite

// In das globale Array kopieren
memcpy(g_ausSkyBoxIndex, ausIndex, 36 * sizeof(unsigned short));
```

5.1.6 Volumentexturen laden

Bei einer Volumentextur wird eine flache Textur zu einem Quader erweitert, so dass sie zusätzlich zur Breite und Höhe eine Tiefe bekommt. Deshalb sind Volumentexturen, wie Umgebungstexturen, ebenfalls dreidimensional. Die dritte Achse ist die w-Achse.

Die Schnittstelle für Volumentexturen heißt *IDirect3DVolumeTexture9*. Die Funktion *D3DXCreateVolumeTextureFromFileEx()* ist für das Laden von Volumentexturen zuständig, die ebenso wie die Umgebungstexturen aus DDS-Dateien geladen werden.

Quellcode:

```
//Volumentextur für das Kaminfeuer
VAR PDIRECT3DVOLUMETEXTURE9 g_pVolumeTexture;
```

```
//******************************************************************
//Laden der Textur für das Kaminfeuer
//******************************************************************
if(FAILED(hResult = D3DXCreateVolumeTextureFromFileEx(g_pD3DDevice, // Device
                 "FeuerNeu2.dds",          // Dateiname
                 D3DX_DEFAULT,           // Breite
                 D3DX_DEFAULT,           // Tiefe
                 D3DX_DEFAULT,           // Höhe
                 1,                      // MIP-Maps
                 0,                      // Verwendungszweck
                  D3DFMT_UNKNOWN,          // Format
                  D3DPOOL_MANAGED,         // Speicherklasse
                 D3DX_FILTER_NONE,        // Filter
                  D3DX_DEFAULT,          // MIP-Map-Filter
```

```
                    0,                          // Color-Key
                    NULL,                         // Unwichtig
                    NULL,                         // Unwichtig
                    &g_pVolumeTexture)))              // Die Textur
{
   // Fehler! Wahrscheinlich werden keine Volumentexturen unterstützt.
   WriteToLog ("&#8226; Volumentextur konnte nicht erstellt werden! <br>");
   return 1;
}
```

5.1.7 Modell aus einer X-Datei laden

Um ein 3D-Modell, das in einer X-Datei gespeichert ist, zu laden, sollte man sich zuerst eine Datenstruktur definieren, die alle Elemente beinhaltet, die zum Laden eines X-Files notwendig sind. Dies sind:

- ein Zähler für die Anzahl der Materialen des 3D-Modells (beim Laden der X-Datei wird jede neue Textur als ein neues Material definiert und die einzelnen Teile des Modells werden nach ihren Materialien geordnet gerendert)
- eine LPD3DXMESH-Struktur, in die die Geometrie der X-Datei geladen wird
- eine D3DMATERIAL9-Struktur, in die die Materialien geladen werden und
- eine LPDIRECT3DTEXTURE9-Struktur, die die Texturen speichert.

Quellcode:

```
//**********************************
//Struktur für ein 3D-Modell
//**********************************
typedef struct X3DMODELL_TYP
{
   DWORD               dwNumMaterials;
   D3DMATERIAL9         *pMeshMaterials;
   LPDIRECT3DTEXTURE9 *pMeshTextures;
   LPD3DXMESH           pMesh;
} X3DMODELL;


//3D-Modell
VAR X3DMODELL Scene;
```

Für das Laden eines 3D-Modells ohne Animation ist die Funktion *D3DXLoadMeshFromX()* zuständig. Sie erwartet:

- als ersten Parameter den Namen der zu ladenden Datei
- als zweiten Parameter den Speicher, in dem das Objekt abgelegt werden soll (Systemspeicher oder Videospeicher)
- als dritten Parameter das **Direct3D**-Device, das die Zeichenbefehle annehmen soll
- als vierten Parameter ein Objekt des Typs LPD3DXBUFFER, das die Funktion mit Informationen über benachbarte Polygonflächen füllt
- als fünften Parameter ebenfalls ein Objekt des Typs LPD3DXBUFFER, das mit Informationen über die verwendeten Materialien des Objektes gefüllt wird
- als sechsten Parameter wieder ein LPD3DXBUFFER-Objekt, das evtl. mit Informationen über mögliche Effekte gefüllt wird
- als siebten Parameter die Anzahl der gefundenen Materialen des 3D-Modells
- und schließlich als letzten Parameter ein LPD3DXMESH-Objekt, in das die Funktion die Geometrie der X-Datei lädt.

Quellcode:

```
//***********************************
//Laden eines 3D-Modells aus einer x-Datei
//***********************************
LPD3DXBUFFER pD3DXMtrlBuffer;
LPD3DXBUFFER ppAdjacency;
LPD3DXBUFFER ppEffectInstances;
Scene.dwNumMaterials = 0L;
Scene.pMeshMaterials = NULL;
Scene.pMeshTextures  = NULL;
Scene.pMesh          = NULL;


// Lade die X File Datei
if (FAILED(D3DXLoadMeshFromX("CityComplete.x",
                              D3DXMESH_SYSTEMMEM,
                              g_pD3DDevice,
                              &ppAdjacency,
                              &pD3DXMtrlBuffer,
                              &ppEffectInstances,
                              &Scene.dwNumMaterials,
                              &Scene.pMesh)))
{
   WriteToLog ("&#8226; City konnte nicht geladen werden! <br>");
 }// if
```

Nach dem Laden eines 3D-Modells müssen die Materialeigenschaften und die Texturnamen aus dem Material-Buffer ausgelesen werden. Anschließend werden so viele Materialobjekte und Texturobjekte erzeugt, wie das Modell Materialien besitzt. Dazu wird für jedes Material das Material aus der D3DXMATERIAL-Struktur kopiert, die ambiente Farbe gesetzt und eine Textur erzeugt. Danach kann der Material-Buffer wieder zerstört werden.

Quellcode:

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

// Speicher allokieren
Scene.pMeshMaterials = (D3DMATERIAL9*)malloc(Scene.dwNumMaterials *
                              sizeof(D3DMATERIAL9));
Scene.pMeshTextures = (LPDIRECT3DTEXTURE9*)malloc(Scene.dwNumMaterials *
                              sizeof(LPDIRECT3DTEXTURE9));
if (!Scene.pMeshMaterials || !Scene.pMeshTextures)
{
   WriteToLog ("&#8226; Fehler: malloc() Mesh Elemente<br>");
}



// Speichere alle Materialien um und lade Texturen
for (DWORD i=0; i<Scene.dwNumMaterials; i++)
{

   Scene.pMeshMaterials[i] = d3dxMaterials[i].MatD3D;

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

   if (FAILED(D3DXCreateTextureFromFile(g_pD3DDevice,
         d3dxMaterials[i].pTextureFilename,
         &Scene.pMeshTextures[i])))
   {
      Scene.pMeshTextures[i] = NULL;
      WriteToLog ("&#8226; Error: DXTexture() failed.<br>");
   }
}
```

5.2 Move-Funktion

Der Ablauf der Grafikanwendung ist im Grunde in zwei Schritte aufgeteilt:

- in die Bewegung (*Move()*-Funktion) und

- in das Zeichnen (*Render()*-Funktion).

Diese beiden Schritte werden ständig währen des gesamten Programmablaufs in der Nachrichtenschleife wiederholt. Damit die Geschwindigkeit der Anwendung auf allen Rechnern gleich ist, wird die *Move()*-Funktion so angepasst, dass sie das Programm um einen beliebigen Zeitpunkt fortbewegt. Hierzu wird die Zeit gemessen, die für einen Durchlauf der Nachrichtenschleife benötigt wird.

Die Windows-Funktion *timeGetTime()* liefert die vergangene Zeit seit dem letzen Systemstart in Millisekunden. Mit deren Hilfe wird der Zeitpunkt vor dem Betreten der *Move()*-Funktion und der Zeitpunkt unmittelbar nach Verlassen der *Render()*-Funktion bestimmt. Die Differenz dieser beiden Werte liefert dann die Zeit, die für einen Durchlauf der Nachrichtenschleife gebraucht wurde. Dieser Wert wird der *Move()*-Funktion als Parameter übergeben und dort zu der globalen Variablen *g_fAnimTime* hinzuaddiert, die die gesamte vergangene Zeit seit dem Programmstart speichert. Diese Variable wird schließlich dazu verwendet, die Grafikapplikation fortzubewegen.

Die Interaktivität der Anwendung ergibt sich dadurch, dass die Sichtmatrix (Kameramatrix) und die Projektionsmatrix einmal pro Bild, je nach den Tastatureingaben des Benutzers, verändert werden. Die Orientierung der Kamera ergibt sich aus

- ihrer Position, einem Vektor, und
- ihrem Drehwinkel, der angibt, in welche Richtung die Kamera sieht.

Die Bewegung der Kamera ist in dieser Anwendung auf die *xz*-Ebene beschränkt.

In der Grafikapplikation wird mit Hilfe der Funktion *GetAsyncKeyState()*, die in *MMSystem.h* und *WinMM.lib* definiert ist, abgefragt, ob der Benutzer irgendwelche Tasten drückt, die für das Programm relevant sein könnten:

- Wenn die Taste "Pfeil links" gedrückt wird, wird der Drehwinkel der Kamera verkleinert, denn dann bedeutet es, dass sie sich gegen den Uhrzeigersinn dreht.
- Wird die Taste "Pfeil rechts" gedrückt, wird der Drehwinkel vergrößert.
- Die Taste "Pfeil hoch" verändert die Kameraposition. Die Blickrichtung, die sich aus dem Sinus und dem Kosinus des Blickwinkels ergibt, wird hinzuaddiert, so dass die Kamera sich nach vorne bewegt.
- Die Taste "Pfeil runter" bewirkt eine Rückwärtsbewegung, indem die Blickrichtung von der Position der Kamera abgezogen wird.

Quellcode:

```
//****************************************************************
//Move-Funktion
//****************************************************************

int Move (float fNumSecsPassed)
{
   //Zeitzähler erhöhen
   g_fAnimTime += fNumSecsPassed;

   D3DXVECTOR3 vCameraDirection;

   // Die Kamera soll sich mit 45° pro Sekunde drehen.
   if(GetAsyncKeyState(VK_LEFT))
      g_fCameraAngleHoriz -= D3DXToRadian(45.0f) * fNumSecsPassed;
   if(GetAsyncKeyState(VK_RIGHT))
      g_fCameraAngleHoriz += D3DXToRadian(45.0f) * fNumSecsPassed;

   // Berechnung der Blickrichtung.
   vCameraDirection = D3DXVECTOR3  (sinf(g_fCameraAngleHoriz),
                                      0.0f, cosf(g_fCameraAngleHoriz));


   // Die Kamera soll sich mit 5 Einheiten pro Sekunde bewegen.
   // Die Blickrichtung ist normalisiert und hat daher die Länge 1.
   if(GetAsyncKeyState(VK_UP))
   {
      g_vCameraPosition += vCameraDirection * 2.0f * fNumSecsPassed;
   }

   if(GetAsyncKeyState(VK_DOWN))
   {
      g_vCameraPosition -= vCameraDirection * 2.0f * fNumSecsPassed;
   }

   return 0;

}//Ende von Move()
```

In der *Render()*-Funktion wird dann eine Sichtmatrix und eine Projektionsmatrix erzeugt, die die aktuellen Werte für den Blickwinkel, die Kameraposition und das Sichtfeld, die in der *Move()*-Funktion berechnet wurden, beinhalten.

5.3 Render-Funktion

Die *Render()*-Funktion ist hauptsächlich für das Zeichnen bzw. für das Berechnen der Grafiken des aktuellen Programmstatus zuständig.

Ganz am Anfang der *Render()*-Funktion wird der Bildpuffer, der Z-Buffer und wenn vorhanden, auch der Stecil-Buffer mit der Methode *Clear()* geleert. Dies sollte immer erfolgen, bevor überhaupt etwas gezeichnet wird. Sonst kann es zu Pixelfehlern kommen.

Quellcode:

```
//*************************************************************
//den Bildpuffer und den Z-Buffer leeren
//*************************************************************
if (FAILED (hResult = g_pD3DDevice->Clear (0,
                            NULL,
                            D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                            D3DCOLOR_XRGB (0, 0, 0),
                            1.0f,
                            0)))
//  IDirect3DDevice9::Clear (Anzahl von Rechtecken;
//                              Zeiger auf ein Array mit den Rechtecken;
//                              Flags für die zu löschenden Oberflächen;
//                              Farbe, mit der gelöscht werden soll;
//                              neuer Z-Wert;
//                              Stencil)
{
   //Fehler beim Leeren
   WriteToLog ("&#8226; Leeren des Bildpuffers & Z-Buffers failed! <br>");

   return 1;
}
```

Nach dem Leeren des Bildpuffers und des Z-Buffers kann das tatsächliche Zeichnen der Szene beginnen. Dies wird **Direct3D** mit der Methode *BeginScene()* mitgeteilt. Wenn man mit dem Zeichnen der 3D-Szene wieder fertig ist, also am Ende der *Render()*-Funktion angelangt ist, wird die Methode *EndScene()* aufgerufen. Und der Aufruf der Methode *Present()* macht den Inhalt des Bildpuffers schließlich sichtbar.

Quellcode:

```
//************************************************************
//Szene zeichnen
//************************************************************

//Szene beginnen: tatsächliches Zeichnen der 3D-Primitiven
g_pD3DDevice->BeginScene ();

//verschiedene Zeichenbefehle

g_pD3DDevice->EndScene ();

//Inhalt des Bildpuffers sichtbar machen
//durch Kopieren in den sichtbaren Videospeicher
g_pD3DDevice->Present (NULL, NULL, NULL, NULL);
// IDirect3DDevice9::Present (Rechteck in Pixelkoordinaten,
//                             das sichtbar gemacht werden soll,
//                             oder NULL für den gesamten Bildpuffer;
//                             Rechteck in Pixelkoordinaten auf dem Zielbild
//                             oder NULL für das gesamte Bild;
//                             Handle des Fensters in dem das Bild
//                             erscheinen soll oder NULL für das
//                             ursprünglich vorgesehene Fenster;
//                             immer NULL)
```

5.3.1 Sichtmatrix und Projektionsmatrix setzen

Die einzelnen Transformationsmatrizen der Transformationspipeline werden mit der Methode *SetTransform()* gesetzt und können mit der Methode *GetTransform()* auch wieder abgefragt werden. Folgende Transformationsmatrizen stehen zur Verfügung:

- die Weltmatrix D3DTS_WORLD,
- die Sichtmatrix oder Kameramatrix D3DTS_VIEW und
- die Projektionsmatrix D3DTS_PROJECTION.

Alle Matrixberechnungen finden innerhalb der *Render()*-Funktion statt.

Um die Kameramatrix zu erzeugen, braucht man die Kameraposition, den Blickpunkt der Kamera und die lokale y-Achse der Kamera, die normalerweise (0, 1, 0) ist, außer die Kamera "rollt". Damit das Programm auf verschiedenen Rechnern gleich schnell abläuft, wurden in der *Move()*-Funktion bestimmte Variablen bereits mit der globalen Zählervariablen verrechnet.

Wenn keine spezielle Sichtmatrix gesetzt wird, dann befindet sich der Beobachter im Punkt (0, 0, 0).

Quellcode:

```
//******************************************************************
//Kamera setzen
//******************************************************************
D3DXMATRIX mCamera;                 //Kameramatrix
D3DXVECTOR3 vCameraLookAt;         //Blickrichtung der Kamera
D3DXVECTOR3 vCameraPosHigh;        //Kameraposition ist oben, also im Raum

ZeroMemory(&mCamera, sizeof(D3DXMATRIX));

vCameraPosHigh = D3DXVECTOR3 (0.0f, 11.5f, 0.0f) + g_vCameraPosition;

vCameraLookAt = vCameraPosHigh +
                     D3DXVECTOR3 (sinf(g_fCameraAngleHoriz),
                                  0.0f,
                                  cosf(g_fCameraAngleHoriz));

//Sichtmatrix erzeugen (linkshändiges Koordinatensystem)
D3DXMatrixLookAtLH (&mCamera, &vCameraPosHigh,
                              &vCameraLookAt,
                              &D3DXVECTOR3 (0.0f, 1.0f, 0.0f));

//Sichtmatrix wird als Kameratransformationsmatrix eingesetzt
g_pD3DDevice->SetTransform (D3DTS_VIEW, &mCamera);
```

Die Funktion *D3DXMatrixPerspectiveFovLH()* erzeugt eine linkshändige Projektionsmatrix. Als Parameter erwartet sie unter anderem das Sichtfeld, das Bildseitenverhältnis, die nahe und die ferne Clipping-Ebene.

Quellcode:

```
//**************************************************************
//Projektion erstellen
//**************************************************************
D3DXMATRIX mProjection;             //Projektionsmatrix


ZeroMemory(&mProjection, sizeof(D3DXMATRIX));

//Bildseitenverhältnis berechnen
fAspect = (float) (g_iWidth) / (float) (g_iHeight);
```

```
//D3DXMatrixPerspectiveFovLH() erzeugt eine linkshändige
//Projektionsmatrix, die auf dem Sichtfeld basiert
D3DXMatrixPerspectiveFovLH (&mProjection,      //Output-Matrix
                    D3DXToRadian (60.0f),   //Sichtfeld 60° in Radian
                    fAspect,                //Bildseitenverhältnis
                    0.01f,                  //nahe Clipping-Ebene
                    250.0f);                //ferne Clipping-Ebene

//Projektionsmatrix einsetzen
g_pD3DDevice->SetTransform (D3DTS_PROJECTION, &mProjection);
```

Die Weltmatrix legt die Position und die Ausrichtung der Objekte in einer 3D-Szene fest. Bevor diese also gezeichnet werden, muss deren Weltmatrix gesetzt werden, die die Objekte in die gewünschte Position verschiebt und dreht.

5.3.2 Sky-Box zeichnen

Die Umgebungstextru für die Sky-Box wurde bereits in der Initialisierungsfunktion der Szene geladen. Genauso wurde dort für die Sky-Box ein großer Würfel erzeugt, auf dem die Umgebungen dann in der *Render()*-Funktion schließlich gezeichnet werden sollen. Es ist zu beachten, dass die Flächen des Würfels nach innen zeigen müssen, damit sie im Programm auch sichtbar sind. Der Würfel wird normalerweise so positioniert, dass die Kamera möglichst in dessen Mittelpunkt steht.

Da die Umgebungstextur dreidimensionale Texturkoordinaten besitzt, muss vor dem Zeichnen der Sky-Box das entsprechende Vertex-Format, das diese Texturkoordinaten enthält, gesetzt werden. Außerdem sollte die Beleuchtung, falls sie vorher aktiv war, ausgeschaltet werden. Nach dem Zeichnen der Sky-Box wird die Beleuchtung und das Vertex-Format der Szene, durch das Setzen der entsprechenden Render-States, wieder aktiviert.

Die bereits geladene und in *g_pEnvMap* gespeicherte Umgebungstextur wird mit der Methode *SetTexture()* in die erste Texturschicht gesetzt und anschließend wird die Sky-Box mit der Methode *DrawIndexedPrimitiveUP()* gezeichnet.

Quellcode:

```
//*********************************************
//Sky-Box erstellen
//*********************************************
// Zu Beginn muss das Vertexformat gesetzt werden.
g_pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEX1 |
                     D3DFVF_TEXCOORDSIZE3(0));

// Die Beleuchtung wird abgeschaltet
g_pD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
```

```
// In die erste Texturschicht wird die Umgebungstextur gesetzt
if(FAILED(hResult = g_pD3DDevice->SetTexture(0, g_pEnvMap)))
   WriteToLog ("&#8226; g_pD3DDevice->SetTexture failed!<br>");

// Jetzt wird die Sky-Box mit DrawIndexedPrimitiveUP gezeichnet
if(FAILED(hResult = g_pD3DDevice->
   DrawIndexedPrimitiveUP(D3DPT_TRIANGLELIST,  // Dreiecksliste
                          0,                   // Beginn bei 0
                          8,                   // Größter Index
                          12,                  // Anz. Dreiecke
                          g_ausSkyBoxIndex,    // Indizes
                          D3DFMT_INDEX16,      // 16 Bits
                          g_aSkyBoxVertex,     // Vertizes
                          sizeof(SSkyBoxVertex)))) // Vertexgröße
{
   // Fehler beim Zeichnen!
   WriteToLog ("&#8226; DrawIndexedPrimitiveUP failded!<br>");
}

// Die Einstellungen wieder zurücksetzen
g_pD3DDevice->SetFVF (D3DFVF_XYZ | D3DFVF_DIFFUSE |
                      D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0));
g_pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
```

5.3.3 Beleuchtung

Es gibt in **Direct3D** verschiedene Lichttypen, die sich dadurch unterscheiden, wie sie ihr Licht abgeben.

- D3DLIGHT_POINT: Punktlichter geben wie Glühbirnen das Licht in alle Richtungen ab – sie sind die teuersten Lichter
- D3DLIGHT_SPOT: Spotlichter bündeln das Licht in einem Lichtkegel, der verschieden groß sein kann
- D3DLIGHT_DIRECTIONAL: Richtungslichter sind extrem weit entfernt und geben ihr Licht deshalb scheinbar nur in eine einzige Richtung ab
- D3DLIGHT_AMBIENT: das allgegenwärtige Hintergrundlicht beeinflusst alle Vertizes genauso, wie die Hintergrundfarbe eines echten Lichts

In **Direct3D** wird die Beleuchtung nicht auf Basis der Dreiecke durchgeführt, sondern für die einzelnen Vertizes, da dies nicht zu extrem scharfen Beleuchtungskanten führt. Man hat außerdem noch die Möglichkeit *Light-Maps* einzusetzen, so dass der Lichtfleck, den eine Lichtquelle auf ein Objekt werfen würde, in Form einer Textur auf dieses Objekt gelegt wird.

Wenn man in **Direct3D** mit Beleuchtung arbeiten will, muss das Render-State D3DRS_LIGHTING auf TRUE gesetzt werden. Wenn man mit der Beleuchtung fertig ist, kann man es ruhig wieder auf FALSE setzen.

Die Struktur für eine Lichtquelle heißt D3DLIGHT9. Die Elemente dieser Struktur sind nicht für alle Lichtquellarten gleich relevant. Vor der Verwendung der D3DLIGHT9-Struktur, sollte diese immer zurückgesetzt werden, damit es nicht zu unerwarteten Effekten kommt.

Wieviele gleichzeitig aktive Lichter unterstützt werden, hängt von der verwendeten Grafikkarte ab, denn **Direct3D** erlaubt theoretisch unendlich viele Lichter, die gleichzeitig aktiv sind. Wie bei Texturen werden mehrere Lichter in Schichten angeordnet und jeder Platz wird ebenfalls durch einen Index beschrieben, wobei auch hier der kleinste Index 0 ist.

Ein Licht wird mit der Methode *SetLight()* gesetzt. Anders als bei Texturen, muss es aber noch "eingeschaltet" werden. Dies erfolgt mit der Methode *LightEnable()*. Der erste Parameter ist der Index des Lichts, das aktiviert werden soll und der zweite Parameter ist TRUE. Soll das Licht hingegen "ausgeschaltet" werden, muss der zweite Parameter FALSE lauten.

Quellcode:

```
//*****************************************************
//Licht erzeugen
//*****************************************************
//Beleuchtung für den Raum der Szene
D3DLIGHT9 PointLightRoom;
//Sonnenlicht
D3DLIGHT9 PointLightSun;

ZeroMemory(&PointLightRoom, sizeof(D3DLIGHT9));

// Punktlicht
PointLightRoom.Type = D3DLIGHT_POINT;
//Streufarbe
PointLightRoom.Diffuse = D3DXCOLOR (0.95f, 0.85f, 0.85f, 0.0f);
//Hintergrundfarbe
PointLightRoom.Ambient = D3DXCOLOR (0.95f, 0.85f, 0.85f, 0.0f);
//Glanzfarbe
PointLightRoom.Specular = D3DXCOLOR (0.95f, 0.85f, 0.85f, 0.0f);
//Position des Lichts
PointLightRoom.Position = D3DXVECTOR3 (0.0f, 15.5f, 0.0f);
//8 Einheiten Reichweite
PointLightRoom.Range = 8.0f;
//keine lineare Lichtabschwächung
PointLightRoom.Attenuation0 = 0.0f;
```

```
//quadratische Lichtabschwächung
PointLightRoom.Attenuation1 = 0.35f;
//keine exponentielle Lichtabschwächung
PointLightRoom.Attenuation2 = 0.0f;

// Licht einsetzen und aktivieren
g_pD3DDevice->SetLight (0, &PointLightRoom);
g_pD3DDevice->LightEnable (0, TRUE);

//das Richtungslicht für die Stadt wird erstellt
ZeroMemory (&PointLightSun, sizeof (D3DLIGHT9));


 //Richtungslicht
 PointLightSun.Type = D3DLIGHT_DIRECTIONAL;
 // Streufarbe
 PointLightSun.Diffuse = D3DXCOLOR (0.45f, 0.4f, 0.4f, 0.0f);
 //Hintergrundfarbe
 PointLightSun.Ambient = D3DXCOLOR (0.45f, 0.4f, 0.4f, 0.0f);
 //Glanzfarbe
 PointLightSun.Specular  = D3DXCOLOR (0.45f, 0.4f, 0.4f, 0.0f);
 //Richtung des Lichts
 PointLightSun.Direction = D3DXVECTOR3 (0.0f, 0.0f, -1.0f);
 //beim Richtungslicht sind die Elemente Position, Reichweite
 //und Lichtabschwächung nicht relevant

 g_pD3DDevice->SetLight (1, &PointLightSun);
 g_pD3DDevice->LightEnable (1, TRUE);

 // Globales schwaches Hintergrundlicht einstellen
 g_pD3DDevice->SetRenderState(D3DRS_AMBIENT,
                         D3DXCOLOR (0.1f, 0.01f, 0.0f, 0.0f));
```

5.3.4 3D-Modell zeichnen

In der Initialisierungsfunktion der Szene wurde das 3D-Modell bereits geladen und alle relevanten Informationen in entsprechende Strukturen eingelesen. Jetzt muss es nur noch gezeichnet werden. Dazu werden alle Materialien des Modells in einer Schleife durchlaufen und dabei jeweils alle Vertices gezeichnet, die ein Material verwendet. Es müssen nur die zugehörigen Materialien und Texturen mit den Methoden *SetMaterial()* und *SetTexture()* gesetzt werden. Das eigentliche Zeichnen erfolgt dann durch die Methode *DrawSubset()*.

Quellcode:

```
//**************************************************
//das 3D-Modell zeichnen
//**************************************************

//City zeichnen
for (DWORD i=0; i<Scene.dwNumMaterials; i++)
{
   // Material und Textur für Device einstellen
   g_pD3DDevice->SetMaterial(&Scene.pMeshMaterials[i]);
   g_pD3DDevice->SetTexture(0, Scene.pMeshTextures[i]);

    // Rendern
   Scene.pMesh->DrawSubset(i);
}
```

5.3.5 Alpha-Blending

In der Grafikanwendung wurde die Spiegelung in den Fenstern durch Alpha-Blending simuliert. Dazu wurde eine Textur generiert, die den Raum aus der Blickrichtung der Fenster zeigt. Außerdem wurde für die Textur das TGA-Format verwendet, da es Platz für einen Alpha-Wert bietet. Der Alpha-Wert der Textur wurde in einem Bildbearbeitungsprogramm gesetzt. In Maya wurde daraufhin eine Fensterfront modelliert, der diese Textur zugewiesen wurde. In der Grafikapplikation musste das 3D-Modell für die Fensterfront dann einfach nur noch mit aktiviertem Alpha-Blending geladen werden.

Wenn das Render-State D3DRS_ALPHAENABLE auf TRUE gesetzt wird, wird bei allen zu zeichnenden Polygonen Alpha-Blending angewendet, bis D3DRS_ALPHAENABLE wieder auf FALSE gesetzt wird. Wie die Blendfaktoren für den neuen und den alten Pixel zu berechnen sind, bestimmen die Render-States

- D3DRS_SRCBLEND: Blendfaktor des neuen Pixels
- und D3DRS_DESTBLEND: Blendfaktor des alten Pixels

Das Render-State D3DRS_BLENDOP bestimmt, welche Verknüpfungsoperation auf den alten und den neuen Pixel angewendet wird.

Beim Alpha-Blending kann der Z-Buffer nur begrenzt eingesetzt werden, denn er ist dafür verantwortlich, dass immer nur der vorderste Pixel gezeichnet wird. Beim Alpha-Blending spielen beim Rendern der Szene aber auch die dahinter liegenden Pixel noch eine wichtige Rolle. Deshalb wird eine Szene, die Alpha-Blending verwendet, auf folgende Weise gezeichnet:

1. Leeren des Bildpuffers und des Z-Buffers.

2. Alle undurchsichtigen Objekte werden ohne Alpha-Blending gezeichnet. Das Culling und das Z-Buffering sind aktiviert.

3. Anschließend werden die durchsichtigen Objekte mit aktiviertem Alpha-Blending von hinten nach vorne gezeichnet. Der Z-Test bleibt dabei aktiviert, das Schreiben in den Z-Buffer und das Culling werden aber ausgeschaltet.

Quellcode

```
//FensterReihe zeichnen
for (i=0; i<Window.dwNumMaterials; i++)
{
   //Alpha-Blending einschalten
   g_pD3DDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, TRUE);
   g_pD3DDevice->SetRenderState (D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
   g_pD3DDevice->SetRenderState (D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
   g_pD3DDevice->SetRenderState (D3DRS_BLENDOP, D3DBLENDOP_ADD);

   g_pD3DDevice->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);
   g_pD3DDevice->SetRenderState (D3DRS_CULLMODE, D3DCULL_NONE);

   // Material und Textur für Device einstellen
   g_pD3DDevice->SetMaterial(&Window.pMeshMaterials[i]);
   g_pD3DDevice->SetTexture(0, Window.pMeshTextures[i]);

   // Rendern
   Window.pMesh->DrawSubset(i);

   //Alpha-Blending wieder ausschalten
   g_pD3DDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, FALSE);
   g_pD3DDevice->SetRenderState(D3DRS_ZWRITEENABLE, TRUE);

}//for (i=0; i<Window.dwNumMaterials; i++)
```

5.3.6 Multi-Texturing

Bei der Simulation des Wassers wurde ebenfalls mit Alpha-Blending gearbeitet. Außerdem wurde aber auch die Texturkoordinaten-Transformation und das Multi-Texturing angewendet.

Beim Multi-Texturing ist es möglich mehrere Texturen übereinander auf ein und dasselbe Polygon zu legen, wobei es nicht notwendig ist, dass die einzelnen Texturen hierfür dieselbe Auflösung haben. Jede der Texturen wird mit der Methode *SetTexture()* in eine andere Texturschicht eingefügt. Sogenannte Texturschicht-States legen dann fest, wie die einzelnen Texturschichten miteinander verknüpft werden.

Die Texturschicht-States beginnen alle mit D3DTTS_ und werden mit der Methode *SetTextureStageState()* gesetzt. Der erste Parameter dieser Methode gibt die Texturschicht an, der zweite Parameter gibt das Texturschicht-State an, das gesetzt werden soll und der dritte Parameter gibt schließlich den Wert an, auf den es gesetzt wird.

Die Texturschicht-States sollten nicht mit den Sampler-State verwechselt werden. Die Sampler-States bestimmen, ob Texturfilter u.ä. zur Anwendung kommen sollen oder nicht, während die Texturschicht-States hauptsächlich beim Multi-Texturing eingesetzt werden, um das "Zusammenspiel" der verschiedenen Texturschichten zu definieren.

Ein Vertex kann für jede Texturschicht ein anderes Paar Texturkoordinaten besitzen, so dass durch unterschiedliche Transformation dieser Texturkoordinaten sogar Animationen möglich sind. Dabei hat jede Texturschicht einen Operator und zwei oder drei Argumente, die durch den Operator verknüpft werden. Das Ergebnis dieser Texturschicht kann dann von der nächsten Texturschicht sogar wieder als Argument verwendet werden. Die Operatoren und die Argumente der Texturschichten gibt es nicht nur für die Farbkanäle, sondern auch für die Alpha-Kanäle.

Das Texturschicht-State D3DTSS_COLOROP bzw. D3DTSS_ALPHAOP bestimmt den Operator einer Texturschicht. Die Farbargumente werden mit den Texturschicht-States D3DTSS_COLORARG1, D3DTSS_COLORARG2 und D3DTSS_COLORARG0 festgelegt. D3DTSS_ALPHAARG1, D3DTSS_ALPHAARG2 und D3DTSS_ALPHAARG0 sind die entsprechenden Versionen für den Alphakanal.

Vor der Texturkoordinaten-Transformation wird **Direct3D** mitgeteilt, wie viele Koordinatenwerte transformiert werden sollen. Bei den üblichen Texturen sind es meistens zwei Texturkoordinaten. In diesem Fall wird das Texturschicht-State D3DTSS_TEXTURETRANSFORMFLAGS auf den Wert D3DTTFF_COUNT2 gesetzt.

Die Texturkoordinaten werden, wie alles andere auch, mittels Matrizen transformiert. 4x4-Matrizen eignen sich allerdings nicht dazu zweidimensionale Texturkoordinaten zu transformieren. Zu diesem Zweck sollte eine Funktion geschrieben werden, die eine 2D-Version einer Matrix erstellt, so dass diese sich für die Texturkoordinaten-Transformation eignet.

Die beiden Wassertexturen wurden bereits in der Initialisierungsfunktion der Szene geladen. Mit der Methode *SetTexture()* werden sie in die zweite und dritte Texturschicht gesetzt. Da die Texturen transformiert werden sollen, um das Wasser besser simulieren zu können, muss die Texturkoordinaten-Transformation für diese beiden Texturschichten aktiviert werden.

Quellcode

```
//Transformationsmatrix für die Wassertextur
D3DXMATRIX mWaterTexture;
//Welt-Transformationsmatrix
D3DXMATRIX mWorld;

ZeroMemory(&mWaterTexture, sizeof(D3DXMATRIX));
ZeroMemory(&mWorld, sizeof(D3DXMATRIX));

//2D-Texturkoordinatentransformation aktivieren
g_pD3DDevice->SetTextureStageState (1,
                    D3DTSS_TEXTURETRANSFORMFLAGS,
                    D3DTTFF_COUNT2);
g_pD3DDevice->SetTextureStageState (2,
                    D3DTSS_TEXTURETRANSFORMFLAGS,
                    D3DTTFF_COUNT2);

//eine Texturmatrix für die zweite Texturschicht generieren
//sie soll ein langsame Rotation der Textur bewirken
D3DXMatrixRotationX (&mWorld, g_fAnimTime * 0.02f);

//eine 2D-Version der Rotationsmatrix erstellen
mWaterTexture = MatrixToTex2DMatrix (mWorld);

//Texturkoordinaten-Transformation in der zweiten Texturschicht
g_pD3DDevice->SetTransform (D3DTS_TEXTURE1, &mWaterTexture);


//eine Texturmatrix für die dritte Texturschicht generieren
//sie soll eine noch langsamere Rotation in die Gegenrichtung bewirken
D3DXMatrixRotationX (&mWorld, g_fAnimTime * -0.01f);

//eine 2D-Version der Rotationsmatrix erstellen
mWaterTexture = MatrixToTex2DMatrix (mWorld);

//Texturkoordinaten-Transformation in der dritten Texturschicht
g_pD3DDevice->SetTransform (D3DTS_TEXTURE2, &mWaterTexture);
```

```
//die zwei Texturen einsetzten
g_pD3DDevice->SetTexture (1, g_apWaterTexture [0]);
g_pD3DDevice->SetTexture (2, g_apWaterTexture [1]);

//es sollen die Texturkoordinaten der ersten Texturschicht
//verwendet werden
g_pD3DDevice->SetTextureStageState (1,
                    D3DTSS_TEXCOORDINDEX, 0);
g_pD3DDevice->SetTextureStageState (2,
                    D3DTSS_TEXCOORDINDEX, 0);

//in der zweiten Texturschicht wird
//die Streufarbe aus Material mit Textur addiert
g_pD3DDevice->SetTextureStageState (1,
                    D3DTSS_COLOROP, D3DTOP_ADD);
g_pD3DDevice->SetTextureStageState (1,
                    D3DTSS_COLORARG1, D3DTA_CURRENT);
g_pD3DDevice->SetTextureStageState (1,
                    D3DTSS_COLORARG2, D3DTA_TEXTURE);

//in der dritten Texturschicht werden die Argumente multipliziert
g_pD3DDevice->SetTextureStageState (2,
                    D3DTSS_COLOROP, D3DTOP_MODULATE);
g_pD3DDevice->SetTextureStageState (2,
                    D3DTSS_COLORARG1, D3DTA_CURRENT);
g_pD3DDevice->SetTextureStageState (2,
                    D3DTSS_COLORARG2, D3DTA_TEXTURE);
```

Nachdem alle notwendigen Einstellungen für den gewünschten Effekt vorgenommen wurden, wird das Polygon für das Wasser gezeichnet. Danach sollte die Texturkoordinaten-Transformation wieder deaktiviert werden. Für die Erstellung der Wolken wurde übrigens eine ähnliche Vorgehensweise gewählt, weshalb sie nicht mehr explizit hier erklärt wird.

Quellcode

```
g_pD3DDevice->SetTextureStageState (1,
                    D3DTSS_TEXTURETRANSFORMFLAGS,
                    D3DTTFF_DISABLE);
g_pD3DDevice->SetTextureStageState (2,
                    D3DTSS_TEXTURETRANSFORMFLAGS,
                    D3DTTFF_DISABLE);
```

5.3.7 Volumentexturen zeichnen

Das Feuer im Kamin soll durch eine Volumentextur dargestellt werden, in der die einzelnen Phasen der brennenden Flamme abgespeichert sind. Damit das Feuer

animiert erscheint, wird die dritte Texturkoordinate der Volumentextur als Zeitachse verwendet. Ist man am Ende der w-Achse angelangt, wird sie einfach wieder von vorne durchlaufen.

Da die Volumentextur dreidimensional ist, muss zuerst wieder das entsprechende Vertex-Format gesetzt werden. Anschließend kann die Volumentextur, die in der Initialisierungsfunktion der Szene bereits in das Programm geladen wurde, in die erste Texturschicht gesetzt werden. Danach wird ein Viereck gezeichnet, auf dem später die Animation in Form der Volumentextur ablaufen soll. Die dritte Texturkoordinate des Vierecks ist dabei die Zeitachse, die angibt, welche Schicht der Volumentextur gezeichnet werden soll. Gibt die w-Achse dabei einen Wert zwischen zwei Schichten an, dann wird zwischen diesen beiden Schichten interpoliert und das Ergebnis gerendert. Die in der Grafikapplikation verwendete Volumentextur besitzt 37 verschiedene Schichten, so dass entlang der w-Achse in kleinen Schritten vorangegangen wird.

Das Viereck mit der eingesetzten Volumentextur wird dann schließlich durch die Methode *DrawPrimitiveUP()* gezeichnet, die für das Zeichnen verschiedener Primitivtypen zuständig ist. Mit dieser Methode können folgende Primitivtypen gezeichnet werden:

- Dreieckslisten: D3DPT_TRIANGLELIST
- verbundene Dreiecke: D3DPT_TRIANGLESTRIP
- Dreiecksfächer: D3DPT_TRIANGLEFAN
- Linienlisten: D3DPT_LINELIST
- verbundene Listen: D3DPT_LINESTRIP
- Punktlisten: D3DPT_POINTLIST

Quellcode

```
//*****************************************************
//das Kaminfeuer zeichnen
//*****************************************************
g_pD3DDevice->SetFVF (D3DFVF_XYZ |
                D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE3(0));

g_pD3DDevice->SetTexture (0, g_pVolumeTexture);

// Das Viereck erstellen, auf dem die Animation abgespielt wird.
// Die w-Texturkoordinate bestimmt dabei,
// wie weit die Animation fortgeschritten ist.
// Zwischen 0 und 1 wird sie genau einmal abgespielt,
// hinter 1 wiederholt sie sich.
```

```
g_aFireVertexBack[0].vPosition =
                D3DXVECTOR3 (-3.797f, 10.179f, -3.082f);
g_aFireVertexBack[1].vPosition =
                D3DXVECTOR3 (-3.797f,  11.279f, -3.082f);
g_aFireVertexBack[2].vPosition =
                D3DXVECTOR3 ( -3.092f, 10.179f, -3.782f);
g_aFireVertexBack[3].vPosition =
                D3DXVECTOR3 ( -3.092f,  11.279f, -3.782f);
g_aFireVertexBack[0].vTexture =
          D3DXVECTOR3 (0.0f, 1.0f, g_fAnimTime * 3 * 0.03333f);
g_aFireVertexBack[1].vTexture =
          D3DXVECTOR3 (0.0f, 0.0f, g_fAnimTime * 3 * 0.03333f);
g_aFireVertexBack[2].vTexture =
          D3DXVECTOR3 (1.0f, 1.0f, g_fAnimTime * 3 * 0.03333f);
g_aFireVertexBack[3].vTexture =
          D3DXVECTOR3 (1.0f, 0.0f, g_fAnimTime * 3 * 0.03333f);

// Nun das Viereck als Dreiecksfolge zeichnen (so brauchen wir nur 4 Vertizes)
if(FAILED(hResult =
   g_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLESTRIP,
                                 2,
                                 g_aFireVertexBack,
                                 sizeof(SSkyBoxVertex))))
{
   // Fehler beim Zeichnen!
   WriteToLog ("&#8226; Zeichnen der Volumentextur fehlgeschlagen!<br>");
   return 1;
}
```

5.3.8 Nebel

Die Nebeleinstellungen werden in **Direct3D** durch spezielle Render-States kontrolliert. Deshalb wurde der Nebel nicht in der *Render()*-Funktion, sondern in der Initialisierungsfunktion der Szene generiert.

In **Direct3D** gibt es verschiedene Nebeltypen:

- Linearer Nebel: hat eine feste Start- und Enddistanz – dazwischen wird sein Einfluss zunehmend (linear) größer

- Exponentieller Nebel: hat keine Anfangs- und Enddistanz, sondern beginnt direkt bei der Kamera und reicht unendlich weit – die Nebeldichte bestimmt, wie stark der Nebeleinfluss bei steigender Entfernung zunimmt

Zusätzlich unterteilt sich der exponentielle Nebel in den einfachen exponentiellen Nebel und in den quadratischen exponentiellen Nebel.

Das Render-State D3DRS_FOGENABLE aktiviert den Nebel, wenn es auf TRUE gesetzt wird. Die Art des Nebels wird mit dem Render-State

D3DRS_FOGVERTEXMODE gesetzt. Die Farbe des Nebels bestimmt wiederum das Render-State D3DRS_FOGCOLOR.

Quellcode

```
//**************************************************************
//Render-States für Nebel
//**************************************************************
//Dichte des Nebels (für exponentiellen Nebel)
float fFogDensity = 0.015f;

//Nebel einschalten
g_pD3DDevice->SetRenderState (D3DRS_FOGENABLE, TRUE);

//quadratischer exponentieller Nebel soll verwendet werden
g_pD3DDevice->SetRenderState (D3DRS_FOGVERTEXMODE, D3DFOG_EXP2);

//Farbe des Nebels bestimmen
g_pD3DDevice->SetRenderState (D3DRS_FOGCOLOR,
                              D3DCOLOR_XRGB (203, 241, 255));

//Nebeldichte setzen
g_pD3DDevice->SetRenderState (D3DRS_FOGDENSITY,
                              *((DWORD*) (&fFogDensity)));
```

5.4 Sound

Weiter oben wurde der theoretische Umgang mit **DirectSound** bereits beschrieben. Hier soll nun noch die praktische Umsetzung, die in diesem Programm erfolgt ist, vorgestellt werden.

Quellcode

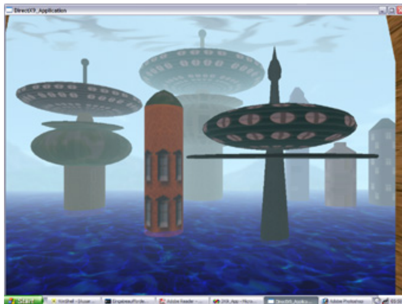
```
//**************************************************************
//Initialisierung von DirectSound
//**************************************************************
DSBUFFERDESC BufferDesc;
HWND window;

if (FAILED (window = FindWindow ("Direct3D Window", "DirectX9_Application")))
   WriteToLog ("&#8226; Window-Handle nicht gefunden! <br>");
```

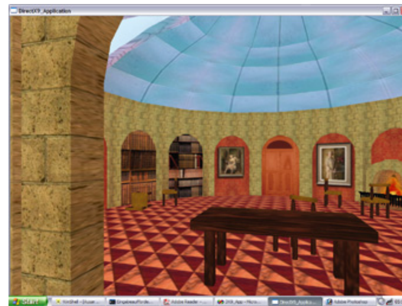
```
// DirectSound-Schnittstelle generieren - Standardgerät verwenden
if(FAILED(DirectSoundCreate8(&DSDEVID_DefaultPlayback, &g_pDSound, NULL)))
{
    // Fehler!
    WriteToLog ("&#8226; DirectSoundCreate8 () ist fehlgeschlagen!<br>");
    return 1;
}

// Kooperationsebene setzen
g_pDSound->SetCooperativeLevel(window, DSSCL_PRIORITY);



//**********************************************************************
//Erstellung des primären SoundBuffers
//**********************************************************************
// DSBUFFERDESC-Struktur für den primären Soundpuffer ausfüllen
ZeroMemory (&BufferDesc, sizeof(DSBUFFERDESC));

BufferDesc.dwSize           = sizeof(DSBUFFERDESC);
BufferDesc.dwFlags          = DSBCAPS_PRIMARYBUFFER;
BufferDesc.dwBufferBytes    = 0;
BufferDesc.dwReserved       = 0;
BufferDesc.lpwfxFormat      = NULL;
BufferDesc.guid3DAlgorithm  = GUID_NULL;

// Primären Soundpuffer erstellen
if(FAILED(g_pDSound->CreateSoundBuffer(&BufferDesc, &g_pPrimaryBuffer, NULL)))
{
    // Fehler!
    WriteToLog ("&#8226; Create PrimaryBuffer ist fehlgeschlagen<br>");
    return 1;
}



//************************************************************
//Laden einer WAV-Datei für das Meeresrauschen
//************************************************************
//LoadWAVFile() ist eine Hilfsfunktion
//für das Einlesen einer WAV-Datei
//ihr werden Flags für die Eigenschaften übergeben,
//die der Programmierer selbst steuern möchte
LoadWAVFile ("Welle.wav",
    DSBCAPS_LOCDEFER | DSBCAPS_CTRLVOLUME | DSBCAPS_CTRLPAN |
    DSBCAPS_CTRLFREQUENCY | DSBCAPS_GLOBALFOCUS ,
    &g_pSoundSea);
```

```
g_pSoundSea->SetVolume (-1000L);
g_pSoundSea->SetPan (0L);
g_pSoundSea->Stop ();
g_pSoundSea->SetCurrentPosition (0);
g_pSoundSea->Play(0, 0, DSBPLAY_LOOPING);
```

5.5 Ergebnisbilder

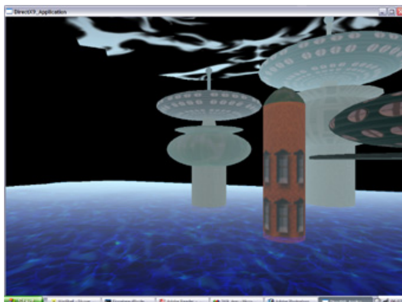
verschiedene Ansichten der fertigen 3D-Welt





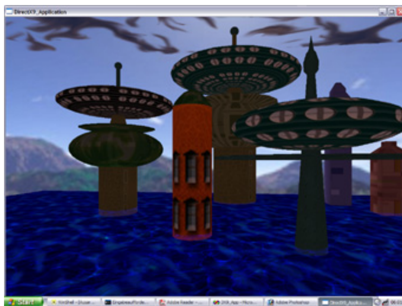
Screenshots, in denen einige Effekte ausgeschaltet sind:

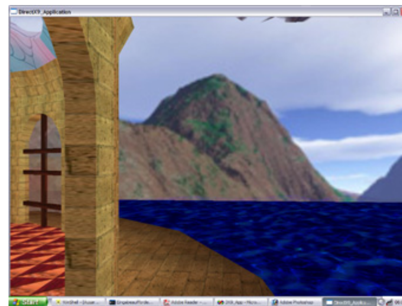
3D-Welt mit deaktivierter Sky-Box



3D-Welt mit deaktiviertem Licht

3D-Welt mit deaktiviertem Nebel





3D-Welt mit deaktiviertem Alpha-Blending

3D-Welt mit aktiviertem Alpha-Blending und Fensterspiegelung

6 Ausblick

Es gibt noch viele Dinge, die ich in dieser Studienarbeit gerne realisiert hätte, da DirectX 9 sehr viele interessante Möglichkeiten bietet.

Mein nächster Schritt wäre gewesen eine Kollisionserkennung und Schatten in die Grafikanwendung einzubauen. Aufgrund der aufwendigen Einarbeitung in DirectX und Maya, ist die Zeit für die Implementierung des Programms aber leider sehr knapp ausgefallen.

Auch könnten die Sachen, die in der Anwendung bereits realisiert wurden, weiter verfeinert werden. Man könnte z.B. die Kamera-Steuerung noch erweitern, so dass der Benutzer größere Bewegungsfreiheit innerhalb der Anwendung genießen kann. Die Beleuchtung könnte weiter ausgebaut und zusätzlich um Light-Maps erweitert werden. Für das Wasser könnte man ein bewegliches Gitternetz konstruieren, das die Bewegungen von Wasser nachahmt, so dass die Simulation nicht mehr nur durch Texturen erfolgt. Die Volumentextur des Kaminfeuers könnte ebenfalls optimiert werden, indem bessere Texturen hierfür generiert werden.

Weiterhin hätte ich gern auch Interaktionsmöglichkeiten innerhalb der 3D-Welt für den Benutzer geschaffen. Man könnte Gegenstände einbauen, die erkundet werden oder sogar Personen, mit denen man kommunizieren kann.

Als Fazit ziehe ich für mich heraus, dass mit DirectX 9 und ausreichend Fantasie gute Grafikanwendungen erstellt werden können.

Literatur

[1] Keywan Mahintorabi, ”Maya – 3D-Grafik und 3D-Animation”, mitp-Verlag, 2003.

[2] David Scherfgen, ”3D-Spieleprogrammierung – Modernes Game Design mit DirectX 9 und C++”, Carl Hanser Verlag, 2003.

[3] Microsoft, ”DirectX Documentation for C++”

[4] Microsoft, ”MSDN Library Visual Studio 6.0a”

[5] Alias Wavefront, ”Maya online help”

[6] Marc Lewerentz, ”Erstellen einer 3D-Engine mit dem Common Files Framework von DirectX für das Rendern von Charakteranimationen in Echtzeit”, Studienarbeit, 2003.

[7] Matthias Merz, ”Spieleprogrammierung”, Schriftliche Ausarbeitung, 1999.

[8] Andre Garefrekes, ”3D Visualisierungstechniken”, Schriftliche Ausarbeitung, 2003.

[9] www.highend3d.com

[10] www.spieleprogrammierer.de

[11] www.gamedev.net

[12] www.gamasutra.com

[13] msdn.microsoft.com

[14] aliaswavefront.com