mitp

Susanne Wigard

inklusive CD

Spieleprogrammierung mit DirectX 11 und C++

Basiswissen: 3D-Mathematik, Physik, objektorientierte Programmierung in Spielen

Grafikprogrammierung: Rendern, Benutzeroberflächen, Shader, Postprocessing

Animation, Audio, Multiplayer-Spiele

Erste Schritte

Der allererste Schritt bei der Programmierung eines DirectX-Spiels besteht darin, die neueste Version des DirectX-SDK (**S**oftware **D**evelopment **K**it) von der Microsoft-Website herunterzuladen.

Um DirectX11 verwenden zu können, benötigen Sie eine moderne Grafikkarte sowie das Betriebssystem Windows Vista oder Windows 7. Ältere Windows-Versionen (oder andere Betriebssysteme) unterstützen DirectX11 nicht.

Die meisten der Beispielprogramme in diesem Buch (mit Ausnahme derjenigen in Kapitel 17) laufen auch auf DX10-Grafikkarten – nur die Beispiele in Kapitel 17, die das Shader-Model 5 verwenden, setzen explizit eine DX11-Karte voraus. Wenn Sie diese Beispiele auf DX10-Hardware ausführen, wird automatisch das so genannte *Reference-Device* verwendet, das die Funktionen der DX11-Grafikkarte in Software emuliert. Sie können die Programme also auch damit testen, nur laufen diese dann extrem langsam.

1.1 Voraussetzungen und Installation

Bevor Sie mit der DirectX-Programmierung beginnen, sollten Sie Ihre Arbeitsumgebung einrichten. Neben dem DirectX-SDK (*Software Development Kit*) selbst benötigen Sie einen Compiler für die verwendete Programmiersprache (normalerweise C++) – besser noch eine integrierte Entwicklungsumgebung (*Integrated Development Environmet*: IDE).

1.1.1 Die Programmiersprache der Wahl: C++

Die Programmierschnittstelle DirectX besteht im Wesentlichen aus einem Satz von DLLs, deren Funktionen über *COM* (*Component Object Model*) angesprochen werden können. Im Prinzip ist COM unabhängig von einer bestimmten Programmiersprache. Es gibt daher eine Reihe von Ansätzen, DirectX auch in anderen Sprachen als C++ zu verwenden. Insbesondere auch für die Verwendung mit Microsoft .Net existieren Anbindungen (*Managed DirectX*, *XNA*). Wenn Sie beispielsweise an der Programmierung in C# interessiert sind, sollten Sie sich unbedingt einmal XNA ansehen (`http://creators.xna.com`). Dabei handelt es sich jedoch um ein komplett eigenes Framework, das zwar auf DirectX aufsetzt, dieses aber in eigene Funktionen verpackt. Teilweise ist die Programmierung damit komfortabler, allerdings auch weniger flexibel.

Für andere Sprachen existieren COM-Wrapper, die den Aufruf der DirectX-Funktionen aus der jeweiligen Sprache heraus ermöglichen. Das DirectX-SDK selbst ist aber klar auf die Programmierung mit der Sprache C++ ausgerichtet. Kommerzielle Spieleentwicklung erfolgt bis heute fast ausschließlich in C++. Auch alle Beispielprogramme des SDK sind in dieser Sprache geschrieben – wenn Sie eine andere Sprache verwenden wollen,

sind Sie weitgehend auf sich gestellt. In diesem Buch wird daher die DirectX-Programmierung in C++ beschrieben.

1.1.2 C++-Entwicklungsumgebung: Microsoft Visual C++

Für die Kompilierung Ihrer Programme können Sie im Prinzip jeden beliebigen C++-Compiler verwenden. Im Windows-Umfeld bietet sich Microsofts Visual Studio an.

Express Edition

Die kostenlos verfügbare Express Edition ist zwar eingeschränkt im Hinblick auf die Fähigkeiten zur Entwicklung in großen Programmierteams, bietet aber dem Hobby-Entwickler alles Notwendige für komfortables Editieren, Kompilieren und Debuggen der eigenen Programme. Die Beispiele aus dem DirectX-SDK sowie auch alle Beispielprogramme auf der Buch-CD lassen sich problemlos mit der Express Edition kompilieren, wobei die Beispiele auf der Buch-CD mit der Version Visual Studio 2008 entwickelt und getestet wurden. Mit gegebenenfalls später erscheinenden neueren Versionen sollten diese Beispiele ebenfalls kompilierbar sein, nicht jedoch mit älteren Versionen (zumindest nicht ohne manuelles Ändern der Versionsnummer in der Projektdatei).

Die jeweils aktuelle Version der Express Edition können Sie herunterladen unter `http://www.microsoft.com/germany/express/download/default.aspx`.

Kostenpflichtige Editionen

Wenn Sie eine der verschiedenen leistungsfähigeren kostenpflichtigen Editionen von Visual Studio besitzen, können Sie diese ebenfalls verwenden.

Windows SDK

Das Windows Software Development KIT (*Windows SDK*), das Sie für die Beispielprogramme ebenfalls benötigen, ist Bestandteil der Visual-Studio-Distribution (auch bei der Express Edition) und braucht daher nicht separat installiert zu werden.

Andere Entwicklungsumgebungen

Andere Entwicklungsumgebungen können in der Regel die Visual-Studio-Projektdateien (sowohl aus dem DirectX-SDK als auch von der Buch-CD) nicht öffnen, so dass Sie selbst entsprechende Projekte anlegen und die Quelltextdateien hinzufügen sowie für das Linken aller benötigten Bibliotheken Sorge tragen müssen. Diese Vorgehensweise ist für Anfänger nicht zu empfehlen – Sie sollten sich dazu schon relativ gut mit der verwendeten Entwicklungsumgebung und deren Compiler- und Linkereinstellungen (oder Makefiles) auskennen. Als Einsteiger arbeiten Sie am besten mit Visual Studio und verwenden die dafür bereitgestellten Projektdateien.

1.1.3 Das DirectX-SDK

Das DirectX-SDK können Sie von der Microsoft-Website kostenlos herunterladen. Die jeweils neueste Version finden Sie hier:

```
http://msdn.microsoft.com/en-us/directx/aa937788.aspx.
```

1.1.4 Der Sample Browser

Nach der Installation des DirectX-SDK finden Sie im Startmenü Einträge für die Dokumentation des SDK sowie verschiedene Tools. Das Programm DIRECTX SAMPLE BROWSER bietet Ihnen einen Überblick über alle mit dem SDK mitgelieferten Beispielprogramme. Diese Beispiele können auch direkt aus dem SDK heraus gestartet oder wahlweise in ein von Ihnen bestimmtes Projektverzeichnis kopiert und dort weiter bearbeitet werden. Auch die Dokumentation zu den Programmen können Sie aus dem Sample Browser heraus anzeigen.

Die SDK-Beispiele zeigen Ihnen die wichtigsten Techniken der DirectX-Programmierung. Es lohnt sich auch immer, nach dem Installieren einer neuen DirectX-Version einmal in den Sample Browser hineinzuschauen, denn gerade auch zu neuen Features kommen regelmäßig neue Beispiele hinzu.

1.1.5 SDK-Beispiele kompilieren und ausführen

Starten Sie nun einmal den Sample Browser. Klicken Sie auf den Link INSTALL PROJECT, um eins der Beispiele zu installieren. Starten Sie dann Visual Studio und öffnen Sie die Projektmappe (die SLN-Datei).

Erstellen und starten Sie das Programm.

Fehler beim Erstellen

Wenn Sie zuerst Visual Studio und dann DirectX installiert haben, sollte DirectX alle notwendigen Projekteinstellungen automatisch vorgenommen haben. Sollten Sie aber Visual Studio nachträglich installiert oder aber irgendwelche Projekteinstellungen von Hand verändert haben, so erhalten Sie möglicherweise einige Fehlermeldungen, dass Headerdateien nicht gefunden werden. In diesem Fall empfiehlt es sich, die im folgenden Abschnitt aufgelisteten Verzeichnispfade zu überprüfen.

Eventuell danach noch auftretende Linker-Fehler deuten auf eine fehlerhafte DirectX-Installation.

Visual Studio-Projekteinstellungen

Damit die Header-LIB- und DLL-Dateien für DirectX von Visual Studio gefunden werden, müssen die entsprechenden Pfade der Entwicklungsumgebung bekanntgegeben werden. Sollten also irgendwelche Dateien nicht gefunden werden, prüfen Sie die Einstellungen über den Menübefehl EXTRAS|OPTIONEN.

Im Dialogfeld OPTIONEN müssen unter PROJEKTE UND PROJEKTMAPPEN|VC++-VERZEICHNISSE die folgenden Einträge vorhanden sein:

Verzeichnisse anzeigen für:	
Includedateien	*SDK ROOT*\INCLUDE
Bibliotheksdateien	*SDK ROOT*\LIB\x86 oder *SDK ROOT*\LIB\x64 Je nach Betriebssystemversion

Dabei ist *SDK ROOT* der Pfad zu Ihrer DirectX-Installation, also z.B:

```
C:\Program Files\ Microsoft DirectX SDK (February 2010)
```

1.1.6 Externe Module

Zusätzlich zu den oben aufgeführten Voraussetzungen benötigen Sie für die Scripting-Beispiele in Kapitel 16 auch noch die Programmiersprachen Python bzw. Lua. Näheres zu Download und Installation finden Sie dort.

1.2 Bestandteile eines DirectX-Programms

Bevor wir nun unser erstes eigenes Beispiel zusammenbauen, sollen einige wichtige Begriffe der 3D-Grafikprogrammierung angeschnitten und diejenigen Elemente kurz vorgestellt werden, die sozusagen die Minimalausstattung eines Spiels darstellen.

1.2.1 Modelle

Die Szene, in der der Spieler herumläuft, sowie auch alle Spielfiguren (*Characters*) werden bekanntlich in einem 3D-Spiel nicht wie bei einem Zeichentrickfilm Bild für Bild von Menschen gezeichnet, sondern liegen als dreidimensionale Modelle vor, die aus einzelnen Vielecken (*Polygonen*) zusammengesetzt sind (in der Regel aus Dreiecken). Die Polygone wiederum werden durch Angabe ihrer Eckpunkte definiert, so dass jedes Modell letztlich im Rechner durch eine Liste von Punkten dargestellt wird zusammen mit der Information, wie diese Punkte zu Dreiecken zusammengesetzt werden. Die Grafikkarte berechnet aus dieser Liste von Punkten und Dreiecken dann die zweidimensionale Darstellung auf dem Bildschirm unter Berücksichtigung des Blickwinkels, der Beleuchtung usw.

Hinweis

In Anlehnung an die Anfänge der 3D-Grafik, als beim Zeichnen eines Modells nur die Kanten der Dreiecke als Linien gezeichnet wurden, bezeichnet man ein solches Modell auch als *Mesh* (deutsch »Netz«), weil das Modell in dieser Darstellung (die auch als *Drahtgittermodus* bezeichnet wird und bis heute in manchen Situationen noch nützlich ist) eben wie ein Gitternetz aus Drähten aussieht (siehe Abbildung 1.6). Wenn man von einem Modell als einem »Mesh« spricht, sind damit meist ausschließlich die Punkte und Dreiecke gemeint, während zu dem Modell als Ganzem natürlich noch weitere Informationen (insbesondere über Materialien, aber auch beispielsweise Animationen) gehören, die ebenfalls mit in der Modelldatei gespeichert sein können.

Man könnte nun die Modelle komplett im Programmcode definieren, indem man nacheinander die Koordinaten aller Eckpunkte festlegt. Für einfache Geometrien wie Dreiecke, Rechtecke, Würfel usw. ist das zweifellos noch machbar, und auch die Polygone einer Kugel oder sogar eines Terrains kann man mit etwas Überlegung komplett vom Programm berechnen lassen (siehe Kapitel 5). Sicherlich gibt es aber keinen Program-

mierer, der auf Anhieb die Koordinaten aller Dreiecke angeben könnte, aus denen beispielsweise ein Hubschrauber besteht. Solche Modelle werden im Normalfall mit Hilfe von *Modellierungsprogrammen* erstellt. Modellierungsprogramme ermöglichen es, komplexe Modelle aus einfachen Grundkörpern zusammenzusetzen, diese mit einer Vielzahl von Werkzeugen zu verformen und anderweitig zu bearbeiten, und speichern schließlich das Ergebnis als eine Liste von Punkten und Dreiecken in einer Datei. Leider sind die Dateiformate solcher Programme keineswegs einheitlich, weshalb die erste Aufgabe bei der Programmierung eines Spiels immer darin besteht, zu entscheiden, welches Modelldateiformat man verwenden möchte, und dann den entsprechenden Programmcode für das Einlesen der Modelle zu schreiben.

Wir werden für unsere ersten Gehversuche das auch von den DirectX-Beispielen im Sample Browser verwendete `sdkmesh`-Dateiformat benutzen, weil das DirectX-SDK einige Modelle in diesem Dateiformat mitbringt und mit dem DXUT-Framework (Abschnitt 1.3.3) auch fertige Funktionen zum Einlesen dieser Modelle zur Verfügung stellt. In späteren Kapiteln werden auch andere Dateiformate diskutiert.

1.2.2 Materialien, Texturen und Beleuchtung

Durch die Angabe der Punkte und Dreiecke eines Modells legen wir lediglich dessen Form fest – wichtig für das Aussehen sind daneben aber auch die Farbe und andere Materialeigenschaften. So gibt es etwa durchsichtige, glühende oder reflektierende Objekte in einem Spiel. Durch das »Bekleben« eines Objektes mit einem Bild (einer *Textur*) wie mit einer Tapete kann man zudem Objekten, die eigentlich eine einfache Geometrie haben, eine komplexe Struktur verleihen – beispielsweise kann man einen Würfel durch Aufbringen eines Bildes mit Türen und Fenstern wie ein Haus aussehen lassen. Auch die Berechnung von Licht und Schatten spielt natürlich für die letztendliche Farbe jedes Bildpunktes eine wichtige Rolle.

Shader

Das Aufbringen von Texturen auf 3D-Modelle sowie auch die Berechnung aller anderen Materialeffekte, der Beleuchtung und sogar die Umrechnung der Punkte des Modells in zweidimensionale Bildschirmkoordinaten erfolgt bei DirectX11 komplett in eigenen Programmen, die auf der Grafikkarte ausgeführt werden, den so genannten *Shadern*. Shader werden in einer eigenen Programmiersprache *HLSL* (*High Level Shader Language*) geschrieben und separat von einem eigenen Compiler übersetzt.

Auch wenn Shader an sich ein eher fortgeschrittenes Thema sind und deshalb auch erst weiter hinten im Buch im Detail vorgestellt werden, kommen wir auch bei den einfachsten Beispielen nicht um die Verwendung eines Shaders herum.

1.2.3 Positionieren von Objekten und Kameraeinstellungen

Die perspektivische Darstellung einer dreidimensionalen Szene hängt nicht nur von der Position aller Punkte des Modells, sondern auch von der Position, Blickrichtung und dem Öffnungswinkel der Kamera ab, mit der diese Szene betrachtet wird. (Natürlich ist beim Rendern einer 3D-Grafik-Szene nirgends eine wirkliche Kamera beteiligt, aber die

Regeln, nach denen aus der dreidimensionalen virtuellen Welt ein zweidimensionales Bild entsteht, gleichen denen bei der Abbildung mit einer Kamera.)

Außerdem müssen die Modellkoordinaten entsprechend der Position, Größe und Orientierung des Objekts umgerechnet werden. Diese Umrechnungen erfolgen ebenfalls mit Hilfe von Standardfunktionen im Shader, aber wir müssen aus dem C++-Programm heraus bestimmte Informationen an ihn übergeben – nämlich genau die Position, Größe und Orientierung innerhalb der Szene sowie die Kameraeinstellungen.

Matrizen

Da die oben genannten Berechnungen mit Hilfe so genannter *Matrizen* recht effizient durchgeführt werden können, erfolgt auch die Übergabe an den Shader in Form solcher Matrizen.

Eine Matrix besteht einfach aus einer Reihe von Fließkommazahlen, die in einem rechteckigen Schema angeordnet werden. In den meisten Fällen verwenden wir Matrizen aus vier Zeilen und vier Spalten. Wie eine solche Liste von Zahlen die Position und Orientierung der Kamera oder eines Modells beschreibt, erfahren Sie in Abschnitt 2.1. Im vorliegenden Kapitel werden wir einfach eingebaute DirectX-Funktionen verwenden, die uns beispielsweise aus der Position und Blickrichtung der Kamera eine Matrix berechnen. Diese Matrix wird dann lediglich an den Shader weitergegeben. Auch für die Position, Größe und Orientierung der einzelnen Objekte in unserer Szene übergeben wir Matrizen an den Shader, die mit eingebauten Funktionen berechnet werden.

1.2.4 Der Render-Loop

Da in einem Spiel die Szene normalerweise nicht statisch ist, sondern sowohl die Kamera ihre Position ändert (weil der Spieler »herumläuft«) als auch die Objekte in der Szene (beispielsweise die Gegner) sich bewegen können, muss das ganze Bild viele Male in der Sekunde neu gezeichnet werden, damit wie bei einem Film das Auge des Betrachters die einzelnen Bilder nicht mehr als solche erkennen kann, sondern der Eindruck einer fortlaufenden Bewegung entsteht. Dieses wiederholte Zeichnen (Rendern) der Szene erfolgt in einer Schleife (dem *Render-Loop*), die das Herzstück jedes 3D-Grafikprogramms darstellt. In der Regel trennt man außerdem noch die Aktualisierung der Szene (z.B. das Neuberechnen der Positionen von bewegten Objekten) vom eigentlichen Rendern. Vor Beginn des Render-Loops müssen alle Objekte angelegt und nach dessen Ende alle Ressourcen wieder freigegeben werden. Somit ergeben sich die wichtigsten Schritte im Ablauf eines 3D-Programms, die sich auch in den Funktionsnamen vieler Grafik-Engines (einschließlich DirectX) widerspiegeln:

- Anlagen aller Objekte (Create)
- Aktualisieren der Szene (Update) und Zeichnen (Render) in einer Schleife
- Aufräumarbeiten (Destroy)

1.3 Das erste Beispiel: Ein Modell laden und anzeigen

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_01`.

Das Erstellen eines DirectX-Grafikprogramms erfordert eine ganze Menge Code, der immer wieder gleich ist und den Sie gerade beim Ausprobieren neuer Techniken nicht jedes Mal wieder werden eintippen wollen. Außerdem kann die schiere Menge, verbunden mit der teilweise recht kryptischen Notation, gerade zu Anfang leicht den Blick auf das Wesentliche verstellen. Für die ersten Gehversuche empfiehlt es sich daher, auf einem der Beispielprogramme EMPTYPROJECT oder SIMPLESAMPLE aus dem DirectX-SDK Sample Browser aufzubauen. Ersteres ist, wie der Name schon sagt, ein leeres DirectX-Projekt, das genau für diesen Zweck vorgesehen ist. Das SIMPLESAMPLE, das wir hier verwenden wollen, enthält noch etwas mehr fertigen Code, u.a. eine rudimentäre grafische Benutzeroberfläche in Form von einigen Buttons.

Im weiteren Verlauf dieses Buchs werden Sie all diesen Programmcode kennen und verstehen lernen und später auch durch eigenen ersetzen, zu Beginn können Sie aber das Grundgerüst einfach als gegeben hinnehmen und sich darauf konzentrieren, an welchen Stellen Sie etwas einfügen müssen, um schnell Ihr Ziel zu erreichen.

Hinweis

Am Ende des Abschnitts (in Abschnitt 1.3.6) finden Sie eine kurze Übersicht der notwendigen Schritte, auf die Sie bei späteren Projekten zurückgreifen können. Beim allerersten Lesen sollten Sie aber die folgenden Ausführungen sorgfältig durchgehen, da weder der Beispielcode noch die Liste allein selbsterklärend sein können. Wenn Sie schon Erfahrung mit 3D-Grafik haben, sollten Sie zumindest den Abschnitt 1.3.5 studieren – dort erfahren Sie, wie das Grundgerüst zu ergänzen ist, um ein 3D-Modell darzustellen.

1.3.1 SimpleSample11 aus dem Sample Browser installieren

Finden Sie also im Sample Browser (der nach der Installation des DirectX-SDK im Startmenü vorhanden sein sollte) das Beispiel `SimpleSample11` – am einfachsten, indem Sie den Programmnamen (als ein Wort geschrieben) oben in das Suchfeld eingeben. Klicken Sie wieder auf den Link INSTALL PROJECT. Im angezeigten Dialogfeld können Sie einen eigenen Namen für das installierte Projekt vergeben und den Installationspfad festlegen. Wenn Sie wollen, können Sie die Projektmappe und das Projekt (die VCPROJ-Datei) noch von Hand umbenennen. Öffnen Sie dann die Projektmappe in der Visual-Studio-Entwicklungsumgebung.

Wenn Sie die Projekteinstellungen, wie weiter vorn in Abschnitt 1.1.5 beschrieben, bereits vorgenommen haben (bzw. diese automatisch durch den DirectX-Installer vorgenommen wurden), sollte das Beispielprogramm ohne weitere Bearbeitung kompilierbar sein – mit einigen Warnungen, die Sie zunächst einmal ignorieren können. Nach dem Starten zeigt das Programm einen leeren, fliederfarbenen Bildschirm an.

Abb. 1.1: Das zunächst noch leere Beispielprogramm

Für die Beispiele auf der Buch-CD wurde zusätzlich zu den hier beschriebenen Schritten noch die Projektdatei umbenannt, damit ihr Name zu dem entsprechenden Kapitel passt (z.B. `Kap01_01.vcproj`).

Hinweis

Je nach Version des DXSDK sind eventuell von den Projektdateien zwei Versionen vorhanden – eine für Visual Studio 2005 und eine für Visual Studio 2008. Die Beispiele dieses Buchs liegen ausschließlich in der Version für VS 2008 vor – Sie sollten nach Möglichkeit ebenfalls diese Version (oder gegebenenfalls eine höhere) verwenden, zumal Visual Studio 2005 in der Express Edition seit Frühjahr 2009 offiziell nicht mehr verfügbar ist.

Das Programm soll jetzt so erweitert werden, dass es ein 3D-Modell (ein *Mesh*) lädt und anzeigt.

1.3.2 Das Grundgerüst SimpleSample

Bevor wir das SIMPLESAMPLE unserem Vorhaben entsprechend erweitern, werfen Sie einmal kurz einen Blick auf den vorhandenen Code.

Dateien

Nach der Installation aus dem Sample Browser enthält unser Projekt (neben den beiden Ordnern `DXUT` und `Resource Files`) vier Dateien.

Die CPP-Dateien mit dem Programmquelltext liegen direkt auf der obersten Ebene unterhalb des Projekts:

- Eine CPP-Datei, deren Name dem von Ihnen gewählten Projektnamen entspricht (z.B. `Kap01_01.cpp`).
- Eine weitere CPP-Datei mit dem Namen `SimpleSample9`.

Die letztere Datei enthält Code, den Sie nur benötigen, wenn Sie von Ihrem Programm gleichzeitig auch eine DirectX9-Version entwickeln möchten. Das entspricht durchaus der Vorgehensweise in der Praxis, da eine Spielentwicklungsfirma ja nicht davon ausgehen kann, dass alle ihre Kunden bereits über eine Grafikkarte verfügen, die die neueste DirectX-Version unterstützt.

Im Ordner `Shaders` sind außerdem Dateien mit Shadercode in der Sprache HLSL (siehe Kapitel 8) enthalten:

- Eine Effektdatei mit der Dateierweiterung `.fx` (`SimpleSample.fx`)
- Eine Datei mit der Dateierweiterung `.hlsl` (`SimpleSample.hlsl`)

Auch die FX-Datei, die für die Verwendung mit dem so genannten Effekt-Framework (Abschnitt 1.11) gedacht ist, wird in der ersten Version des DirectX-SDK von Februar 2010 nur von der DX9-Version des Programmcodes benutzt. Die DX11-Version verwaltet den Shader und die an diesen übergebenen Daten selbst ohne die Hilfe des Effekt-Frameworks. Sie verwendet die HLSL-Datei.

SimpleSample9.cpp und den entsprechenden Code entfernen

Da dies ein Buch über DirectX11 ist, wollen wir auf die DirectX9-Version verzichten. Entfernen Sie also die Datei `SimpleSample9.cpp` aus dem Projekt. (Sie können die Datei dabei ruhig löschen, sie wird nicht mehr benötigt.)

Wenn Sie nun das Programm neu erstellen, erleben Sie aber eine böse Überraschung: Der Linker beschwert sich über eine ganze Reihe nicht aufgelöster Symbole – nämlich genau all jene Funktionen, die in der Datei `SimpleSample9.cpp` definiert waren. Die andere, noch vorhandene CPP-Datei enthält nämlich noch die Funktionsprototypen dieser Funktionen – der Compiler beschwert sich deshalb nicht, wenn die Funktionen aufgerufen werden, der Linker sucht aber die dazugehörende Implementierung.

Entfernen Sie also den folgenden Codeblock aus der CPP-Datei:

```
extern bool CALLBACK IsD3D9DeviceAcceptable(D3DCAPS9* pCaps, D3DFORMAT
AdapterFormat, D3DFORMAT BackBufferFormat, bool bWindowed, void* pUserContext );
extern HRESULT CALLBACK OnD3D9CreateDevice( IDirect3DDevice9* pd3dDevice,
const D3DSURFACE_DESC* pBackBufferSurfaceDesc, void* pUserContext );
extern HRESULT CALLBACK OnD3D9ResetDevice( IDirect3DDevice9* pd3dDevice, const
D3DSURFACE_DESC* pBackBufferSurfaceDesc, void* pUserContext );
extern void CALLBACK OnD3D9FrameRender( IDirect3DDevice9* pd3dDevice, double
fTime, float fElapsedTime, void* pUserContext );
extern void CALLBACK OnD3D9LostDevice( void* pUserContext );
extern void CALLBACK OnD3D9DestroyDevice( void* pUserContext );
```

Außerdem müssen wir noch weiter unten einen Block aus der Funktion `wWinMain` entfernen:

```
DXUTSetCallbackD3D9DeviceAcceptable( IsD3D9DeviceAcceptable );
DXUTSetCallbackD3D9DeviceCreated( OnD3D9CreateDevice );
DXUTSetCallbackD3D9DeviceReset( OnD3D9ResetDevice );
DXUTSetCallbackD3D9DeviceLost( OnD3D9LostDevice );
DXUTSetCallbackD3D9DeviceDestroyed( OnD3D9DestroyDevice );
DXUTSetCallbackD3D9FrameRender( OnD3D9FrameRender );
```

Nun lässt sich das Programm wieder erstellen und ausführen.

Hinweis

Wenn Sie wollen, können Sie natürlich auch einfach die DX9-Datei im Projekt belassen und diese Codezeilen beibehalten.

WinMain

Wenn Sie schon einmal Windows-Anwendungen in C++ programmiert haben, wissen Sie, dass die Funktion `WinMain` (bzw. hier die UNICODE-Version davon, `wWinMain`) der Einstiegspunkt einer solchen Anwendung ist.

Fenstertitel

Bei unserem Beispielprogramm ruft `wWinMain` lediglich eine Reihe von Funktionen aus den Hilfsdateien im Ordner `DXUT` auf (Letztere werden weiter unten vorgestellt). Wenn Sie möchten, können Sie hier in der Zeile

```
DXUTCreateWindow( L"Kap01_01");
```

den Fenstertitel Ihres Programms ändern:

```
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPWSTR lpCmdLine, int nCmdShow)
{
    // Enable run-time memory check for debug builds.
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    // DXUT will create and use the best device (either D3D9 or D3D11)
    // that is available on the system depending on which D3D callbacks are
    // set below

    // Set DXUT callbacks
    DXUTSetCallbackMsgProc(MsgProc);
    DXUTSetCallbackKeyboard(OnKeyboard);
    DXUTSetCallbackFrameMove(OnFrameMove);
    DXUTSetCallbackDeviceChanging(ModifyDeviceSettings);
```

```
    DXUTSetCallbackD3D11DeviceAcceptable(IsD3D11DeviceAcceptable);
    DXUTSetCallbackD3D11DeviceCreated(OnD3D11CreateDevice);
    DXUTSetCallbackD3D11SwapChainResized(OnD3D11ResizedSwapChain);
    DXUTSetCallbackD3D11SwapChainReleasing( OnD3D11ReleasingSwapChain);
    DXUTSetCallbackD3D11DeviceDestroyed(OnD3D11DestroyDevice);
    DXUTSetCallbackD3D11FrameRender(OnD3D11FrameRender);

    InitApp();

    DXUTInit(true, true, NULL); // Parse the command line, show msgboxes on
    // error, no extra command line params
    DXUTSetCursorSettings(true, true); // Show the cursor and clip it when in
    // full screen
    DXUTCreateWindow(L"Ein Mesh laden");

    // Only require 10-level hardware
    DXUTCreateDevice(D3D_FEATURE_LEVEL_10_0, true, 640, 480);
    DXUTMainLoop(); // Enter into the DXUT ren  der loop

    return DXUTGetExitCode();
}
```

Listing 1.1: Die Funktion wWinMain

Hinweis

Die englischen Kommentare in obigem und einigen der folgenden Listings sind so aus dem Quelltext des Originalbeispiels im DirectX-SDK übernommen.

Rückruffunktionen

Hauptbestandteil der oben vorgestellten `wWinMain`-Funktion sind Aufrufe der `DXUTSetCallback...`-Funktionen.

Hiermit werden so genannte *Rückruffunktionen* (englisch *Callbacks*) festgelegt. Eine Rückruffunktion ist eine Funktion, die vom Programmierer der Anwendung bereitgestellt und vom System oder einem Framework (hier DXUT, siehe unten) in bestimmten Situationen automatisch aufgerufen wird. Ein Grundgerüst für diese Funktionen ist im SimpleSampleII bereits enthalten – wir müssen dieses gegebenenfalls nur noch um eigenen Code ergänzen.

Die wichtigsten vier Rückruffunktionen sind im obigen Listing fett gedruckt:

- `OnD3D11CreateDevice` soll aufgerufen werden, wenn das so genannte *Device* erstellt wurde (engl. *create*). Da die meisten Initialisierungen, die wir in unserem eigenen Code durchführen müssen, das Vorhandensein eines Device voraussetzen, ist **`OnD3D11CreateDevice`** der geeignete Ort für solche einmaligen Initialisierungen (z.B. das Laden von Modellen, Texturen etc.).
- `OnD3D11DestroyDevice` wird entsprechend beim »Zerstören« (engl. *destroy*) des Device aufgerufen. Hier sind also auch alle eigenen Aufräumarbeiten durchzuführen.

- `OnD3D11FrameRender` wird jedes Mal aufgerufen, wenn die Szene neu gezeichnet wird – also je nach Geschwindigkeit der Grafikkarte bis zu mehr als hundert Mal pro Sekunde. Hier wird auch unser eigener Code zum Zeichnen untergebracht.
- `OnFrameMove` schließlich wird jeweils vor dem Zeichnen aufgerufen. Hier können Sie also alle Aktualisierungen wie Bewegungen von Objekten, Physik etc. durchführen.

Hinweis

Die Namen der Rückruffunktionen sind im Prinzip egal, die hier genannten sind die im SIMPLESAMPLE verwendeten, aber wenn Sie wollen, können Sie die Funktionen auch umbenennen.

Hintergrundwissen: Referenzzählung

Ein in der C++-Programmierung häufig wiederkehrendes Thema ist die Speicherverwaltung: Wenn Speicherplatz für Objekte dynamisch (mit `new`) angefordert wird, muss dieser vom Programm auch explizit wieder freigegeben werden. Das wird dann zum Problem, wenn mehrere Objekte einen Zeiger auf dasselbe untergeordnete Objekt enthalten (zum Beispiel: Mehrere Modelle verwenden dasselbe Material). Wenn nun eins der Objekte den Zeiger freigibt, sind davon auch alle anderen betroffen. Allgemein ist nicht klar, welcher Programmteil für das Freigeben solcher gemeinsam genutzten Zeiger verantwortlich ist.

Eine Standardlösung dieses Problems besteht darin, dass man beim Kopieren einer Zeigervariablen nicht die Adresse kopiert, sondern den kompletten Inhalt (»tiefe Kopie«), damit der Zeiger eben *nicht* gemeinsam genutzt wird. In manchen Situationen wird auf diese Weise aber sehr viel Speicherplatz verschwendet, und gerade, wenn Objekte aus DLLs geladen werden, kann auch ein merklicher Performanceverlust entstehen. In einer 3D-Grafikanwendung ist ein solcher Performanceverlust natürlich nicht tolerierbar.

Eine andere Möglichkeit, die Frage des Freigebens von gemeinsam genutztem Speicher zu regeln, besteht in der so genannten *Referenzzählung*. Dabei hält jedes Objekt selbst nach, wie viele Zeigerverweise auf dieses Objekt bestehen. Die Clientobjekte (die einen Zeiger auf das Objekt verwenden) geben den Zeiger nicht frei, sondern teilen dem Objekt nur mit, dass sie diesen nicht mehr benötigen. Erst wenn das letzte Clientobjekt seinen Anspruch abgemeldet hat, entfernt das Objekt sich selbst aus dem Speicher. Auf diese Weise können Clientobjekte beliebig Zeiger gemeinsam nutzen (vorausgesetzt, bei jedem Kopieren eines solchen Zeigers wird der Referenzzähler des betreffenden Objekts um eins erhöht), und es wird kein Speicher freigegeben, der noch von anderen Objekten benötigt wird.

Die COM-Technologie, auf der auch DirectX basiert, realisiert Referenzzählung mittels der Methoden `AddRef` und `Release`. Sie dürfen daher für COM-Zeiger nicht `delete` aufrufen, sondern melden lediglich mit `Release`, wenn Sie das Objekt nicht mehr benötigen. Dies geschieht normalerweise in der Rückruffunktion `OnD3D11DestroyDevice`.

In unseren eigenen Klassen werden wir meist eine `Destroy`-Methode implementieren, die `Release` für alle verwendeten COM-Ressourcen aufruft.

Device und Device Context

Was müssen wir uns nun unter dem geheimnisvollen *Device* vorstellen, das man offenbar zum Zeichnen benötigt? Ganz allgemein versteht man unter einem Device ein Gerät – hier ein Ausgabegerät. Das Device-Objekt im DirectX-Programmcode stellt uns Funktionen zur Verfügung, mit denen wir das Zeichnen von 3D-Grafik kontrollieren. Letztlich ist das Device also die Repräsentation unserer Grafikkarte im Programm.

Device Context

Um das Rendern in mehreren Threads zu erleichtern, trennt DX11 im Gegensatz zu DX10 das Erstellen von Ressourcen und das Rendern in ein Device und einen (oder mehrere) *Device Context* (Gerätekontext).

Zum jedem Device gehört genau ein *Immediate Context*, der Render-Daten direkt an den Grafikkartentreiber schickt.

Hinweis

Eine Reihe von Funktionen, die in DX10 über das Device aufgerufen wurden, muss man in DX11 über den Device Context aufrufen, z.B. die Funktionen zum Zugriff auf die Input-Assembler-Stufe (`IAGetInputLayout`, `IASetInputLayout` usw.), auf die Output-Merger-Stufe (`OMGetDepthStencilState`, `OMSetDepthStencilState` usw.), auf die Rasterize-Stufe (`RSGetViewports`, `RSSetViewports` usw.) und die Stream-Output-Stufe (`SOGetTargets`, `SOSetTargets`), auf die Shader-Stufen (`VSGetShader`, `VSSetShader` usw.) und die Funktionen zum Rendern (`Draw`, `DrawIndexed`, `DrawAuto` usw.).

Rendern

Das Schreiben eines DirectX-Programms besteht nach dem bisher Erläuterten im Wesentlichen aus dem Ergänzen der im vorhergehenden Abschnitt vorgestellten Rückruffunktionen. Sehen Sie sich beispielsweise einmal das Grundgerüst von `OnD3D11FrameRender` im SIMPLESAMPLE an. Die Funktion bekommt einen Zeiger auf das Device-Objekt übergeben und verwendet ihn für verschiedene Zeichenvorgänge.

Hintergrundfarbe

Oben in der Funktion `OnD3D11FrameRender` finden Sie die Zeilen

```
float ClearColor[4] = { 0.176f, 0.196f, 0.667f, 0.0f };
ID3D11RenderTargetView* pRTV = DXUTGetD3D11RenderTargetView();
//...
pd3dImmediateContext->ClearRenderTargetView(pRTV, ClearColor);
```

Hier wird der Hintergrund des Programmfensters durch Ausfüllen mit einer bestimmten Farbe (im Beispiel Blau) gelöscht.

Ändern Sie die erste dieser Zeilen einmal in

```
float ClearColor[4] = {0.0f, 0.0f, 0.0f, 0.0f};
```

Der Hintergrund ist jetzt schwarz. Die vier `float`-Zahlen geben den Rot-, Grün- und Blau-Anteil sowie den Alpha-Wert (Durchsichtigkeit) einer Farbe jeweils als Zahl zwischen null und eins an. (Probieren Sie also auch einfach einmal andere Farben.)

Vorsicht

Generell ist es keine gute Idee, bei Tests die Szene mit schwarzem Hintergrund zu rendern: Wenn nämlich irgendetwas mit der Beleuchtung nicht stimmt, erscheinen die gezeichneten Objekte ebenfalls schwarz, so dass Sie nicht erkennen können, ob überhaupt irgendetwas gezeichnet wird. Setzen Sie deshalb die Hintergrundfarbe sicherheitshalber erst einmal wieder zurück.

1.3.3 DXUT

Die Dateien im bei allen Sample-Browser-Beispielen eingebundenen Ordner `DXUT` (für »DirectX Utilities«) enthalten Hilfsfunktionen und -klassen, die Sie auch für Ihre eigenen Projekte verwenden können. Da diese Funktionalitäten im Quelltext vorliegen, können Sie dabei sogar beliebige Anpassungen vornehmen oder den Code als Vorlage für Ihre eigenen Entwicklungen verwenden. Je weiter Sie fortschreiten, werden Sie dabei wahrscheinlich immer mehr davon auch durch eigene Programmteile ersetzen.

1.3.4 Modelle in einer Datei

Die in einer Szene in einem Spiel verwendeten Modelle (Gebäude, Fahrzeuge, Personen, Waffen usw.) werden in der Regel in einem separaten Modellierungsprogramm erstellt und in einer Datei gespeichert. Das Spiel liest diese Informationen aus der Datei und zeigt das Modell an.

Mediendaten

Medien (auch *Ressourcen* oder *Assets* genannt) wie Modelle, Texturen, Sounds etc. liegen normalerweise in separaten Dateien vor, die mit dem Spiel ausgeliefert werden. Am besten legen Sie für solche Ressourcen ein eigenes Unterverzeichnis `Media` in dem Projektverzeichnis des jeweiligen Beispiels an. Die DXUT-Funktion `DXUTFindDXSDKMediaFileCch`, die wir zum Ermitteln des tatsächlichen Speicherorts der Mediendateien verwenden werden (Abschnitt 1.4.1), findet Ressourcen dort auch ohne komplette Pfadangabe.

Hinweis

Da für die Beispiele in diesem Buch Mediendateien häufig mehrfach verwendet werden, liegt der `Media`-Ordner auf der Buch-CD nicht innerhalb des Projektverzeichnisses, sondern zwei Ebenen höher, auf der gleichen Ebene wie die Kapitelverzeichnisse. Auch hier werden die Dateien von `DXUTFindDXSDKMediaFileCch` gefunden.

Das Dateiformat SDKMesh

Während DirectX9 noch Utility-Klassen zum direkten Laden des weit verbreiteten »hauseigenen« `.x`-Dateiformats mitbrachte, fehlt in DirectX11 (Stand Anfang 2010) eine unmit-

telbare Unterstützung für dieses Format. Über die Frage, ob diese zu einem späteren Zeitpunkt noch »nachgerüstet« werden soll, wurde viel spekuliert, aber eine klare Antwort steht noch aus. Tatsächlich sind kommerzielle Dateiformate wie beispielsweise das `fbx`-Format für die Bedürfnisse moderner Spiele besser geeignet. Die mit den Beispielen aus dem Sample Browser installierten DXUT-Klassen unterstützen in der DXII-Version lediglich ein neueres Dateiformat mit der Dateierweiterung `.sdkmesh`. Während viele Modellierungsprogramme (gegebenenfalls mit Hilfe spezieller Plugins) in der Lage sind, `.x`-Dateien zu erzeugen, sucht man nach einer Exportmöglichkeit für `.sdkmesh` höchstwahrscheinlich vergeblich. Auch den vielen im Internet (z.B. unter `www.turbosquid.com`) frei erhältlichen Modelldateien im `.x`-Format oder in Formaten, die mit Standardtools (z.B. dem kostenlosen Modellierungsprogramm BLENDER) in das `.x`-Format konvertiert werden können, steht nichts Entsprechendes für das neue Format gegenüber.

Mit dem DirectX-SDK wird aber das Konvertierungsprogramm `MeshConvert.exe` ausgeliefert, mit dem Sie `.x`-Modelle in das Format `.sdkmesh` umwandeln können.

Hinweis

Beachten Sie, dass `MeshConvert.exe` die Dateierweiterung standardmäßig nicht ändert – die neue Datei hat also, obwohl sie nun das SDKMesh-Format aufweist, immer noch die Erweiterung `.x`. Sie können das ändern, indem Sie mit der Option `/o` den Ausgabedateinamen (einschließlich Erweiterung) festlegen.

Laut Kommentar im Originalquelltext ist das SDKMesh-Format nicht für Produktionszwecke gedacht, sondern ausschließlich für die Beispielprogramme – für echte DirectX-Spiele wird empfohlen, ein eigenes Dateiformat zu entwickeln. Auch das `.x`-Format ist ja mittlerweile etwas in die Jahre gekommen und genügt nicht mehr allen Ansprüchen moderner Grafikprogrammierung.

Wir werden auf das Thema eines eigenen Dateiformats in einem späteren Kapitel zurückkommen. Die in den Beispielen enthaltene Klasse `DXUTSDKMesh`, die das SDKMesh-Format verwendet, kann aber für solche Eigenentwicklungen durchaus als Muster dienen. Im Augenblick reicht sie für unsere Zwecke völlig aus und erspart Ihnen die Mühe, sich mit den technischen Einzelheiten beim Laden und Anzeigen eines Modells auseinanderzusetzen. Da die Klasse im Quelltext vorliegt, haben Sie aber jederzeit die Möglichkeit, dort nachzusehen, wenn Sie einmal wissen möchten, wie es funktioniert. Wenn Sie in den folgenden Kapiteln gelernt haben, mit Vertexbuffern und Indexbuffern umzugehen, werden Sie den entsprechenden Code auch leicht verstehen können.

Für den Moment müssen wir also mit der Unbequemlichkeit leben, unsere (vielleicht schon vorhandenen) `.x`-Modelle zu konvertieren – oder Sie verwenden einfach diejenigen aus dem DirectX-SDK und von der Buch-CD.

Tipp

Den Quelltext des Programms `MeshConvert.exe` finden Sie im DirectX-SDK im Ordner `Utilities\Source`.

1.3.5 Ergänzungen im Programmcode

Wenn Sie ein geeignetes Modell gefunden haben (oder Sie nehmen einfach das aus dem Beispielprogramm zum Buch), können wir uns daranmachen, die Codezeilen zum Laden und Zeichnen des Modells in das SIMPLESAMPLE-Grundgerüst einzufügen. Hier müssen Sie also erstmals eigenen Programmcode schreiben – das Laden eines Modells ist im SIMPLESAMPLE noch nicht eingebaut.

Oben in der CPP-Datei (bei den anderen Variablendeklarationen) fügen wir eine Variable für das Mesh ein:

```
CDXUTSDKMesh g_Mesh;
```

In `OnD3D11CreateDevice` (an der durch `// Create other render resources here` markierten Stelle) werden die DXII-Objekte initialisiert.

Das Modell wird geladen, indem wir die Methode `Create` des `SDKMesh`-Objekts aufrufen:

```
HRESULT hr;
//...
V_RETURN(g_Mesh.Create(pd3dDevice, L"earth\\earth.sdkmesh", true));
```

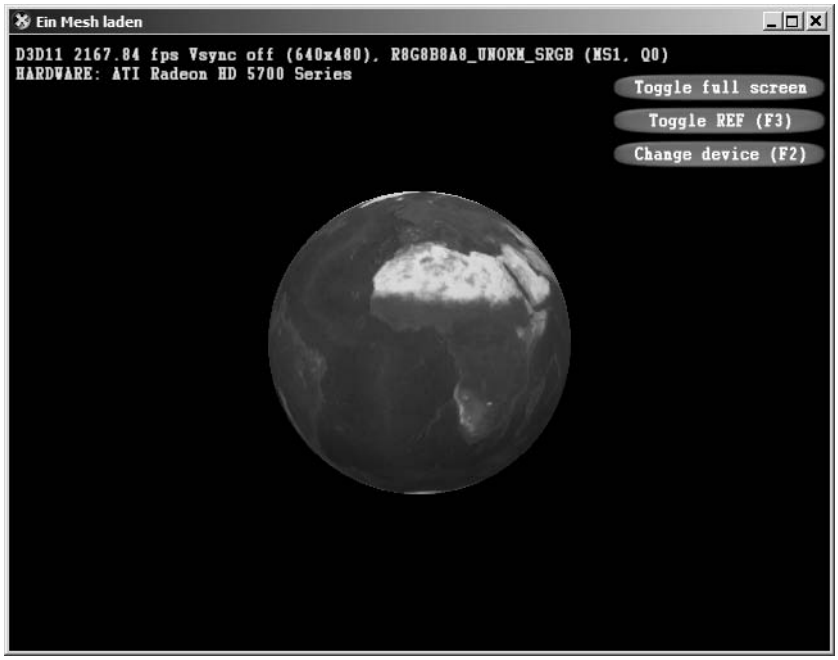

Abb. 1.2: Das gerenderte Modell

In `OnD3D11DestroyDevice` werden die verwendeten Objekte wieder aufgeräumt. Die `Destroy`-Methode der Klasse `SDKMesh` erledigt die Aufräumarbeiten für das Mesh:

```
g_Mesh.Destroy();
```

In der Funktion `OnD3D11FrameRender` schließlich wird das Modell einfach durch Aufruf der `Render`-Methode gezeichnet:

```
// Render objects here...
g_Mesh.Render(pd3dImmediateContext, 0, -1, -1);
```

Die Methode hat neben dem Device Context eine Reihe von Parametern, die angeben, ob eine oder mehrere Texturen, deren Namen in dem Modell gespeichert sind, verwendet werden oder ob Sie sich um das Bereitstellen der Textur(en) selbst kümmern. Neben der gewöhnlichen Textur kommen hier weitere für Bumpmapping und Glanzlichter in Frage, die wir aber ignorieren. Für den ersten dieser Parameter übergeben wir aber `0`, das heißt, wir überlassen alle Aufgaben im Zusammenhang mit der Textur der Klasse `CDXUTSDKMesh`.

Sie können das Programm jetzt bereits starten.

Mit dem Mausrad können Sie das Objekt heranzoomen, und bei gedrückter Maustaste können Sie es durch Bewegen der Maus drehen.

1.3.6 Die Schritte im Überblick (SimpleSample)

Zusammenfassend hier noch einmal eine Liste der notwendigen Schritte, um das SimpleSample-Grundgerüst so zu erweitern, dass ein Modell aus einer Datei angezeigt werden kann.

- SimpleSampleII aus dem Sample Browser installieren
- Eventuell Projektdatei umbenennen
- Gegebenenfalls `SimpleSample9.cpp` sowie den entsprechenden Programmcode entfernen
- Bei Bedarf in `wWinMain` den Fenstertitel anpassen
- Globale Variable für das Mesh deklarieren
- `OnD3D11CreateDevice` ergänzen
- `OnD3D11DestroyDevice` ergänzen
- `OnD3D11FrameRender` ergänzen
- Mediendateien (hier das Mesh) in den Ordner Media einfügen (falls nicht bereits in gemeinsamem Verzeichnis vorhanden)

1.4 Ein Modell in ein leeres Projekt einfügen

Um die verschiedenen Teile des im SimpleSample bereits vorhandenen Programmcodes genauer zu untersuchen, wollen wir nun ein ähnliches Beispiel einmal komplett von Anfang an aufbauen, ausgehend von der EmptyProject-Vorlage aus dem DirectX Sample Browser. Der einzufügende Programmcode ist identisch mit dem im SimpleSample, Sie können den Text also von dort kopieren, wenn Sie möchten. Ziel dieses Abschnitts ist lediglich, zu zeigen, welche Programmteile für welche Funktionalität gebraucht werden und jeweils zu erklären, wie diese Programmteile funktionieren.

Außerdem können Sie für die weiteren Beispiele darauf zurückkommen, wenn Sie mit einem leeren Projekt beginnen.

Installieren Sie also die Vorlage aus dem SDK heraus. Die vorbereitenden Schritte (Umbenennen der Projektdatei, Entfernen des DX9-Codes etc.) sind dieselben wie die beim SIMPLESAMPLE Dargestellten. Im Folgenden wird beschrieben, welche Ergänzungen im Einzelnen vorzunehmen sind.

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_02`.

1.4.1 Shader und Konstantenbuffer

Shader sind, wie bereits erwähnt, Programme, die nicht auf dem Hauptprozessor (CPU) ihres Rechners, sondern auf der Grafikkarte ausgeführt werden.

Im einfachsten Fall verwenden wir den Shader, um aus der Position von Punkten des Modells im 3-dimensionalen Raum in Abhängigkeit von der Position des Objekts, der Kamera und den Bildschirmabmessungen die Position der entsprechenden Bildpunkte (Pixel) auf dem Bildschirm zu berechnen. Außerdem kann auch die Farbe dieser Bildpunkte im Shader bestimmt werden.

Mit der dahinterliegenden Mathematik werden wir uns später befassen, und einen Shader selbst zu programmieren brauchen Sie auch noch nicht, da im SIMPLESAMPLE eine fertige Shaderdatei mit Vertex- und Pixelshader enthalten ist, die wir auch hier verwenden wollen. Sie können die Datei `SimpleSample.hlsl` einfach aus dem vorhergehenden Projekt kopieren. Wenn Sie wollen, können Sie auch einen Filter SHADERS in das neue Projekt einfügen und die Datei dort aufnehmen.

An dieser Stelle soll zunächst einmal der Programmcode zum Einlesen und Kompilieren der Shaderdatei sowie zum Setzen aller von diesem Shader benötigten Parameter in das Beispiel eingefügt werden.

Um zum Auffinden der Shaderdatei die DXUT-Funktion `DXUTFindDXSDKMediaFileCch` verwenden zu können (siehe unten), müssen wir die Datei `SDKmisc.h` inkludieren:

```
#include "SDKmisc.h"
```

Die dazugehörende CPP-Datei nehmen wir ins Projekt auf. Diese wiederum erfordert auch die Dateien `DXUTgui.cpp` und `DXUTsettingsdlg.cpp`.

Definieren Sie dann oben in der Quelltextdatei Ihres Hauptprogramms Variablen für Vertexshader und Pixelshader und die Konstantenbuffer:

```
ID3D11VertexShader*         g_pVertexShader11 = NULL;
ID3D11PixelShader*          g_pPixelShader11 = NULL;
```

Shaderparameter

An den Shader des SIMPLESAMPLE müssen (zunächst) die folgenden Parameter übergeben werden:

- Die *World-View-Projection-Matrix* `m_mWorldViewProjection` regelt die oben erwähnte Berechnung von Bildschirmpunkten aus den Eckpunkten der Dreiecke eines Modells (siehe Abschnitt 1.6).
- Die *World-Matrix* `m_mWorld` gibt die Position eines Objekts im Raum an (siehe Kapitel 2.1).
- Die Farbe des Materials bei Beleuchtung mit Umgebungslicht bzw. mit diffusem Streulicht werden in den Variablen `m_MaterialAmbientColor` und `m_MaterialDiffuseColor` übergeben. Näheres dazu erfahren Sie in Abschnitt 1.4.

Die Übergabe dieser Parameter erfolgt in so genannten *Konstantenbuffern* (*Constant Buffers*). Das sind Speicherbereiche, die vom C++-Code aus gefüllt und vom Shader ausgelesen werden. Das Beispielprogramm dieses Kapitels sowie auch die folgenden verwenden zwei solcher Buffer. Die dazu passenden Strukturen werden oben in der CPP-Datei deklariert.

Der Buffer `CB_VS_PER_OBJECT` fasst alle Informationen zusammen, die für jedes Objekt in der Szene neu berechnet werden müssen:

```
#pragma pack(push,1)
struct CB_VS_PER_OBJECT
{
    D3DXMATRIX  m_mWorldViewProjection;
    D3DXMATRIX  m_mWorld;
    D3DXVECTOR4 m_MaterialAmbientColor;
    D3DXVECTOR4 m_MaterialDiffuseColor;
};
```

Listing 1.2: Per-Object-Buffer

Der Buffer `CB_VS_PER_FRAME` dagegen muss nur einmal in jedem Frame übergeben werden. Er wird weiter unter in Abschnitt 1.4 vorgestellt.

```
struct CB_VS_PER_FRAME
{
    D3DXVECTOR3 m_vLightDir;
    float       m_fTime;
    D3DXVECTOR4 m_LightDiffuse;
};
#pragma pack(pop)
```

Listing 1.3: Per-Frame-Buffer

Hinweis

Der Parameter `m_fTime` wird in dem Shader des SIMPLESAMPLE nicht ausgewertet. Es ist dafür gedacht, die Anzahl der verstrichenen Sekunden an den Shader zu übermitteln, wenn man zeitabhängige Effekte realisieren möchte.

Außerdem deklarieren wir globale Variablen für die beiden Buffer:

```
ID3D11Buffer*                   g_pcbVSPerObject11 = NULL;
ID3D11Buffer*                   g_pcbVSPerFrame11 = NULL;
```

OnD3D11CreateDevice

In `OnD3D11CreateDevice` kompilieren wir wieder den Vertex- und Pixelshader und erzeugen die beiden Konstantenbuffer:

```
WCHAR str[MAX_PATH];
DXUTFindDXSDKMediaFileCch(str, MAX_PATH, L"SimpleSample.hlsl");

DWORD dwShaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;
#if defined(DEBUG) || defined(_DEBUG)
   dwShaderFlags |= D3D10_SHADER_DEBUG;
#endif
ID3DBlob* pVertexShaderBuffer = NULL;
D3DX11CompileFromFile(str, NULL, NULL, "RenderSceneVS", "vs_4_0_level_9_1",
   dwShaderFlags, 0, NULL, &pVertexShaderBuffer, NULL, NULL);
ID3DBlob* pPixelShaderBuffer = NULL;
D3DX11CompileFromFile(str, NULL, NULL, "RenderScenePS", "ps_4_0_level_9_1",
   dwShaderFlags, 0, NULL,&pPixelShaderBuffer, NULL, NULL);

pd3dDevice->CreateVertexShader(pVertexShaderBuffer->GetBufferPointer(),
   pVertexShaderBuffer->GetBufferSize(), NULL, &g_pVertexShader11);
pd3dDevice->CreatePixelShader(pPixelShaderBuffer->GetBufferPointer(),
   pPixelShaderBuffer->GetBufferSize(), NULL, &g_pPixelShader11);
```

Listing 1.4: Kompilieren der Shader

```
D3D11_BUFFER_DESC cbDesc;
ZeroMemory(&cbDesc, sizeof(cbDesc));
cbDesc.Usage = D3D11_USAGE_DYNAMIC;
cbDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
cbDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;

cbDesc.ByteWidth = sizeof(CB_VS_PER_OBJECT);
pd3dDevice->CreateBuffer(&cbDesc, NULL, &g_pcbVSPerObject11);

cbDesc.ByteWidth = sizeof(CB_VS_PER_FRAME);
pd3dDevice->CreateBuffer(&cbDesc, NULL, &g_pcbVSPerFrame11);
```

Listing 1.5: Anlegen der Konstantenbuffer

Hinweis

`DXUTFindDXSDKMediaFileCch` überprüft verschiedene Pfade, um Mediendateien wie z.B. Shaderdateien, 3D-Modelle oder Texturdateien zu finden – so brauchen Sie nicht immer den kompletten Pfad anzugeben, wenn Sie alle Ihre Medien in einem gemeinsamen Verzeichnis aufbewahren, das sich beispielsweise unterhalb des Ausführungsverzeichnisses oder auch wie bei den Beispielen auf der Buch-CD zwei Ebenen darüber befindet.

Auch Dateipfade zu Texturen in Modelldateien sollten nach Möglichkeit als relative Pfade (oder nur Dateinamen) exportiert werden, damit Sie die Verzeichnisstruktur selbst festlegen können, in der Sie Ihre Medien organisieren.

OnD3D11DestroyDevice

In `OnD3D11DestroyDevice` sollten nun noch die Variablen wieder aufgeräumt werden:

```
void CALLBACK OnD3D11DestroyDevice(void* pUserContext)
{
    SAFE_RELEASE(g_pVertexShader11);
    SAFE_RELEASE(g_pPixelShader11);

    //...

    SAFE_RELEASE(g_pcbVSPerObject11);
    SAFE_RELEASE(g_pcbVSPerFrame11);
}
```

Listing 1.6: In `OnD3D11DestroyDevice` einzufügender Code

OnD3D11FrameRender

In `OnD3D11FrameRender` wird der Buffer dann befüllt:

```
// Get the projection & view matrix from the camera class
D3DXMATRIX mWorld = *g_Camera.GetWorldMatrix();
D3DXMATRIX mView = *g_Camera.GetViewMatrix();
D3DXMATRIX mProj = *g_Camera.GetProjMatrix();
D3DXMATRIX mWorldViewProjection = mWorld * mView * mProj;

HRESULT hr;
D3D11_MAPPED_SUBRESOURCE MappedResource;
V(pd3dImmediateContext->Map(g_pcbVSPerObject11, 0, D3D11_MAP_WRITE_DISCARD, 0,
    &MappedResource));
CB_VS_PER_OBJECT* pVSPerObject = (CB_VS_PER_OBJECT*)MappedResource.pData;

//Matrizen
D3DXMatrixTranspose( &pVSPerObject->m_mWorldViewProjection, &mWorldViewProjection);
D3DXMatrixTranspose(&pVSPerObject->m_mWorld, &mWorld);

//Material
pVSPerObject->m_MaterialAmbientColor = D3DXVECTOR4(0.3f, 0.3f, 0.3f, 1.0f);
pVSPerObject->m_MaterialDiffuseColor = D3DXVECTOR4(0.7f, 0.7f, 0.7f, 1.0f);

pd3dImmediateContext->Unmap(g_pcbVSPerObject11, 0);

//Beleuchtung
pd3dImmediateContext->Map(g_pcbVSPerFrame11, 0, D3D11_MAP_WRITE_DISCARD, 0,
    &MappedResource);
CB_VS_PER_FRAME* pVSPerFrame = (CB_VS_PER_FRAME*)MappedResource.pData;
pVSPerFrame->m_vLightDir = D3DXVECTOR3(0,0.707f,-0.707f);
pVSPerFrame->m_LightDiffuse = D3DXVECTOR4(1.f, 1.f, 1.f, 1.f);
```

```
pd3dImmediateContext->Unmap(g_pcbVSPerFrame11, 0);

pd3dImmediateContext->VSSetConstantBuffers(0, 1, &g_pcbVSPerObject11);
pd3dImmediateContext->VSSetConstantBuffers(1, 1, &g_pcbVSPerFrame11);
```

Listing 1.7: Befüllen des Per-Object-Konstantenbuffers

Sie werden über die Einzelheiten dieses Programmcodes noch mehr erfahren – im Moment genügt es völlig, wenn Sie wissen, dass damit Informationen an den Shader übergeben werden. Für unsere ersten Beispiele werden wir diese Zeilen auch jeweils unverändert beibehalten. Auch die Shaderdatei wird in den folgenden Beispielen zunächst immer diejenige aus dem SIMPLESAMPLE sein.

Weitere Shaderparameter für Licht, Material und Textur werden weiter unten vorgestellt.

Nach dem Füllen der Buffer übergeben wir die Shader an den Device Context:

```
pd3dImmediateContext->VSSetShader(g_pVertexShader11, NULL, 0);
pd3dImmediateContext->PSSetShader(g_pPixelShader11, NULL, 0);
```

Listing 1.8: Übergeben der Shader an den Device Context

Linker-Einstellungen

Um den Shader kompilieren zu können, müssen Sie zusätzlich zu den bereits eingebundenen Bibliotheken auch noch die Datei `d3dcompiler.lib` in Ihr Projekt aufnehmen. Zeigen Sie dazu das Eigenschaftenfenster des Projekts an, wählen Sie in der Strukturansicht links den Eintrag KONFIGURATIONSEIGENSCHAFTEN|LINKER|EINGABE und tragen Sie die LIB-Datei unter ZUSÄTZLICHE ABHÄNGIGKEITEN ein.

1.4.2 Kamera

Wie bereits erwähnt, regeln die World-, View- und Projections-Matrizen, die beim Rendern an den Shader übergeben werden, die Umrechnung der Positionsinformationen jedes Vertex in die letztendliche Positionen von Pixeln auf dem Bildschirm.

Genauer ausgedrückt, legt die *World-Matrix* die Position, Rotation und Skalierung des Objekts fest. (Es wird also von seinem eigenen lokalen Koordinatensystem in das Weltkoordinatensystem transformiert, siehe Abschnitt 2.1.)

Die *View-Matrix* berücksichtigt dann Position und Blickrichtung des Betrachters (z.B. erscheint das Objekt größer, wenn der Betrachter näher herangeht, und kleiner, wenn er sich entfernt).

Mittels der Projection-Matrix schließlich werden die dreidimensionalen Koordinaten auf den zweidimensionalen Bildschirm projiziert, wobei festzulegen ist, welcher Ausschnitt der Welt abgebildet werden soll, und natürlich auch die Abmessungen des Bildschirmausschnitts berücksichtigt werden müssen, in den wir hineinzeichnen.

Die durch die View- und Projection-Matrizen dargestellten Informationen spielen genauso auch beim Abbilden einer Szene mit einer Kamera eine Rolle: Auch hier muss die Position und Blickrichtung der Kamera festgelegt werden, und der Öffnungswinkel des Objektivs bestimmt die Größe des Bereichs, der auf dem Bild zu sehen sein wird (z.B. zeigt bei einem Weitwinkelobjektiv ein Bild gleicher Größe mehr von der Welt als bei

einem Normalobjektiv). Schließlich wird durch das Seitenverhältnis des Bildes und diesen Öffnungswinkel auch der vertikale Ausschnitt aus der Welt, der auf dem Bild zu sehen sein wird, bereits festgelegt, wenn das Bild nicht verzerrt sein soll.

Es ist deshalb üblich, auch in der 3D-Grafik von einer Kamera zu sprechen, durch die die Szene betrachtet wird. Veränderungen an den Kameraeinstellungen bewirken eine Veränderung des sichtbaren Bildes. Aus Programmierersicht ist eine Kamera lediglich eine Zusammenfassung der View- und Projection-Matrix.

DXUT stellt einige solcher Kameraklassen zur Verfügung, die es uns bequem ermöglichen, aus der Position und Blickrichtung des Beobachters die View-Matrix und aus dem Öffnungswinkel sowie dem Seitenverhältnis des Bildschirmausschnitts die Projection-Matrix zu berechnen.

Dabei ist auch gleich noch eine einfache Steuerung der Kamera mit Tastatur und Maus vorgesehen: Je nach Kameratyp kann die Kamera z.B. mittels der Pfeiltasten durch die Szene gesteuert und mit der Maus die Blickrichtung gedreht werden.

Kommerzielle Spiele verwenden ähnliche, teilweise aber auch noch weit aufwändigere Kamerasteuerungen.

Verwendung

Um Ihr Programm mit einer Kamera auszustatten, müssen Sie zunächst einmal oben in die Datei Ihres Hauptprogramms eine `include`-Direktive für die entsprechende Headerdatei einfügen:

```
#include "DXUTCamera.h"
```

Die dazugehörende Implementierungsdatei `DXUTCamera.cpp` wird ins Projekt eingebunden. Die Kamera benötigt außerdem noch die Quelltextdatei `DXUTres.cpp`, die wir ebenfalls aufnehmen.

Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf den Ordner `DXUT` und wählen Sie aus dem Kontextmenü den Befehl HINZUFÜGEN|VORHANDENES ELEMENT. Sie finden die beiden CPP-Dateien im Verzeichnis `DXUT/Optional` innerhalb Ihres Projektverzeichnisses. Wenn Sie möchten, können Sie die Headerdateien ebenfalls mit ins Projekt aufnehmen – das ist aber nicht unbedingt notwendig.

Variablen

Definieren Sie dann eine globale Variable für die Kamera:

```
CModelViewerCamera g_Camera;
```

OnD3D11CreateDevice

In `OnD3D11CreateDevice` wird die View-Matrix der Kamera initialisiert:

```
HRESULT CALLBACK OnD3D11CreateDevice(ID3D11Device* pd3dDevice,
    const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
    void* pUserContext)
{
```

```
    //...
    D3DXVECTOR3 vecEye(0.0f, 0.0f, -5.0f);
    D3DXVECTOR3 vecAt (0.0f, 0.0f, -0.0f);
    g_Camera.SetViewParams(&vecEye, &vecAt);

    return S_OK;
}
```

Listing 1.9: `OnD3D11CreateDevice`

OnD3D11ResizedSwapChain

Das Einstellen der Projektionsparameter wird in `OnD3D11ResizedSwapChain` durchgeführt:

```
HRESULT CALLBACK OnD3D11ResizedSwapChain(ID3D11Device* pd3dDevice,
    IDXGISwapChain *pSwapChain, const DXGI_SURFACE_DESC* pBackBufferSurfaceDesc,
    void* pUserContext)
{
    float fAspectRatio = pBackBufferSurfaceDesc->Width /
    (FLOAT)pBackBufferSurfaceDesc->Height;
    g_Camera.SetProjParams(D3DX_PI / 4, fAspectRatio, 0.1f, 1000.0f);
    g_Camera.SetWindow(pBackBufferSurfaceDesc->Width,
        pBackBufferSurfaceDesc->Height);
    g_Camera.SetButtonMasks(MOUSE_LEFT_BUTTON, MOUSE_WHEEL,
    MOUSE_MIDDLE_BUTTON);

    return S_OK;
}
```

Listing 1.10: `OnD3D11ResizedSwapChain`

OnFrameMove

In `OnFrameMove` wird die Kamera aktualisiert:

```
void CALLBACK OnFrameMove(double fTime, float fElapsedTime, void* pUserContext)
{
    g_Camera.FrameMove(fElapsedTime);
}
```

Listing 1.11: `OnFrameMove`

MsgProc

Neben dem Aufruf von `Update` müssen Sie auch noch in `MsgProc` der Kamera Gelegenheit geben, auf Tastatur- und Mausnachrichten zu reagieren:

```
LRESULT CALLBACK MsgProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam,
    bool* pbNoFurtherProcessing, void* pUserContext)
{
    g_Camera.HandleMessages(hWnd, uMsg, wParam, lParam);
```

```
    return 0;
}
```

Listing 1.12: `MsgProc`

OnD3D11FrameRender

In `OnD3D11FrameRender` schließlich zeichnen wir die Szene, wobei die View- und Projection-Matrizen der Kamera verwendet werden (Listing 1.18). Andere Elemente der Konstantenbuffer wie Licht- und Materialeigenschaften können dabei natürlich genauso festgelegt werden (siehe Abschnitt 1.4). Bevor dieser Programmcode lauffähig ist, müssen außerdem noch die Konstantenbuffer selbst und ein Shader eingefügt und initialisiert werden, wie weiter vorn beschrieben.

1.4.3 Einfügen eines Modells

Sie können nun noch in Ihr Programm ein Modell einfügen (vgl. Abschnitt 1.3.5), und das Beispiel dann ausprobieren.

Denken Sie aber daran, dass Sie eine `include`-Anweisung für die Datei `SDKmesh.h` einfügen:

```
#include "SDKmesh.h"
```

Die dazugehörende CPP-Datei wird ins Projekt aufgenommen.

Vertexlayout

Um der Grafikkarte mitzuteilen, wie genau die Punkte (Vertices) in unserem Modell aufgebaut sind (ob sie z.B. nur Positionswerte enthalten oder darüber hinaus weitere Informationen wie die Farbe), benötigen wir außerdem ein Inputlayout (auch Vertexlayout genannt). Da dieses vom verwendeten Modell und den Materialien (letztlich vom Shader) abhängt, ist es nicht in der SDKMesh-Klasse enthalten, sondern muss in unserem eigenen Code erstellt werden.

Das SIMPLESAMPLE-Beispiel enthält (oben in der CPP-Datei) bereits eine entsprechende Variable, in das EMPTYPROJECT-Beispiel müssen wir diese aber noch einfügen:

```
ID3D11InputLayout*          g_pLayout11 = NULL;
```

Auch den Programmcode zum Anlegen eines für unsere Zwecke geeigneten Vertexlayouts müssen wir in `OnD3D11CreateDevice` einfügen:

```
const D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,  0,
    D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL",   0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
    D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT,    0, 24,
    D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

```
V_RETURN(pd3dDevice->CreateInputLayout(layout, ARRAYSIZE(layout),
    pVertexShaderBuffer->GetBufferPointer(),
    pVertexShaderBuffer->GetBufferSize(),
    &g_pLayout11));
```

Listing 1.13: Anlegen des Vertexlayouts

In `OnD3D11FrameRender` muss vor dem Zeichnen des Mesh-Objekts das Inputlayout dem Device zugewiesen werden. Auch dieser Code ist im SIMPLESAMPLE bereits vorhanden, muss im EMPTYPROJECT aber noch ergänzt werden:

```
pd3dImmediateContext->IASetInputLayout( g_pLayout11 );
```

In `OnD3D11DestroyDevice` wird dann auch das Vertexlayout wieder freigegeben:

```
SAFE_RELEASE(g_pLayout11);
```

Das Makro `SAFE_RELEASE` ruft `Release` nur dann auf, wenn der Zeiger nicht `NULL` ist, und setzt diesen danach auf `NULL`. Es ist in `DXUT.h` wie folgt definiert:

```
#define SAFE_RELEASE(p)      { if (p) { (p)->Release(); (p)=NULL; } }
```

Sampler State

Wenn das Modell eine Textur hat, benötigen Sie evtl. zusätzlich auch noch einen Sampler State (siehe Abschnitt 1.4.7). Falls Sie diesen auf `NULL` setzen, wird eine Voreinstellung verwendet. Diese ist zwar hier auch brauchbar, allerdings erhalten Sie bei der Ausführung im Debug-Modus eine ganze Reihe Warnmeldungen im Ausgabefenster Ihrer Entwicklungsumgebung.

1.4.4 Die Schritte im Überblick (EmptyProject)

So wie in Abschnitt 1.3.6 bereits für das SIMPLESAMPLE geschehen, sollen hier noch einmal zusammengefasst die Schritte aufgelistet werden, die notwendig sind, um in einem auf dem EMPTYPROJECT aufbauenden Beispielprogramm ein Modell anzuzeigen und mit einer Kamera zu betrachten:

- EMPTYPROJECT11 aus dem Sample Browser installieren
- Die Visual-Studio-2005-Version der Projektdatei (`vcproj`) kann entfernt werden
- Gegebenenfalls `EmptyProject9.cpp` sowie den entsprechenden Programmcode entfernen
- Gegebenenfalls in `wWinMain` den Fenstertitel anpassen
- Für das Mesh `SDKMesh.cpp` und `SDKmisc.cpp` und außerdem `DXUTres.cpp`, `DXUTgui.cpp` und `DXUTsettingsdlg.cpp` sowie für die Kamera `DXUTcamera.cpp` ins Projekt aufnehmen, `SDKMesh.h`, `SDKmisc.h` sowie `DXUTcamera.h` inkludieren
- Shaderdatei anlegen (kopieren, ggf. umbenennen) und ins Projekt aufnehmen
- `d3dcompiler.lib` als Linker-Eingabe zum Projekt hinzufügen

- Globale Variablen deklarieren
- Rückruffunktionen ergänzen:
  - `OnD3D11CreateDevice`
  - `OnD3D11ResizedSwapChain`
  - `OnFrameMove`
  - `MsgProc`
  - `OnD3D11DestroyDevice`
  - `OnD3D11FrameRender`
- gegebenenfalls Mediendateien (hier das Mesh) in den Ordner `Media` einfügen

1.4.5 Beleuchtung

Dass unser Modell im oben beschriebenen Beispiel nicht komplett schwarz erscheint, liegt daran, dass das SimpleSample bereits Informationen über Lichtverhältnisse und Materialeigenschaften an den Shader übergibt. Wir werden diesen Programmcode nun anpassen, um unsere eigenen Beleuchtungs- und Materialeinstellungen zu verwenden.

Bei der Beleuchtung eines Modells mit einer gerichteten Lichtquelle werden diejenigen Teile stärker beleuchtet, die der Lichtquelle zugewandt sind. Unser Konstantenbuffer (die Struktur `CB_VS_PER_FRAME`, Listing 1.3) enthält deshalb Elemente sowohl für die Richtung (`m_vLightDir`) als auch die Farbe (`m_LightDiffuse`) dieser gerichteten Komponente.

Das Anlegen des Konstantenbuffers mit den Per-Frame-Daten wurde bereits in Abschnitt 1.4.1 gezeigt.

OnD3D11FrameRender

Auch der Programmcode zum Befüllen der Lichtparameter ist in `OnD3D11FrameRender` bereits vorhanden:

```
HRESULT hr;
D3D11_MAPPED_SUBRESOURCE MappedResource;
pd3dImmediateContext->Map(g_pcbVSPerFrame11, 0, D3D11_MAP_WRITE_DISCARD, 0,
   &MappedResource);
CB_VS_PER_FRAME* pVSPerFrame = (CB_VS_PER_FRAME*)MappedResource.pData;
```

- Das Licht soll schräg von vorn und oben einfallen:

```
    pVSPerFrame->m_vLightDir = D3DXVECTOR3(0,0.707f,-0.707f);
```

- Die Farbe des gerichteten Lichtes ist Weiß:

```
    pVSPerFrame->m_LightDiffuse = D3DXVECTOR4(1.f, 1.f, 1.f, 1.f);
    pd3dImmediateContext->Unmap( g_pcbVSPerFrame11, 0 );
    pd3dImmediateContext->VSSetConstantBuffers(1, 1,
    &g_pcbVSPerFrame11);
```

Probieren Sie nun einmal die folgenden Änderungen:

```
pVSPerFrame->m_vLightDir = D3DXVECTOR3(1.0f,0.0f,0.0f);
pVSPerFrame->m_LightDiffuse = D3DXVECTOR4(1.0f, 0.0f, 0.0f, 1.0f);
```

Hinweis

Da die Werte hier alle konstant sind, könnte man die Zeilen für das Setzen der Konstantenbuffer auch in `OnD3D11CreateDevice` statt in `OnD3D11FrameRender` einfügen, so dass die Zuweisung nur einmal erfolgt und nicht jedes Mal, wenn das Bild neu gezeichnet wird. In einem wirklichen Spiel ändern sich die Werte vieler Shadervariablen aber dynamisch. Manche dieser Variablen müssen in jedem Frame (also für jedes gezeichnete Einzelbild) gesetzt werden, andere (wie etwa die Materialfarbe) können sogar für jedes einzelne Objekt andere Werte bekommen usw. Der geeignete Zeitpunkt zum Setzen dieser Variablen ist also durchaus nicht immer einheitlich.

1.4.6 Materialeigenschaften

Genau genommen müssen wir auch bei den Materialeigenschaften unterscheiden zwischen der Farbe, in der das Material bei Beleuchtung mit ungerichtetem Umgebungslicht (Ambient Color) erscheint, und der Farbe, in der es sich im diffusen Streulicht zeigt (Diffuse Color). Zudem zeigen glatte Materialien (z.B. Plastik) bei Beleuchtung aus der Blickrichtung so genannte *Glanzlichter* (*Specular Reflections*), die eine deutlich hellere Farbe haben oder sogar weiß sind (Specular Color).

Konstantenbuffer

Die Struktur `CB_VS_PER_OBJECT` (Listing 1.2) enthält passende Elemente für die Ambient- und die Diffuse-Farbe.

Die Variable `m_MaterialAmbientColor` wird für die Farbe des Materials bei Beleuchtung mit Umgebungslicht verwendet. Dabei berücksichtigen wir hier erst einmal nicht die eigene Farbe dieses Umgebungslichts. Die Ambient-Farbe des Materials bestimmt also allein die Helligkeit der vom Licht abgewandten Seite unserer Modelle. Die Farbe bei Beleuchtung mit dem Richtungslicht dagegen ergibt sich aus der Kombination von `m_MaterialDiffuseColor` mit der Lichtfarbe.

Glanzlichter kommen bei Verwendung des Standardshaders aus dem SIMPLE SAMPLE nicht vor, wir können sie daher hier zunächst außer Acht lassen.

OnD3D11FrameRender

Auch der Code zum Setzen der Materialfarbe ist in `OnD3D11FrameRender` bereits vorhanden:

```
pVSPerObject->m_MaterialAmbientColor = D3DXVECTOR4( 0.3f, 0.3f, 0.3f, 1.0f );
pVSPerObject->m_MaterialDiffuseColor = D3DXVECTOR4( 0.7f, 0.7f, 0.7f, 1.0f );
```

Probieren Sie beispielsweise die folgende Änderung der Ambient-Farbe:

```
pVSPerObject->m_MaterialAmbientColor = D3DXVECTOR4(0.0f, 0.0f, 1.0f, 1.0f);
```

Abb. 1.3: Modell mit Textur und Beleuchtung

Änderung in der Shaderdatei

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_03`.

Zwar sieht unser Modell mit der Textur schon recht gut aus, die Wirkung der Beleuchtung könnte man ohne sie aber vielleicht besser ausprobieren. Dazu müssen Sie im Pixelshader eine kleine Änderung vornehmen:

Finden Sie in der HLSL-Datei die Zeile

```
return g_txDiffuse.Sample(g_samLinear, In.TextureUV) *  In.Diffuse;
```

und ersetzen Sie sie durch

```
return In.Diffuse;
```

Statt die diffuse Farbe mit der Farbe aus der Textur zu multiplizieren, geben wir Erstere also direkt als Ergebnis zurück.

Die genauere Bedeutung dieses, in der Shadersprache HLSL geschriebenen Codes wird in Kapitel 8 diskutiert. Das Ergebnis ist das gewünschte: Unser Modell wird nun farbig dargestellt, wobei die unbeleuchtete Seite blau erscheint, während auf der beleuchteten Seite die Materialfarbe und die Farbe des Lichts gemischt werden.

Wenn Sie wollen, können Sie nun die Hintergrundfarbe wieder auf Schwarz setzen, wie weiter oben beschrieben. Experimentieren Sie auch mit den Farben von Licht und Material sowie der Lichteinfallsrichtung. (Abbildung 1.4 zeigt ein blaues Material, beleuchtet von rechts mit rotem Licht.)

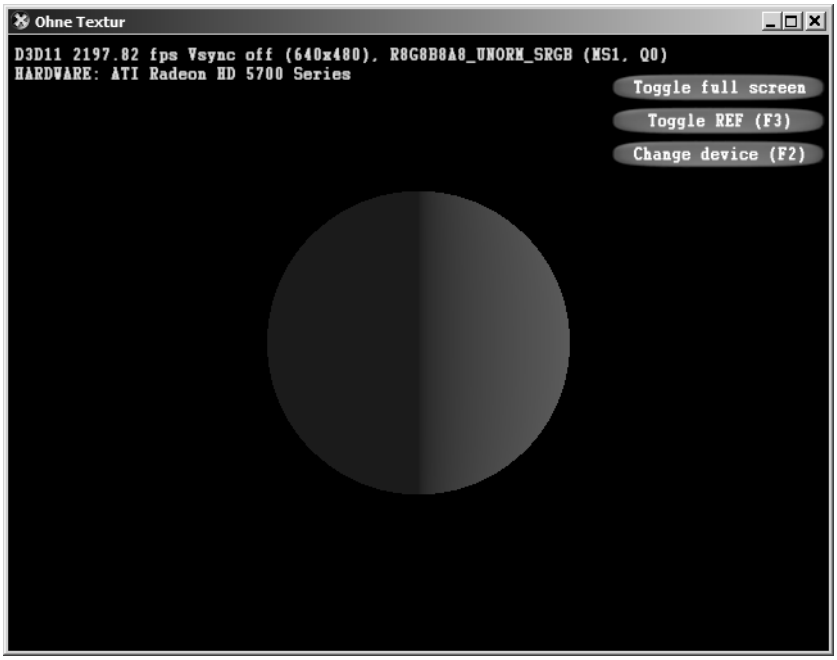

Abb. 1.4: Modell mit Beleuchtung ohne Textur

1.4.7 Texturen

Eine *Textur* ist, wie schon erwähnt, ein Bild, das wie eine Tapete auf das Modell »geklebt« wird. In der Regel erfolgt die Zuweisung einer oder mehrerer Texturen bereits im Modellierungsprogramm, so dass der C++-Code (zusammen mit dem Shader) lediglich dafür sorgen muss, dass die Textur beim Rendern auch angewandt wird. Wie Sie das auch ohne die Klasse `CDXUTSDKMesh` tun können, erfahren Sie weiter unten im vorliegenden Abschnitt.

Texturkoordinaten

Wie kommt es eigentlich, dass die in den Beispielprogrammen verwendete Textur auf das Modell »passt«? Das Modell selbst muss dazu natürlich die Information enthalten, welcher Teil der Textur wohin gehört. Normalerweise speichert dazu schon das Modellierungsprogramm zu jedem Vertex zwei `float`-Werte zwischen `0` und `1`, die so genannten *Texturkoordinaten* (meist als `u` und `v` bezeichnet). Dabei beschreiben die Werte (0, 0) die linke obere und (1, 1) die rechte untere Ecke der Textur. Der Shadercode holt dann anhand dieser Texturkoordinaten den Farbwert aus der Textur (wobei die Werte der Texturkoordinaten zwischen den Vertices interpoliert werden).

Das Erstellen dieser Texturkoordinaten im Modellierungsprogramm kann so erfolgen, dass eine einfache Geometrie (meist ein Würfel, eine Kugel, ein Zylinder oder einfach eine ebene Fläche) um das Objekt »herumgelegt« und die Texturkoordinaten von dieser Geometrie auf das Objekt projiziert werden (*UV-Mapping* – oder *UVW-Mapping*, wenn das Objekt mit dreidimensionalen Texturkoordinaten versehen wird). Dies ist bei Objekten mit homogener Struktur (Holz, Metall etc.) oft ausreichend.

Soll ein Objekt (z.B. ein Charactermodell) aber detailliert »bemalt« werden, so muss quasi die Außenhaut des Objekts in eine ebene Fläche »abwickeln« (*Texture Unwrapping*), wobei die einzelnen Dreiecke, aus denen das Objekt besteht, in dieser Fläche eingezeichnet werden. Danach kann man in einem Grafikprogramm die einzelnen Flächen des Objekts bequem anmalen. Das Texture Unwrapping im Modellierungsprogramm ist bei komplexen Geometrien meist nicht ganz einfach und stellt auch an das Programm höhere Ansprüche als einfaches UV-Mapping.

Sampler State

Einzelheiten dazu, wie die Grafikkarte das Nachschlagen eines Farbwertes in einer Textur ausführt, regeln Sie mit Hilfe eines *Sampler State*. Dieser legt beispielsweise fest, was mit Texturkoordinatenwerten passiert, die größer als 1 sind, ob zum Bestimmen eines Farbwertes auch die benachbarten Punkte in der Textur heranzogen werden (*Textur-Filterung*) oder ob und wie mehrere Versionen einer Textur mit unterschiedlichem Detaillierungsgrad (*Level of Detail, LOD*) verwendet werden.

Das SIMPLESAMPLE hat schon eine passende Variable:

```
ID3D11SamplerState* g_pSamLinear = NULL;
```

Auch der Code für die Initialisierung in `OnD3D11CreateDevice` ist im SIMPLESAMPLE bereits vorhanden:

```
D3D11_SAMPLER_DESC samDesc;
ZeroMemory(&samDesc, sizeof(samDesc));
samDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
samDesc.AddressU = samDesc.AddressV = samDesc.AddressW =
   D3D11_TEXTURE_ADDRESS_WRAP;
samDesc.MaxAnisotropy = 1;
samDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
samDesc.MaxLOD = D3D11_FLOAT32_MAX;
pd3dDevice->CreateSamplerState(&samDesc, &g_pSamLinear);
```

Listing 1.14: In `OnD3D11CreateDevice` einzufügender Code

In `OnD3D11FrameRender` wird der Sampler State gesetzt:

```
pd3dImmediateContext->PSSetSamplers(0, 1, &g_pSamLinear);
```

Auch `OnD3D11DestroyDevice` muss ergänzt werden, um den Sampler State wieder freizugeben:

```
SAFE_RELEASE(g_pSamLinear);
```

Laden einer Textur aus einer Bilddatei

In den bisherigen Beispielen erledigte die Klasse `CDXUTSDKMesh` das Laden der Textur und die Übergabe an den Shader. Sie können aber auch selbst eine Textur aus einer Datei laden und diese statt der von `CDXUTSDKMesh` geladenen an den Shader übergeben.

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_04`.

Abb. 1.5: Erdkugel mit externer Textur (Beleuchtung hier weiß)

Variable

Deklarieren Sie dazu eine Variable vom Typ `ID3D11ShaderResourceView`:

```
ID3D11ShaderResourceView* g_pRV = NULL;
```

OnD3D11CreateDevice

Das Laden der Textur erfolgt mit der DXUT-Hilfsfunktion `CreateTextureFromFile` in `OnD3D11CreateDevice`:

```
DXUTFindDXSDKMediaFileCch(str, MAX_PATH, L"Genetica\\Absolute Rust.jpg");
DXUTGetGlobalResourceCache().CreateTextureFromFile(pd3dDevice,
    str, &g_pRV);
```

`DXUTFindDXSDKMediaFileCch` findet wieder die Ressource (in diesem Fall die Bilddatei) im Verzeichnis `media` (hier in einem Unterverzeichnis davon).

OnD3D11DestroyDevice

In `OnD3D11DestroyDevice` muss die ResourceView-Variable wieder freigegeben werden:

```
SAFE_RELEASE(g_pRV);
```

OnD3D11FrameRender

Beim Rendern setzen wir die Shader-Ressource auf unser `ResourceView`-Objekt. An die `Render`-Methode des Mesh-Objekts übergeben wir diesmal nicht `0`, sondern `-1` – das Mesh-Objekt soll die Shader-Variable ja nun nicht mehr setzen:

```
ID3D11ShaderResourceView* aRViews[1] = {g_pRV };
pd3dImmediateContext->PSSetShaderResources(0, 1, aRViews);

g_Mesh.Render(pd3dImmediateContext);
```

Da `-1` die Voreinstellung ist, können wir den Parameter auch ganz weglassen.

1.5 RenderStates

Der Zustand der Grafikkarte beim Rendern wird repräsentiert durch verschiedene so genannte *State*-Objekte, die nach den unterschiedlichen Stufen der Rendering-Pipeline gruppiert werden können.

Für uns interessant sind hier insbesondere die Rasterizer-Stufe (*Rasterizer Stage, RS*) und die Output-Merger-Stufe (*Output Merger Stage*, OM).

Zum Rasterizer Stage gehört als State-Objekt der *Rasterizer State*, zum Output Merger Stage die State-Objekte *Depth-Stencil State* und *Blend State*.

1.5.1 Rasterizer State

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_05`.

Der Rasterizer State bestimmt unter anderem den Füll-Modus (*Fill Mode*), der festlegt, ob die Dreiecke ausgefüllt oder nur als Drahtgittermodell (*Wireframe*) gezeichnet werden, sowie den Cull-Modus (*Cull Mode*), der bestimmt, ob beide Seiten eines Dreiecks oder nur eine gezeichnet werden, und im zweiten Fall, welche Seite.

Drahtgitter-Modus

Um unser Modell im Wireframe-Modus anzeigen zu können, benötigen wir eine Variable für den Rasterizer State (oben in der Quelltextdatei unseres Programms):

```
ID3D11RasterizerState* g_pRasterState = NULL;
```

Für das Initialisieren dieser Variablen mit der Methode `CreateRasterizerState` des Device-Objekts müssen wir eine Struktur des Typs `D3D11_RASTERIZER_DESC` füllen. Wir erledigen dies mit einer eigenen Funktion, die ebenfalls `CreateRasterizerState` heißt:

```
void CreateRasterizerState(ID3D11Device* pd3dDevice)
{
    D3D11_RASTERIZER_DESC desc;
    desc.FillMode = D3D11_FILL_WIREFRAME;
    desc.CullMode = D3D11_CULL_BACK;
    desc.FrontCounterClockwise = true;
    desc.DepthBias = false;
    desc.DepthBiasClamp = 0;
    desc.SlopeScaledDepthBias = 0;
    desc.DepthClipEnable = true;
    desc.ScissorEnable = false;
    desc.MultisampleEnable = false;
    desc.AntialiasedLineEnable = false;

    pd3dDevice->CreateRasterizerState(&desc, &g_pRasterState );
}
```

Listing 1.15: `CreateRasterizerState`

Backface Culling

Neben dem Element `FillMode`, das hier den Wert `D3D11_FILL_WIREFRAME` erhält, sind insbesondere auch noch die Elemente `CullMode` und `FrontCounterClockwise` von Interesse: Der Wert `D3D11_CULL_BACK` für den `CullMode` sorgt dafür, dass die Rückseiten der Dreiecke nicht gezeichnet werden (*Backface Culling*). Welche Seiten dabei als Vorder- bzw. Rückseite angesehen werden, wird durch den Umlaufsinn der Eckpunkte bestimmt: Hat des Element `FrontCounterClockwise` den Wert `true`, so ist diejenige Seite die Vorderseite, auf der die Vertices fortlaufend im Uhrzeigersinn angeordnet sind.

OnD3D11CreateDevice

Das Anlegen des `RasterizerState`-Objekts (hier mit unserer Funktion `CreateRasterizerState`) erfolgt in `OnD3D11CreateDevice`:

```
CreateRasterizerState(pd3dDevice);
```

OnD3D11FrameRender

Vor dem Rendern müssen wir nun nur noch mit der `RSSetState`-Funktion des Device-Objekts den Zustand umschalten:

```
pd3dDevice->RSSetState(g_pRasterState);
```

Falls nicht alle, sondern nur bestimmte Objekte mit den besonderen Einstellungen (hier in der Drahtgitter-Ansicht) gerendert werden sollen, müssen Sie gegebenenfalls vorher den alten Zustand mit `RSGetState` speichern und nach dem Rendern wiederherstellen (siehe Abschnitt 3.10.4).

OnD3D11DestroyDevice

In `OnD3D11DestroyDevice` wird das Rasterizer-State-Objekt wieder freigegeben:

```
SAFE_RELEASE(g_pRasterState);
```

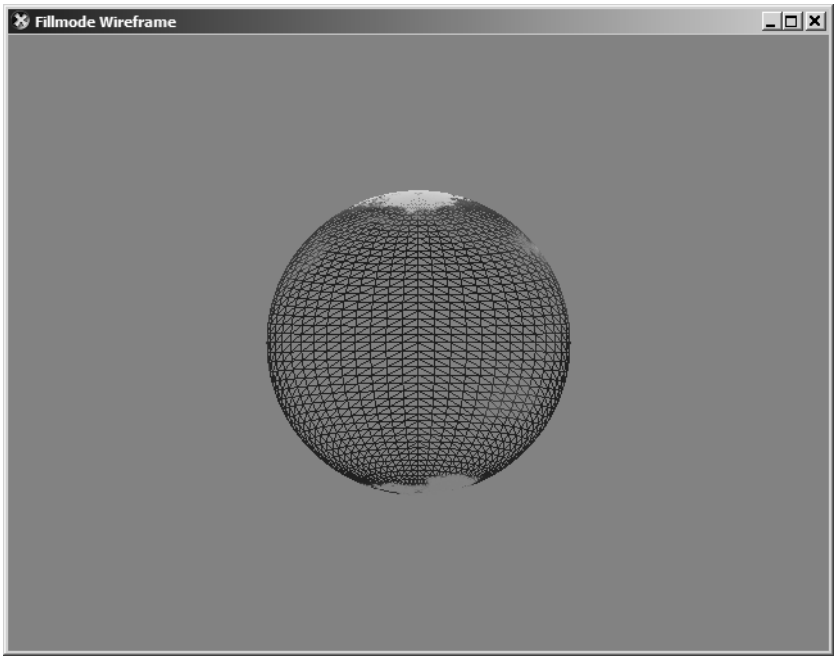

Abb. 1.6: Fillmode Wireframe

1.5.2 Depth-Stencil State

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_06`.

Der Depth-Stencil State legt die Einzelheiten beim *Z-Buffering* fest. Z-Buffering löst das Problem, dass beim Zeichnen dreidimensionaler Objekte natürlich diejenigen Objekte, die weiter hinten liegen, von den davorliegenden verdeckt werden müssen. Tatsächlich muss auch ein Objekt nicht komplett vor einem anderen liegen, sondern kann auch teilweise von diesem verdeckt werden, in anderen Teilen aber selbst das andere Objekt verdecken. Wir müssen also letztlich für jeden Punkt auf dem Bildschirm einzeln untersuchen, welcher der möglicherweise von verschiedenen Objekten an dieser Stelle gezeichneten Punkte am weitesten vorn liegt, und nur diesen tatsächlich sichtbar machen. Genau dazu dient das Z-Buffering (oder auch *Depth-Buffering*): Beim Zeichnen wird für jeden Bildpunkt (dessen Lage am Bildschirm ja bereits durch seine x- und y-Koordinaten vollständig bestimmt ist) die z-Koordinate (durch die Projektionsmatrix

umgerechnet auf den Bereich von 0 bis 1) in den Depth-Buffer geschrieben. Ist dort für dieselbe x- und y-Koordinate bereits ein Wert vorhanden (und folglich auf dem Bildschirm an dieser Position ebenfalls bereits ein Punkt gezeichnet), so überschreibt der neue Punkt den vorhandenen nur dann, wenn der neue Punkt einen kleineren z-Wert hat als der vorhandene, also weiter vorn liegt.

Allerdings ist Z-Buffering nicht in allen Situationen erwünscht: So müssen beispielsweise bei teilweise durchsichtigen Objekten ja auch die dahinterliegenden noch sichtbar sein – oder beim Zeichnen einer Skybox, die immer vor allen anderen Objekten gerendert wird, kann man davon ausgehen, dass sich nichts im Z-Buffer befindet und kann sich (bzw. der Grafikkarte) daher auch die Mühe des Vergleichs ersparen. Deshalb kann die Verwendung des Z-Buffers von der Anwendung recht detailliert gesteuert werden.

Im Einzelnen können wir mit dem Depth-Stencil State festlegen, ob die z-Koordinate der Punkte überhaupt beim Rendern in den Depth-Buffer geschrieben wird (*DepthWriteMask*), ob vorhandene Werte im Depth-Buffer beim Rendern berücksichtigt werden (*DepthEnable*) und wie der z-Wert eines neu zu zeichnenden Punktes mit dem im Depth-Buffer vorhandenen verglichen werden soll (*DepthFunc*).

Auch für den Depth-Stencil State fügen wir eine globale Variable oben in unsere Quelltextdatei ein:

```
ID3D11DepthStencilState* g_pDepthStencilState = NULL;
```

Wie beim Rasterizer State ist eine Struktur zu füllen, die dann an die Methode `CreateDepthStencilState` des Device-Objekts übergeben wird. Wir schreiben dafür wieder eine eigene Funktion, die ebenfalls den Namen `CreateDepthStencilState` bekommt:

```
void CreateDepthStencilState(ID3D11Device* pd3dDevice)
{
   D3D11_DEPTH_STENCIL_DESC depthStencilDesc;
   depthStencilDesc.DepthEnable = true;
   depthStencilDesc.DepthFunc = D3D11_COMPARISON_LESS;
   depthStencilDesc.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ZERO;
   depthStencilDesc.StencilEnable = false;

   pd3dDevice->CreateDepthStencilState(&depthStencilDesc,
      &g_pDepthStencilState);
}
```

Listing 1.16: `CreateDepthStencilState`

OnD3D11CreateDevice

Der Aufruf unserer Funktion für das Initialisieren des Depth-Stencil State erfolgt wie beim Rasterizer State in `OnD3D11CreateDevice`:

```
CreateDepthStencilState(pd3dDevice);
```

OnD3D11FrameRender

Vor dem Rendern wird der Zustand mit `OMSetDepthStencilState` umgeschaltet:

```
pd3dDevice->OMSetDepthStencilState(g_pDepthStencilState, 0);
```

Auch den Depth-Stencil State kann man bei Bedarf vorher speichern und nach dem Rendern wieder herstellen (siehe Abschnitt 3.10.3).

OnD3D11DestroyDevice

In `OnD3D11DestroyDevice` wird das Depth-Stencil-State-Objekt wieder freigegeben:

```
SAFE_RELEASE(g_pDepthStencilState);
```

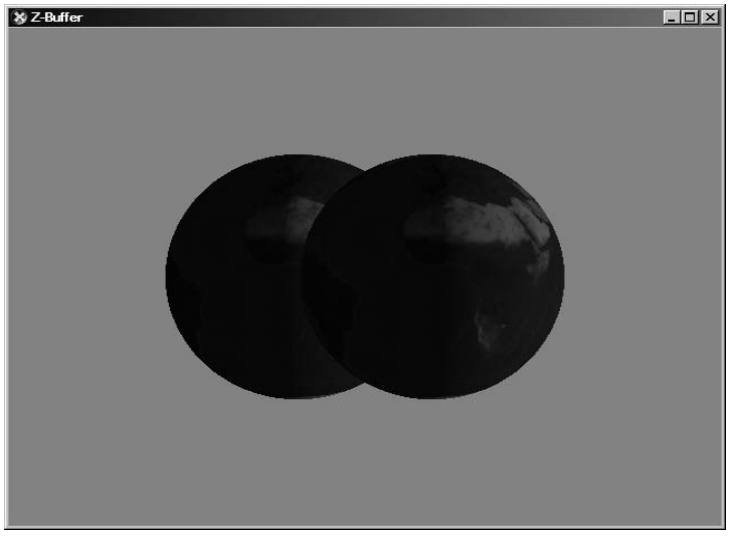

Abb. 1.7: Rendern ohne Z-Buffer – die zweite Kugel verdeckt die erste komplett.

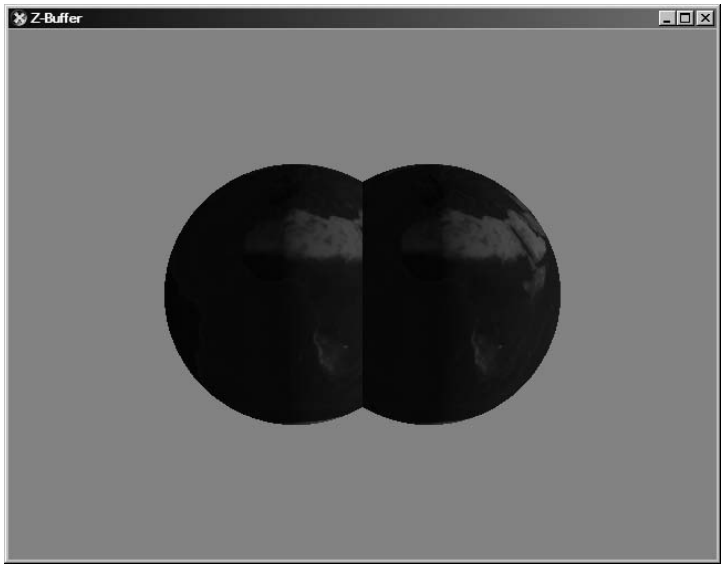

Abb. 1.8: Rendern mit Z-Buffer

1.5.3 Blend State

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_07`.

Mit dem Alpha Blend State kontrollieren Sie die Einzelheiten beim Rendern transparenter oder teilweise transparenter Objekte.

Alpha-Blending

Das *Alpha-Blending* hat seinen Namen von dem *Alpha-Kanal*, der sowohl in Zeichenprogrammen als auch in der 3D-Grafik als zusätzliche Information zu der Farbe eines Bildpunkts (oder eines Vertex) angegeben werden kann und der die Durchsichtigkeit (Transparenz) dieses Punktes bestimmt. Der Alpha-Kanal für eine Textur (bestehend aus den Alpha-Werten (α) aller Punkte dieser Textur) kann je nach Dateiformat in einer separaten Datei oder auch mit in derselben Bilddatei gespeichert sein. Häufig gibt man auch Farbwerte für jeden Punkt direkt mit vier Komponenten (statt drei für die Rot-, Grün- und Blau-Anteile (*RBG*)) an, wobei die vierte Komponente den Alpha-Wert angibt (*RGBA*). Auch die Strukturen, mit denen Farbwerte in DirectX dargestellt werden, enthalten in der Regel eine vierte Komponente für den Alpha-Wert. Ist dieser `1`, ist der Punkt komplett undurchsichtig (opak), ist er `0`, so ist der Punkt komplett transparent.

Der Alpha Blend State legt fest, wie beim Zeichnen eines transparenten Objekts im Vordergrund die Farbe eines bereits vorher gezeichneten Objekts im Hintergrund (*Destination*) mit der Farbe des darüber zu zeichnenden Objekts (*Source*) kombiniert werden. (Dieses Mischen von Vorder- und Hintergrundfarbe wird als *Blending* bezeichnet.) Im Normalfall verwendet man dafür gerade den α-Wert aus der Textur (oder der Vertexfarbe) des Vordergrundobjekts: Dieser wird mit der Vordergrundfarbe, der Wert (1-α) mit der Hintergrundfarbe multipliziert, und das Ergebnis bestimmt die Farbe des Punkts am Bildschirm.

Dazu erhält das Element `SrcBlend` der Struktur für die Initialisierung des Alpha Blend State (vom Typ `D3D11_BLEND_DESC`) den Wert `D3D11_BLEND_SRC_ALPHA` und das Element `DestBlend` den Wert `D3D11_BLEND_INV_SRC_ALPHA`.

Additives Blending

Gelegentlich möchte man aber für die Kombination von Vorder- und Hintergrund nicht den Alpha-Kanal verwenden (z.B. weil man eine Textur benutzen möchte, die nicht über einen Alpha-Kanal verfügt). Wenn man die Farbwerte von Vordergrund und Hintergrund einfach addiert, erscheinen diejenigen Teile des Vordergrundbildes mit schwarzer Farbe (alle drei Farbwerte 0) durchsichtig – allerdings werden Bildteile, die sowohl im Vorder- als auch im Hintergrundbild farbig sind, aufgehellt, weil auch diese Farbwerte addiert werden. In einfachen Fällen kann man aber auch damit sehr schöne Transparenzeffekte erzielen. Hierzu setzen Sie die Elemente `SrcBlend` und `DestBlend` der `D3D11_BLEND_DESC`-Struktur beide auf `D3D11_BLEND_ONE`, das heißt, die Anteile von Vorder- und Hintergrund gehen beide mit einem Faktor `1` in das Gesamtergebnis ein.

In den Z-Buffer darf dabei nicht geschrieben werden – ansonsten würden auch schwarze Bildteile des Vordergrunds, die ja eigentlich transparent sein sollen, weiter hinten liegende Objekte überdecken. Das gilt im Übrigen natürlich auch bei Verwendung des Alpha-Kanals für das Blending.

Transparente Objekte müssen aber ohnehin stets zuletzt gezeichnet werden: Nur wenn die dahinter durchscheinenden Bildteile bereits gezeichnet sind, kann ja die Kombination von Vorder- und Hintergrund richtig funktionieren.

Neben dem echten Alpha-Blending und dem additiven Blending sind mittels der verschiedenen Konstanten für `SrcBlend` und `DestBlend` auch andere, exotischere Kombinationen möglich – die hier genannten stellen aber die am häufigsten auftretenden Fälle dar.

Um in unserem Beispielprogramm transparente Objekte rendern zu können, fügen wir eine Variable für den Blend State oben in die Quelltextdatei ein:

```
ID3D11BlendState* g_pAlphaBlendState;
```

Das Auffüllen der dazugehörenden Struktur und Initialisieren der Variablen mit `CreateBlendState` erledigen wir mit einer eigenen Funktion, die den Namen `CreateAlphaBlendState` bekommt:

```
void CreateAlphaBlendState(ID3D11Device* pd3dDevice, bool bAdditive)
{
   D3D11_BLEND_DESC desc;
   ZeroMemory(&desc, sizeof(D3D11_BLEND_DESC));
   desc.BlendEnable[0] = true;
   desc.RenderTargetWriteMask[0] = D3D11_COLOR_WRITE_ENABLE_ALL;

   if(bAdditive)
   {
      desc.SrcBlend = D3D11_BLEND_ONE;
      desc.DestBlend = D3D11_BLEND_ONE ;
   }
   else
   {
      desc.SrcBlend = D3D11_BLEND_SRC_ALPHA;
      desc.DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
   }
   desc.BlendOp = D3D11_BLEND_OP_ADD;

   //Alpha der Destination bleibt erhalten
   desc.SrcBlendAlpha = D3D11_BLEND_ZERO;
   desc.DestBlendAlpha = D3D11_BLEND_ONE;
   desc.BlendOpAlpha = D3D11_BLEND_OP_ADD;

   pd3dDevice->CreateBlendState(
      &desc, &g_pAlphaBlendState);
}
```

Listing 1.17: `CreateAlphaBlendState`

Mit dem `bool`-Prameter `bAdditive` können wir entscheiden, ob echtes Alpha-Blending oder additives Blending verwendet werden soll.

Der Alpha-Wert des Hintergrunds wird übrigens hier nicht mit demjenigen des Vordergrunds kombiniert, sondern bleibt immer erhalten. Beim Rendern mehrerer transparenter Objekte übereinander können durchaus komplexe Situationen auftreten, bei denen man das Zusammenwirken von Alpha-Blending und eventuell Z-Buffering sorgfältig planen muss. In der Regel wird es dabei hilfreich sein, alle durchsichtigen Objekte nach allen undurchsichtigen zu zeichnen und die transparenten Objekte untereinander nach dem z-Wert vorzusortieren.

OnD3D11CreateDevice

In `OnD3D11CreateDevice` rufen wir unsere Funktion auf:

```
CreateAlphaBlendState(pd3dDevice, true);
```

OnD3D11FrameRender

Vor dem Rendern wird der Blend State mit `OMSetBlendState` umgeschaltet:

```
pd3dDevice->OMSetBlendState(g_pAlphaBlendState, NULL, 0xffffffff);
```

Auch der Blend State kann (wie der Rasterize State und der Depth-Stencil State) vorher gespeichert und nach dem Rendern wieder hergestellt werden (vgl. Abschnitt 3.10.2)

OnD3D11DestroyDevice

In `OnD3D11DestroyDevice` wird das Blend-State-Objekt wieder freigegeben:

```
SAFE_RELEASE(g_pAlphaBlendState);
```

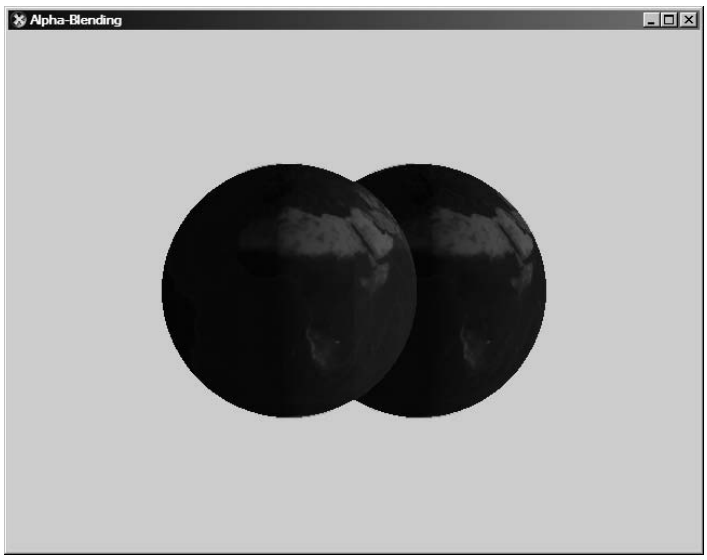

Abb. 1.9: Alpha-Blending

1.6 Verschiedene Kameratypen

Für verschiedene Zwecke sind verschiedene Arten von Kameras geeignet; die verschiedenen DXUT-Kameras unterscheiden sich in der Steuerung mit Maus und Tastatur.

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_08`.

CBaseCamera

`CBaseCamera` ist die gemeinsame Basisklasse der DXUT-Kameras. Sie stellt die grundlegenden Funktionen zur Verfügung, die allen Kameratypen gemeinsam sind.

CModelViewerCamera

Die Modelviewer-Kamera ist, wie der Name schon andeutet, weniger für Spiele gedacht als zum Betrachten von Modellen. Die Entfernung der Kamera zum Objekt bleibt dabei konstant. Mit der Maus kann das Objekt gedreht und so von allen Seiten betrachtet werden.

In Spielen eignet sich eine solche Kamera zum Beispiel, um Gegenstände in einem Inventar genauer zu untersuchen.

Beim SIMPLESAMPLE-Beispielprojekt ist die voreingestellte Kamera eine Modelviewer-Kamera. Normalerweise werden Sie das ändern, da Sie bei einem Spiel eher durch die Szene hindurch gehen (oder fliegen) wollen.

CFirstPersonCamera

Die First-Person-Kamera ist der typische von First-Person-Shootern (in Deutschland auch Ego-Shooter genannt) bekannte Kameratyp: Die Szene wird aus der Sicht des Spielers gezeigt, wobei dieser von seiner eigenen Figur meist nur die nach vorn ausgestreckte Hand (in der Regel mit einer Waffe) zu sehen bekommt.

In unseren Beispielen verzichten wir auf die Hand mit der Waffe – Sie steuern also einfach mit den Pfeiltasten die Kamera durch die Szene.

Im Gegensatz dazu wird eine Kamera, bei der der Spieler eine Figur durch die Szene steuert und die Kamera dieser folgt, als Third-Person-Kamera bezeichnet.

Verwendung im Beispielprogramm

Für die meisten unserer Beispiele werden wir die First-Person-Kamera verwenden. Ändern Sie dazu die Zeile im SIMPLESAMPLE, in der die Kamera deklariert wird, in

```
CFirstPersonCamera      g_Camera;
```

Entfernen Sie außerdem aus `OnD3D11ResizedSwapChain` die Zeilen

```
g_Camera.SetWindow(pBackBufferSurfaceDesc->Width,
   pBackBufferSurfaceDesc->Height );
g_Camera.SetButtonMasks( MOUSE_LEFT_BUTTON, MOUSE_WHEEL, MOUSE_MIDDLE_BUTTON );
```

Die First-Person-Kamera hat die in diesen Zeilen verwendeten Methoden nicht.

Während bei der Modelviewer-Kamera nicht nur die View- und Projection-Matrizen, sondern auch die World-Matrix von der Kamera bestimmt werden, ist bei einer First- oder Third-Person-Kamera die World-Matrix nur vom Objekt abhängig. Solange sich das Objekt in der Szene nicht bewegt oder seine Größe ändert, ist die World-Matrix sogar konstant (siehe Abschnitt 2.1.2).

Wir fügen deshalb eine globale Variable für diese World-Matrix oben in unsere Programmdatei ein:

```
D3DXMATRIX  g_mWorld;
```

Hinweis

In den Beispielen dieses Kapitels verwenden wir jeweils nur ein Szenenobjekt, also auch nur eine World-Matrix. Wenn Sie mehrere Objekte in die Szene einfügen, wird natürlich auch jedes davon seine eigene World-Matrix haben, die seine Position und Orientierung beschreibt. In Kapitel 3 werden wir eine Klasse für Grafikobjekte entwickeln, die diese World-Matrix und andere Informationen verwaltet.

Die World-Matrix hat zwar nicht direkt mit der Kamera zu tun, geht aber zusammen mit den Kameraparametern in die Berechnung der World-View-Projection-Matrix mit ein.

Die Funktion `OnD3D11FrameRender` wird entsprechend abgeändert. Die World-Matrix ist einfach die Einheitsmatrix, die wir mit der Funktion `D3DXMatrixIdentity` in die Variable hineinschreiben:

```
void CALLBACK OnD3D11FrameRender(ID3D11Device* pd3dDevice, ID3D11DeviceContext*
    pd3dImmediateContext, double fTime, float fElapsedTime, void* pUserContext)
{
    // Clear render target and the depth stencil
    //...

    D3DXMATRIX  mWorldViewProjection;
    D3DXMATRIX  mView;
    D3DXMATRIX  mProj;

    D3DXMatrixIdentity(&g_mWorld);
    // Get the projection & view matrix from the camera class
    mProj = *g_Camera.GetProjMatrix();
    mView = *g_Camera.GetViewMatrix();
    mWorldViewProjection = g_mWorld * mView * mProj;

    // Set the constant buffers
    D3D11_MAPPED_SUBRESOURCE MappedResource;
    pd3dImmediateContext->Map(g_pcbVSPerObject11, 0, D3D11_MAP_WRITE_DISCARD, 0,
        &MappedResource);
    CB_VS_PER_OBJECT* pVSPerObject = (CB_VS_PER_OBJECT*)MappedResource.pData;
    D3DXMatrixTranspose(&pVSPerObject->m_mWorldViewProjection,
    &mWorldViewProjection);
    D3DXMatrixTranspose(&pVSPerObject->m_mWorld, &g_mWorld);
```

```
    //...
    pd3dImmediateContext->Unmap(g_pcbVSPerObject11, 0);
    pd3dImmediateContext->VSSetConstantBuffers(0, 1, &g_pcbVSPerObject11);

    //Objekte rendern...
}
```

Listing 1.18: Änderungen in `OnD3D11FrameRender` für eine Szene mit einer First-Person-Kamera

Hinweis

Die *Einheitsmatrix* oder auch *Identitätsmatrix* als World-Matrix beschreibt ein Objekt, das im Raum weder verschoben noch gedreht noch skaliert ist. Es wird also einfach so, wie es in der Modelldatei gespeichert ist, im Koordinatenursprung gezeichnet (siehe Abschnitt 2.1).

COrthoCamera

Die *orthogonale Kamera* projiziert das Bild nicht perspektivisch, sondern rechtwinklig – man erhält also das Bild eines Punktes, indem man von diesem eine Senkrechte auf die Bildebene zieht. Längen und Winkel bleiben dabei erhalten.

Damit eignet sich die orthogonale Kamera hervorragend zur Herstellung einer Karte für eine Szene oder ein Level, die man in einem Spiel als Orientierungshilfe für den Spieler einblendet.

Eine eigene Kameraklasse

Hinweis

Das Beispielprogramm `Kap05_05` auf der Buch-CD zeigt die Verwendung der Ortho-Kamera zum Rendern einer Mini-Karte, die in einer Bildschirmecke angezeigt wird. Die hier vorgestellte Kameraklasse finden Sie im Projekt MyEngine.

Die Technik des Renderns in eine Textur wird in Abschnitt 9.1 erklärt.

Für das Verständnis des Folgenden ist die orthogonale Kamera zunächst nicht wichtig, sie wird lediglich der Vollständigkeit halber hier schon mit vorgestellt.

Sie können diesen Abschnitt beim ersten Lesen zunächst auch überspringen und später darauf zurückkommen.

Mit Hilfe der DirectX-Funktion `D3DXMatrixOrthoLH` kann man eine Projektionsmatrix für eine orthogonale Kamera erzeugen. Wir leiten eine eigene Kameraklasse von der DXUT-Klasse `CBaseCamera` ab:

```
class COrthoCamera : public CBaseCamera
{
public:
    COrthoCamera(void);
    ~COrthoCamera(void);
```

```
    virtual void SetProjParams(FLOAT w, FLOAT h,
       FLOAT fNearPlane, FLOAT fFarPlane);
    virtual void SetViewParams(
       D3DXVECTOR3* pvEyePt, D3DXVECTOR3* pvLookatPt);

    virtual void FrameMove(FLOAT fElapsedTime){}
};
```

Listing 1.19: Die Klasse COrthoCamera

Konstruktor und Destruktor sind leer, der Konstruktor ruft implizit den Standardkonstruktor der Basisklasse auf:

```
COrthoCamera::COrthoCamera(void){ }
COrthoCamera::~COrthoCamera(void){ }
```

Listing 1.20: Konstruktor und Destruktor

In der überschriebenen Methode `SetProjParams` wird die Projektionsmatrix erstellt:

```
void COrthoCamera::SetProjParams(FLOAT w, FLOAT h,
   FLOAT fNearPlane, FLOAT fFarPlane)
{
   m_fFOV       = 0;
   m_fAspect    = w / h;
   m_fNearPlane  = fNearPlane;
   m_fFarPlane   = fFarPlane;
   D3DXMatrixOrthoLH(&m_mProj, w, h, fNearPlane, fFarPlane);
}
```

Listing 1.21: SetProjParams

Die ersten beiden Parameter von `SetProjParams` sind hier die Höhe und Breite des abzubildenden Szenenausschnitts. Die anderen beiden Parameter wie in der Basisklasse legen die Entfernung der *Near Plane* und der *Far Plane* fest (die kleinste bzw. größte noch mit der Kamera abbildbare Entfernung, siehe Abschnitt 2.2.1). Einen Öffnungswinkel gibt es bei der orthogonalen Kamera nicht.

Die Methode `SetViewParams` könnte eigentlich unverändert aus der Basisklasse übernommen werden – leider hat die Implementierung in der Klasse `CBaseCamera` aber eine Besonderheit, die sie gerade für eine Ansicht der Szene senkrecht von oben unbrauchbar macht: Der so genannt Up-Vektor (siehe Abschnitt 2.2.1), der zusammen mit der Blickrichtung die Orientierung der Kamera bestimmt, zeigt bei `CBaseCamera` immer in Richtung der y-Achse. Das ist für einen aufrecht stehenden Beobachter, der die Kamera geradehält, auch sinnvoll – der Up-Vektor heißt ja auch deshalb so, weil er normalerweise nach oben zeigt. Wenn nun aber die Blickrichtung ebenfalls in Richtung der y-Achse zeigt (die beiden Vektoren also parallel bzw. antiparallel sind), dann ist die Orientierung der Kamera nicht mehr eindeutig festgelegt. In diesem Fall muss man den Up-Vektor anders wählen, etwa in Richtung der x- oder z-Achse. Wir berücksichtigen das in unserer neuen Version von `SetViewParams`:

```
VOID COrthoCamera::SetViewParams(
   D3DXVECTOR3* pvEyePt, D3DXVECTOR3* pvLookatPt)
{
```

```
    if(NULL == pvEyePt || NULL == pvLookatPt)
        return;

    m_vDefaultEye = m_vEye = *pvEyePt;
    m_vDefaultLookAt = m_vLookAt = *pvLookatPt;

    D3DXVECTOR3 vUp(0,1,0);

    //Falls gewählte Blickrichtung parallel zu vUp ist, vUp neu wählen
    D3DXVECTOR3 viewDir;
    D3DXVec3Normalize(&viewDir,
        D3DXVec3Subtract(&viewDir, pvLookatPt, pvEyePt));
    if(1 - abs(D3DXVec3Dot(&vUp, &viewDir)) <= 0.0001f)
        vUp = D3DXVECTOR3(1, 0, 0);

    D3DXMatrixLookAtLH(&m_mView, pvEyePt, pvLookatPt, &vUp);

    D3DXMATRIX mInvView;
    D3DXMatrixInverse(&mInvView, NULL, &m_mView);

    D3DXVECTOR3* pZBasis = (D3DXVECTOR3*)&mInvView._31;

    m_fCameraYawAngle = atan2f(pZBasis->x, pZBasis->z);
    float fLen = sqrtf(pZBasis->z * pZBasis->z + pZBasis->x * pZBasis->x);
    m_fCameraPitchAngle = -atan2f(pZBasis->y, fLen);
}
```

Listing 1.22: `SetViewParams`

Man könnte auch den Up-Vektor als zusätzlichen Parameter an die Methode übergeben.

Verwendung im Beispielprogramm

Die Deklaration der Ortho-Kamera erfolgt wie bei den anderen Typen oben in der Quelltextdatei des Hauptprogramms:

```
COrthoCamera              g_MinimapCamera;
```

In `OnD3D11ResizedSwapChain` werden die Projektionsparameter eingestellt:

```
float fAspectRatio =
    pBackBufferSurfaceDesc->Width / (FLOAT)pBackBufferSurfaceDesc->Height;
g_MinimapCamera.SetProjParams(60, 60 / fAspectRatio, 1.0f, 100.0f);
```

Für die Herstellung einer Karte betrachten wir die Szene senkrecht von oben:

```
D3DXVECTOR3 vecEyeMinimap(0.0f, 60, 0.0f);
D3DXVECTOR3 vecAtMinimap (0.0f, 0,  0.0f);
g_MinimapCamera.SetViewParams(&vecEyeMinimap, &vecAtMinimap);
```

Die Entfernung der Kamera von der Szene spielt für die Abbildung keine Rolle, sie muss allerdings für alle sichtbaren Szenenobjekte zwischen Near Plane und Far Plane liegen.

Beim Rendern müssen nun die View- und Projektion-Matrizen dieser Kamera verwendet werden.

CCollisionCamera

Eine Kamera mit Kollisionserkennung wird in Abschnitt 5.7.6 vorgestellt.

1.7 Texte anzeigen

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_09`.

Wenn Sie einfache Textmeldungen direkt auf dem Bildschirm anzeigen möchten, können Sie die Klasse `CDXUTTextHelper` verwenden.

Verwendung im SimpleSample

Was für die Textausgabe mit `CDXUTTextHelper` im Einzelnen zu tun ist, können Sie in dem Beispiel SimpleSample aus dem DirectX-SDK sehen. Dort wird eine Variable für das `CDXUTTextHelper`-Objekt oben in der CPP-Datei deklariert:

```
CDXUTTextHelper*          g_pTxtHelper = NULL;
```

Listing 1.23: Globale Variablen

In der Methode `OnD3D11CreateDevice` wird der DXUT-Dialog-Resource-Manager angelegt und an den Konstruktor der Klasse `CDXUTTextHelper` übergeben:

```
ID3D11DeviceContext* pd3dImmediateContext = DXUTGetD3D11DeviceContext();
g_DialogResourceManager.OnD3D11CreateDevice(pd3dDevice, pd3dImmediateContext);
g_pTxtHelper = new CDXUTTextHelper(pd3dDevice, pd3dImmediateContext,
    &g_DialogResourceManager, 15);
```

Listing 1.24: `OnD3D11CreateDevice`

Da das `CDXUTTextHelper`-Objekt den Dialogressourcenmanager verwendet, ist es außerdem notwendig, dass dessen Methode `OnD3D11ResizedSwapChain` in der Rückruffunktion `OnD3D11ResizedSwapChain` unseres Programms aufgerufen wird:

```
g_DialogResourceManager.OnD3D11ResizedSwapChain(pd3dDevice,
    pBackBufferSurfaceDesc);
```

Die Methode `OnD3D11FrameRender` im SimpleSample ruft ihrerseits lediglich `RenderText` auf. Diese Methode (Listing 1.26) erledigt die Einzelheiten der Textausgabe.

```
void CALLBACK OnD3D11FrameRender(ID3D11Device* pd3dDevice, ID3D11DeviceContext*
    pd3dImmediateContext, double fTime, float fElapsedTime, void* pUserContext)
{
    //...
```

```
    RenderText();
}
```

Listing 1.25: Textausgabe im SIMPLESAMPLE

Beachten Sie, dass die Textausgabe nach dem Rendern der eigentlichen Szene erfolgen muss, damit der Text nicht von den Szenenobjekten überdeckt wird.

Das Zeichnen von Text wird durch Aufrufe der Methoden `Begin` und `End` der Klasse `CDXUTTextHelper` eingerahmt. Zum Festlegen der Zeichenposition verwenden wir die Methode `SetInsertionPos`, zum Festlegen der Vordergrundfarbe die Methode `SetForegroundColor`. Das eigentliche Zeichnen erfolgt mit der Methode `DrawTextLine`. Die Texte, die ausgegeben werden sollen, sind natürlich anwendungsabhängig.

```
void RenderText()
{
    g_pTxtHelper->Begin();
    g_pTxtHelper->SetInsertionPos( 5, 5 );
    g_pTxtHelper->SetForegroundColor(D3DXCOLOR( 1.0f, 1.0f, 0.0f, 1.0f ));
    g_pTxtHelper->DrawTextLine(L"Hallo, Welt");
    g_pTxtHelper->End();
}
```

Listing 1.26: `RenderText`

Vergessen Sie auch nicht, in `OnD3D11DestroyDevice` das Objekt wieder freizugeben:

```
SAFE_DELETE(g_pTxtHelper);
```

Listing 1.27: `OnD3D11DestroyDevice`

Das Makro `SAFE_ DELETE` ruft `delete` nur dann auf, wenn der Zeiger nicht `NULL` ist, und setzt diesen danach auf `NULL`:

```
#define SAFE_DELETE(p)      { if (p) { delete (p);     (p)=NULL; } }
```

1.8 Eingabegeräte

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_10`.

Um die Abfrage der verschiedenen Eingabegeräte zu demonstrieren, soll beim Drücken einer Taste auf der Tastatur oder einer Maustaste einfach nur eine entsprechende Textmeldung angezeigt werden.

Die für die Textausgabe verwendete globale Variable ist dieselbe wie im vorhergehenden Beispiel. Zusätzlich deklarieren wir noch eine Zeichenkettenvariable für die Meldung selbst, die abhängig von der jeweils gedrückten Taste und von der Mausposition verändert werden kann:

```
WCHAR g_strMessage[255];
```

In `OnD3D11CreateDevice` wird wie gehabt das Objekt für die Textausgabe initialisiert und in `OnD3D11DestroyDevice` wieder freigegeben.

Für die Textausgabe selbst verwenden wir, ebenfalls wie im vorhergehenden Beispiel, eine Hilfsfunktion `RenderText`, die hier den Inhalt unserer Zeichenkettenvariablen ausgibt:

```
void RenderText()
{
   g_pTxtHelper->Begin();
   g_pTxtHelper->SetInsertionPos(5, 5);
   g_pTxtHelper->SetForegroundColor(D3DXCOLOR(1.0f, 1.0f, 0.0f, 1.0f));
   g_pTxtHelper->DrawTextLine(g_strMessage);
   g_pTxtHelper->End();
}
```

Listing 1.28: `RenderText`

Die Funktion wird wieder in `OnD3D11FrameRender` aufgerufen.

1.8.1 Tastatur

Für die Abfrage der Tastatur sehen die SDK-Beispiele die Rückruffunktion `OnKeyboard` vor, die Sie nur zu ergänzen brauchen.

OnKeyboard

Als Parameter werden vom Framework eine Zahl für die gedrückte Taste (der *Virtual Key Code*), zwei Bool-Variablen, die angeben, ob die Taste gedrückt (oder losgelassen) wurde und ob dabei auch noch die Alt-Taste gedrückt wurde, sowie eine Zeigervariable für benutzerdefinierte Informationen an die Funktion `OnKeyboard` übergeben. Eine Liste der Virutal Key Codes finden Sie in der Headerdatei `WinUser.h`. (Klicken Sie einfach im Quelltext mit der rechten Maustaste auf einen solchen Key Code (z.B. hier `VK_F1`) und wählen Sie aus dem Kontextmenü den Befehl GEHE ZU DEFINITION, dann wird die Datei angezeigt.)

Unser Beispiel prüft in einer einfachen Fallunterscheidung, welche Taste gedrückt wurde, und setzt die Textmeldung entsprechend:

```
void CALLBACK OnKeyboard(UINT nChar, bool bKeyDown, bool bAltDown,
   void* pUserContext)
{
   if(bKeyDown)
   {
      switch(nChar)
      {
         case VK_F4:
         {
            StringCchPrintf(g_strMessage, 255, L"F4 gedrückt");
         }
         break;
         case VK_F5:
         {
```

```
            StringCchPrintf(g_strMessage, 255, L"F5 gedrückt");
         }
         break;
      }
   }
}
```

Listing 1.29: `OnKeyboard`

1.8.2 Maus

Den Programmcode zur Abfrage der Maus können wir mit in das Beispielprogramm zur Tastaturabfrage unterbringen – wir gehen also davon aus, dass die nötigen Objekte für die Textausgabe, wie weiter vorn beschrieben, bereits vorhanden sind.

OnMouse

Auch für die Mausabfrage sieht DXUT eine Rückruffunktion vor. Diese ist beim SIMPLESAMPLE allerdings noch nicht vorbereitet (wohl aber beim EMPTYPROJECT) – Sie müssen den entsprechenden Programmcode also gegebenenfalls selbst einfügen. Am besten fügen Sie den Prototyp oben in der Quelltextdatei bei den anderen Rückruffunktionen hinzu:

```
void CALLBACK OnMouse(bool bLeftButtonDown, bool bRightButtonDown,
   bool bMiddleButtonDown, bool bSideButton1Down, bool bSideButton2Down,
   int nMouseWheelDelta, int xPos, int yPos, void* pUserContext);
```

Die »Verdrahtung« erfolgt in `wWinMain`:

```
DXUTSetCallbackMouse(OnMouse);
```

Die Implementierung der Funktion können Sie nun an beliebiger Stelle in die Quelltextdatei einfügen. Anhand der übergebenen Parameter können wir feststellen, welche Maustaste(n) gedrückt wurde(n) und an welcher Position sich der Mauszeiger befindet:

```
void CALLBACK OnMouse(bool bLeftButtonDown, bool bRightButtonDown,
   bool bMiddleButtonDown, bool bSideButton1Down, bool bSideButton2Down,
   int nMouseWheelDelta, int xPos, int yPos, void* pUserContext)
{
   WCHAR* strText = L"";

   if(bLeftButtonDown)
      strText =  L"Linke Maustatste gedrückt";
   if(bRightButtonDown)
      strText =  L"Rechte Maustatste gedrückt";
   if(bMiddleButtonDown)
      strText =  L"Rechte Maustatste gedrückt";
   StringCchPrintf(g_strMessage, 255, L"%s Mausposition: (%d, %d)",
      strText, xPos, yPos) ;
}
```

Listing 1.30: `OnMouse`

1.8.3 Gamepad

Auch ein Gamepad, wie man es von Konsolen her kennt, kann an den PC angeschlossen und mit DirectX zur Steuerung eines Spiels verwendet werden. Am besten eignet sich dafür Microsofts »hauseigener« Xbox-Controller, der im Versandhandel zusammen mit Treibern für Windows erhältlich ist.

Das mit dem DirectX-SDK ausgelieferte Beispielprogramm XInputSimpleController demonstriert sehr schön, wie man vom Programm aus den Zustand der verschiedenen Buttons etc. abfragen kann.

Auch die DXUT-Kamera ist bereits auf die Verwendung eines Gamepads eingerichtet: Mit dem linken Thumbstick kann die Kamera bewegt und mit dem rechten die Blickrichtung gesteuert werden. Damit funktionieren auch (fast) alle Beispielprogramme dieses Buchs ohne weitere Programmierung schon mit dem Xbox-Controller.

Das Beispielprogramm `Kap04_01` auf der Buch-CD zeigt außerdem die Verwendung des Steuerkreuzes sowie die Verwendung der Vibrationsmotoren für »Rumble«-Effekte. Es wird in Abschnitt 4.3.2 beschrieben.

1.9 Sounds

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_11`.

Um bereits Ihre allerersten Beispielprogramme und Minispiele mit einfachen Sounds zu versehen, können Sie mit der Anweisung `PlaySound` WAV-Dateien abspielen. Um beispielsweise beim Start des Programms ein Intro abzuspielen, fügen Sie die folgende Zeile hinter dem Kommentar

```
// Perform any application-level initialization here
```

in `wWinMain` ein:

```
PlaySound(L"..\\..\\media\\sounds\\intro.wav", 0, SND_ASYNC);
```

Das Flag `SND_ASYNC` bewirkt, dass die Funktion sofort zurückkehrt – das Programm also weiterläuft, während der Sound im Hintergrund abgespielt wird. Ohne dieses Flag würde die Programmausführung anhalten, bis der Sound zu Ende abgespielt ist.

In unserem Beispiel sollen weitere Klänge beim Drücken der Funktionstasten F4 bzw. F5 ertönen. Ergänzen Sie dazu die Funktion `OnKeyboard`:

```
void CALLBACK OnKeyboard( UINT nChar, bool bKeyDown, bool bAltDown, void* pUser-
Context)
{
   if( bKeyDown )
   {
      switch( nChar )
      {
```

```
            case VK_F4:
            {
               PlaySound(L"..\\..\\media\\sounds\\photon_gun_shot.wav ", 0,
                  SND_ASYNC);
            }
            break;
            case VK_F5:
            {
               PlaySound(L"..\\..\\media\\sounds\\mother_ship.wav", 0,
                  SND_ASYNC);
            }
            break;
         }
      }
   }
```

Listing 1.31: Abspielen von Sounds bei Tastendruck

Wie Sie mit Hilfe der XAudio2-Programmierschnittstelle und des XACT-Tools Musik, Hintergrundgeräusche und Soundeffekte für ein komplexeres Spiel präzise steuern können, erfahren Sie in Kapitel 12.

1.10 DXUT gemeinsam nutzen

Wenn Sie nur ein einziges Spiel programmieren, an dem Sie über Jahre hinweg arbeiten, ist es Ihnen vermutlich recht, dass der Sample Browser die DXUT-Dateien in Ihrem Projektverzeichnis abgelegt hat.

Wenn Sie aber mehrere kleinere Projekte nacheinander durchführen (z.B. wenn Sie die Beispiele dieses Buchs durcharbeiten) und insbesondere wenn Sie dabei sogar noch eigene Änderungen am DXUT-Code vornehmen, dann kann es sich als störend auswirken, dass von diesen Dateien jedes Mal wieder eine eigene Kopie ins Projektverzeichnis gestellt wird.

1.10.1 DXUT in einem gemeinsamen Verzeichnis verwenden

Der einfachste Weg, den DXUT-Quellcode für mehrere Projekte gemeinsam zu nutzen, besteht darin, diesen in einem zentralen Verzeichnis aufzubewahren und in Ihre Projekte nur jeweils einen Verweis auf diese Dateien aufzunehmen.

Mit Visual Studio ist das ohne Weiteres möglich: Kopieren Sie einfach den kompletten DXUT-Ordner aus einem beliebigen Projekt an einen zentralen Speicherort (bei den Beispieldateien zum Buch ist das ein Ordner auf derselben Ebene wie die Kapitelordner).

Fügen Sie dann in Ihr Projekt einen leeren Ordner `DXUT` ein, indem Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt klicken und aus dem Kontextmenü den Befehl HINZUFÜGEN|NEUER FILTER wählen. Klicken Sie dann wiederum mit der rechten Maustaste auf diesen Ordner und wählen Sie aus dem Kontextmenü den Befehl HINZUFÜGEN|VORHANDENES ELEMENT. Wenn Sie nun in dem angezeigten Dateiauswahldialog die zentral gespeicherten DXUT-Dateien auswählen, werden diese als Verweis ins Projekt aufgenommen. Jede Änderung wirkt sich somit auf alle Projekte aus, mit denen diese Dateien auf die beschriebene Weise verknüpft werden.

So können Sie auch beim Erscheinen einer neuen Version des DirectX-SDK leicht die DXUT-Dateien austauschen und so von allen Bugfixes profitieren.

Bei den auf Grundlage des SIMPLESAMPLE oder EMPTYPROJECT erstellten Projekten müssen Sie allerdings auch noch eine kleine Änderung im Quelltext vornehmen: In der Ressourcendatei (`SimpleSample.rc` oder `EmptyProject11.rc` im Ordner `Resource Files`) wird auf das Symbol `directx.ico` im Ordner `DXUT\Optional` verwiesen. Ändern Sie diese Zeile so, dass die Datei an dem neuen, zentralen Speicherort gefunden wird. Bei den Beispielen auf der Buch-CD beispielsweise befindet sich der DXUT-Ordner auf derselben Ebene wie die Kapitelverzeichnisse – also zwei Ebenen oberhalb der Projektverzeichnisse und heißt `DXUT11`. In diesem Fall muss der Pfad wie folgt angepasst werden:

```
IDI_MAIN_ICON    ICON    "..\\..\\DXUT11\\Optional\\directx.ico"
```

(Die Pfadangabe `..` steht jeweils für das übergeordnete Verzeichnis.)

Für die Beispiele dieses Buchs benötigen Sie normalerweise die folgenden DXUT-Dateien:

- Aus dem Ordner `Core`: alle (`DXUTmisc.cpp`, `DXUT.cpp`, `DXUTDevice9.cpp` und `DXUTDevice11.cpp`)
- Aus dem Ordner `Optional`: `DXUTcamera.cpp`, `DXUTres.cpp`, `SDKmesh.cpp` sowie `SDKmisc.cpp`, `DXUTgui.cpp` und `DXUTsettingsdlg.cpp`.

Die dazugehörenden Headerdateien müssen nicht unbedingt ins Projekt aufgenommen werden, da sie ja inkludiert werden – es ist aber praktisch, die Headerdateien ebenfalls im Projekt zu haben, weil man sie dann vom Projektmappen-Explorer aus jederzeit bequem öffnen kann.

Auch die Datei `dpiaware.manifest` können Sie übrigens (statt aus dem lokalen `DXUT`-Verzeichnis) aus dem gemeinsam genutzten Verzeichnis einbinden.

Include-Pfad

Der Pfad zu den Headerdateien muss aber auch in den Projekteigenschaften angepasst werden. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt und wählen Sie im Kontextmenü den Befehl EIGENSCHAFTEN. Stellen Sie dann unter KONFIGURATIONSEIGENSCHAFTEN|C/C++|ALLGEMEIN den Eintrag ZUSÄTZLICHE INCLUDEVERZEICHNISSE ein auf den Speicherort, an den Sie die DXUT-Dateien kopiert haben. Mehrere Pfade können dabei, durch Semikolon getrennt, hintereinander angegeben werden. Bei den Beispielen auf der Buch-CD lautet der Eintrag `..\..\DXUT11\Core;..\..\DXUT11\Optional`, also befinden sich die DXUT-Dateien in den Verzeichnissen `DXUT11\Core` und `DXUT11\Optional` zwei Ebenen oberhalb des Projektverzeichnisses.

Hinweis

Der Original-Eintrag beim EMPTYPROJECT-Beispiel lautet `DXUT\Core;DXUT\Optional` – ohne den Vorsatz `..\..\`. Dort liegen die DXUT-Verzeichnisse *innerhalb* des Projektverzeichnisses statt zwei Ebenen darüber.

1.10.2 DXUT als Bibliothek kompilieren

Wenn Sie den DXUT-Code im Normalfall nicht bearbeiten, sondern einfach nur in Ihrem Projekt verwenden oder gar mit Ihrem Spiel weitergeben möchten, können Sie auch aus den entsprechenden Dateien eine Bibliothek (Library) kompilieren, die dann durch den Linker zum Projekt hinzugebunden wird.

In den Unterverzeichnissen `Core` und `Optional` des DXUT-Ordners finden Sie dazu je ein Projekt (`.vcproj`) und eine dazugehörende Projektmappe (`.sln`). Sie können die Projektmappe in Visual Studio öffnen oder aber einfach (mit dem Befehl HINZUFÜGEN|VORHANDENES PROJEKT im Kontextmenü der Projektmappe) die Projekte in Ihre eigene Projektmappe aufnehmen.

Beim Erstellen der Projekte entsteht in dem Unterordner `Debug` eine LIB-Datei, die Sie über die Linkereinstellungen (die Sie über den Befehl EIGENSCHAFTEN im Kontextmenü des Projekts anzeigen) in das Projekt einbinden.

1.11 Das Effekt-Framework

Wie Sie in den vorhergehenden Beispielen gesehen haben, ist der Programmcode zum Laden und Kompilieren des Shaders, zum Anlegen und Füllen der Konstantenbuffer, zum Setzen der Vertex- und Pixelshader, des Sampler State, des Shader Resource View für die Textur usw. doch recht aufwändig. Bei großen Projekten mit vielen unterschiedlichen Shadern gilt das natürlich in noch viel höherem Maße. Um die Aufgabe etwas zu erleichtern, stellt DirectX das *Effect-Framework* zur Verfügung. Es bildet die Schnittstelle zwischen dem C+-Programm und dem Shadercode.

Hinweis

Das Beispielprogramm zu diesem Abschnitt finden Sie auf der Begleit-CD im Verzeichnis `Kap01_12`.

Effekt und Technique

Ein Effekt repräsentiert dabei den gesamten aus einer Datei eingelesenen und kompilierten Shadercode mit einem oder mehreren Vertexshadern, Pixelshadern (sowie gegebenenfalls weiteren Shadertypen) und mindestens einer so genannten *Technique*, die angibt, welcher Vertexshader mit welchem Pixelshader für einen bestimmten Rendervorgang zusammengehört.

Dabei kann eine Shaderdatei durchaus mehrere solche Techniques enthalten, die unterschiedliche Techniken (mit jeweils eigenen Shaderfunktionen) darstellen, die wahlweise auf die 3D-Objekte angewendet werden können. So kann beispielsweise eine Shaderdatei Techniques sowohl für die einfarbige Darstellung als auch für eine Darstellung mit Textur enthalten. Beim Anzeigen eines Objekts (*Rendern*) muss man sich aber letztlich immer für eine bestimmte Technique entscheiden.

Die FX-Datei

Das SIMPLESAMPLE-Beispiel im DirectX-SDK unterstützt, wie Sie schon gesehen haben, neben der DX11-Version auch jeweils eine DX9-Version der Programme, die für ältere

Grafikkarten gedacht ist. Zwar verwendet die DX11-Version (jedenfalls in der zum Zeitpunkt der Entstehung dieses Buchs vorliegenden Ausgabe des SDK) das Effekt-Framework nicht, wohl aber die DX9-Version. Sehen Sie sich einmal die dafür vorgesehene Datei `SimpleSample.fx` an, um einen Eindruck vom Aufbau einer solchen Effektdatei zu bekommen.

Wir könnten diese fertige Effektdatei im Prinzip auch für unsere DX11-Programme verwenden – allerdings verwendet sie das ältere Shader Model 2. Sie sehen das an der Definition der Technique unten in der Datei:

```
technique RenderScene
{
   pass P0
   {
      VertexShader = compile vs_2_0 RenderSceneVS();
      PixelShader  = compile ps_2_0 RenderScenePS();
   }
}
```

Der Vertexshader wird hier mit dem Target `vs_2_0` und der Pixelshader mit dem Target `ps_2_0` kompiliert.

Gerade die interessanten Fähigkeiten neuerer Grafikkarten können aber erst mit den aktuellen Shader Models 4 und 5 genutzt werden.

Um die FX-Datei an DirectX11 anzupassen, müssen wir also zunächst einmal das Target umstellen auf `vs_4_0` und `ps_4_0`. Falls Sie eine DX11-Grafikkarte haben, können Sie auch `vs_5_0` und `ps_5_0` verwenden – auf DX10-Karten geht das dann allerdings nur mit dem Reference-Device. Das Shader Model 5 wird in den Beispielen dieses Buchs deshalb nur dort eingesetzt, wo es tatsächlich notwendig ist.

Hinweis

Um auf DX10-Grafikkarten das Reference-Device zu erzwingen, damit auch Shader-Model-5-Code lauffähig ist, ändern Sie in `wWinMain` den ersten Parameter von `DXUTCreateDevice` von `D3D_FEATURE_LEVEL_10_0` in `D3D_FEATURE_LEVEL_11_0`.

Leider hat sich die HLSL-Syntax seit dem Shader Model 2 in vielerlei Hinsicht geändert – es sind also an der FX-Datei noch eine ganze Reihe weiterer Anpassungen vorzunehmen. Das Ergebnis finden Sie in der Datei `Kap01_12.fx` auf der-Buch-CD:

```
//--------------------------------------------------------------------------
// Globale Variablen
//--------------------------------------------------------------------------
cbuffer cbPerObject
{
   matrix  g_mWorldViewProjection;
   matrix  g_mWorld;
   float4  g_MaterialAmbientColor;
   float4  g_MaterialDiffuseColor;
}
```

```
cbuffer cbPerFrame
{
   float3           g_vLightDir;
   float            g_fTime;
   float4           g_LightDiffuse;
};

Texture2D g_txDiffuse;

//--------------------------------------------------------------------------
// Textur-Sampler
//--------------------------------------------------------------------------
SamplerState g_samLinear
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};

//--------------------------------------------------------------------------
// Ein- und Ausgabestrukturen
//--------------------------------------------------------------------------
struct VS_INPUT
{
   float4 Position    : POSITION; // vertex position
   float3 Normal      : NORMAL;   // this normal comes in per-vertex
   float2 TextureUV   : TEXCOORD0;// vertex texture coords
};

struct VS_OUTPUT
{
   float4 Position    : SV_POSITION; // vertex position
   float4 Diffuse     : COLOR0;     // vertex diffuse color
                              //(note that COLOR0 is clamped from 0..1)
   float2 TextureUV   : TEXCOORD0;   // vertex texture coords
};

//--------------------------------------------------------------------------
// Vertexshader
//--------------------------------------------------------------------------
VS_OUTPUT RenderSceneVS(VS_INPUT input)
{
   VS_OUTPUT Output;
   float3 vNormalWorldSpace;

   // Transform the position from object space to homogeneous projection space
   Output.Position = mul(input.Position, g_mWorldViewProjection);

   // Transform the normal from object space to world space
```

```
    vNormalWorldSpace = normalize(mul(input.Normal, (float3x3)g_mWorld));
                                    // normal (world space)
    // Calc diffuse color
    Output.Diffuse.rgb = g_MaterialDiffuseColor * g_LightDiffuse *
       max(0,dot(vNormalWorldSpace, g_vLightDir)) + g_MaterialAmbientColor;
    Output.Diffuse.a = 1.0f;

    // Just copy the texture coordinate through
    Output.TextureUV = input.TextureUV;

    return Output;
}

//--------------------------------------------------------------------------
// Pixelshader
//--------------------------------------------------------------------------
float4 RenderScenePS(VS_OUTPUT In) : SV_TARGET
{
    // Lookup mesh texture and modulate it with diffuse
    return g_txDiffuse.Sample(g_samLinear, In.TextureUV) * In.Diffuse;
}

//--------------------------------------------------------------------------
// Technique
//--------------------------------------------------------------------------
technique11 RenderScene
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_4_0, RenderSceneVS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, RenderScenePS()));
    }
}
```

Listing 1.32: Effekt-Datei für DirectX11

Wie Sie sehen, sind die Shader selbst, die Konstantenbuffer, die Texturvariable und der Sampler State im Wesentlichen identisch mit denjenigen aus der Datei `Simple-Sample.hlsl` (daher auch die englischen Original-Kommentare).

Allerdings wird der Sampler State jetzt gar nicht mehr aus dem C++-Code übergeben, sondern komplett in der Effektdatei definiert.

Hinweis

Auch den Blend State, den Rasterizer State und den Depth-Stencil State (siehe Abschnitt 1.5) könnten wir in der Effektdatei definieren. Davon wird in den Beispielen dieses Buchs aber im Normalfall kein Gebrauch gemacht, weil das Setzen dieser States aus dem C++-Programm uns eine größere Kontrolle darüber ermöglicht, wann die States jeweils umgeschaltet werden – dies ist ein wichtiger Aspekt im Hinblick auf die Performance, da ein häufiges Umschalten der Render States sich negativ auswirkt.

Für den Sampler State (der in den Beispielen ohnehin fast immer derselbe ist) machen wir aber von der größeren Bequemlichkeit Gebrauch, die uns das Effekt-Framework bietet.

Kompilieren des Effekt-Frameworks

Anders als in früheren DirextX-Versionen ist in DX11 (Februar-2010-Release) das Effekt-Framework nicht Bestandteil der eigentlichen DirectX-Installation, sondern wird mit dem DirectX-SDK im Quelltext ausgeliefert – Sie müssen also die Bibliothek erst selbst aus diesen Quelltexten kompilieren. Der Ordner `Samples\C++\Effects11` innerhalb des Verzeichnisses der SDK-Installation enthält dazu das Projekt `Effects11_2008.vcproj`, das Sie am besten in Ihre Visual-Studio-Projektmappe aufnehmen, damit Sie beim Debuggen auch problemlos in dessen Quelltextdateien verzweigen können.

Hinweis

Wenn Sie das Projekt nicht aus der in demselben Verzeichnis befindlichen Projektmappe `Effects11_2008.sln` heraus, sondern aus Ihrer eigenen Projektmappe erstellen, so müssen Sie in den Projekteigenschaften unter KONFIGURATIONSEIGENSCHAFTEN|C/C++|ALLGEMEIN den Eintrag ZUSÄTZLICHE INCLUDEVERZEICHNISSE ändern in `"$(ProjectDir)Binary";"$(ProjectDir)Inc"`.

Beim Erstellen dieses Projekts entsteht die Datei `D3DX11Effects.lib` (Release-Konfiguration) bzw. `D3DX11EffectsD.lib` (Debug-Konfiguration).

Diese Datei muss nun zu Ihren Projekten hinzugelinkt werden. Sie können das tun, indem Sie in den Projekteigenschaften unter Konfigurationseigenschaften KONFIGURATIONSEIGENSCHAFTEN|LINKER|EINGABE die Datei im Feld ZUSÄTZLICHE ABHÄNGIGKEITEN mit in die Liste aufnehmen.

Erweitern Sie auch den Linker-Eingabepfad (KONFIGURATIONSEIGENSCHAFTEN|LINKER|ALLGEMEIN|ZUSÄTZLICHE BIBLIOTHEKSVERZEICHNISSE) um das Verzeichnis, in dem die LIB-Datei nach dem Erstellen liegt. Wo genau das ist, hängt von den Projekteinstellungen des Projekts EFFECTS11 ab (normalerweise das `Debug`-Unterverzeichnis in Ihrem Projektmappenverzeichnis).

Ebenso müssen Sie dem Compiler mitteilen, wo die Include-Datei `d3dx11effect.h` zu finden ist, die von allen Projekten eingebunden werden muss, die das Effekt-Framework verwenden (KONFIGURATIONSEIGENSCHAFTEN|C/C++|ZUSÄTZLICHE INCLUDEVERZEICHNISSE).

Hinweis

Noch einfacher als oben beschrieben können Sie die LIB-Datei in Ihr Projekt einbinden, wenn das Projekt `Effects11_2008.vcproj` mit in Ihrer Projektmappe enthalten ist: In diesem Fall brauchen Sie nur im Kontextmenü des jeweiligen Beispielprogramms im Projektmappen-Explorer auf PROJEKTABHÄNGIGKEITEN zu klicken und das Projekt EFFECTS11_2008 in der Liste auszuwählen.

Da Sie sowohl den Linker-Eingabepfad als auch den Pfad für die Include-Datei in allen folgenden Projekten benötigen werden, empfiehlt es sich, diese Einstellungen einmalig im OPTIONEN-Dialogfeld (EXTRAS|OPTIONEN im Visual-Studio-Hauptmenü) unter PROJEKTE UND PROJETKMAPPEN|VC++-VERZEICHNISSE vorzunehmen (siehe auch Abschnitt 1.1.5).

Verwendung im Programm

Auf Details des HLSL-Programmcodes wird in Kapitel 8 eingegangen. Hier wollen wir nun zunächst einmal sehen, wie das Effekt-Framework dabei helfen kann, die Shadervariablen mit Werten zu versorgen.

Sie können dazu entweder das SIMPLESAMPLE-Beispiel verwenden und allen Shader-bezogenen Programmcode daraus entfernen, oder Sie beginnen gleich mit dem EMPTYPROJECT und führen die in Abschnitt 1.4.2 beschriebenen Schritte aus. Auch im letzteren Fall können Sie natürlich den gesamten Shader-bezogenen Teil und auch die Programmteile für den Sampler State weglassen. Die Programmzeile für das Anlegen des Vertexlayouts (siehe Listing 1.13) werden wir noch etwas abändern müssen (Listing 1.36), alles andere bleibt aber genauso.

Zunächst einmal müssen wir die Headerdatei des Effekt-Frameworks inkludieren:

```
#include "d3dx11effect.h"
```

Fügen Sie nun oben in die Quelltextdatei Variablen für den Effekt und die Technique ein:

```
ID3DX11Effect*          g_pEffect = NULL;
ID3DX11EffectTechnique* g_pTechnique = NULL;
```

Einlesen des Effekts

In `OnD3D11CreateDevice` wird der Effekt kompiliert. Der Programmcode ähnelt dem bereits bekannten für das separate Kompilieren von Vertex- und Pixelshader. Allerdings wird der Funktion `D3DX11CompileFromFile` für den vierten Parameter (die Einstiegsfunktion) eine leere Zeichenkette übergeben, und an Stelle von `CreateVertexShader` und `CreatePixelShader` verwenden wir jetzt `D3DX11CreateEffectFromMemory`:

```
WCHAR str[MAX_PATH];
DXUTFindDXSDKMediaFileCch(str, MAX_PATH, L"Kap02_01.fx");
DWORD dwShaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;
#if defined(DEBUG) || defined(_DEBUG)
dwShaderFlags |= D3D10_SHADER_DEBUG;
#endif

ID3DBlob* pBlob = NULL;
V_RETURN(D3DX11CompileFromFile(str, NULL, NULL, "", "fx_5_0", dwShaderFlags, 0,
   NULL, &pBlob, NULL, NULL));
V_RETURN(D3DX11CreateEffectFromMemory(pBlob->GetBufferPointer(),
   pBlob->GetBufferSize(), dwShaderFlags, pd3dDevice, &g_pEffect));
```

Listing 1.33: Einlesen und Kompilieren eines Effekts

Effektparameter

Um Werte für die globalen Variablen des Effekts aus dem C++-Programm an den Shader übergeben zu können, deklarieren wir nun passende Effektvariablen:

```
//Effektvariablen
ID3DX11EffectMatrixVariable*       g_pmWorldViewProj = NULL;
ID3DX11EffectMatrixVariable*       g_pmWorld = NULL;

//Material
ID3DX11EffectVectorVariable* g_pDiffuseVariable = NULL;

//Textur
ID3D11ShaderResourceView* g_pRV = NULL;
ID3DX11EffectShaderResourceVariable* g_ptxDiffuseVariable = NULL;

//Licht
ID3DX11EffectVectorVariable* g_pLightDirVariable = NULL;
ID3DX11EffectVectorVariable* g_pLightDiffuseVariable = NULL;
ID3DX11EffectVectorVariable* g_pAmbientVariable = NULL;
```

Listing 1.34: Deklaration der Effektvariablen

Den Programmcode zur Verknüpfung dieser Variablen mit den dazugehörenden Shaderparametern fügen wir in `OnD3D10CreateDevice` ein:

```
ID3D11DeviceContext* pd3dImmediateContext = DXUTGetD3D11DeviceContext();

//Effektvariablen
g_pmWorldViewProj = g_pEffect->
   GetVariableByName("g_mWorldViewProjection")->AsMatrix();
g_pmWorld = g_pEffect->
   GetVariableByName("g_mWorld")->AsMatrix();

//Licht
g_pLightDirVariable = g_pEffect->GetVariableByName("g_vLightDir")->AsVector();
g_pLightDiffuseVariable = g_pEffect->GetVariableByName("g_LightDiffuse")->
   AsVector();
g_pAmbientVariable = g_pEffect->GetVariableByName("g_MaterialAmbientColor")->
   AsVector();

//Material
g_pDiffuseVariable = g_pEffect->GetVariableByName("g_MaterialDiffuseColor")->
   AsVector();

//Textur
DXUTFindDXSDKMediaFileCch(str, MAX_PATH, L"Genetica\\Born to Rule.jpg");
DXUTGetGlobalResourceCache().CreateTextureFromFile(pd3dDevice,
   pd3dImmediateContext, str, &g_pRV);
g_ptxDiffuseVariable = g_pEffect->GetVariableByName("g_txDiffuse")->
   AsShaderResource();
```

Listing 1.35: Effektvariablen verknüpfen

Die Effektvariable für die Textur brauchen wir nur, wenn wir eine externe Textur verwenden wollen. In diesem Fall muss natürlich auch die Textur selbst geladen werden – wie bereits bekannt mit `CreateTextureFromFile`.

Der Technique-Variablen wird (ebenfalls in `OnD3D10CreateDevice`) eine in der Effektdatei vorhandene Technique zugewiesen:

```
g_pTechnique = g_pEffect->GetTechniqueByName("RenderScene");
```

Auch das Anlegen des Vertexlayouts erfolgt jetzt etwas anders als bei den Beispielen ohne das Effekt-Framework (wobei die Struktur `layout` dieselbe bleibt wie in Listing 1.13):

```
UINT numElements = sizeof(layout)/sizeof(layout[0]);
D3DX11_PASS_DESC PassDesc;
g_pTechnique->GetPassByIndex(0)->GetDesc(&PassDesc);
pd3dDevice->CreateInputLayout(layout, numElements, PassDesc.pIAInputSignature,
    PassDesc.IAInputSignatureSize, &g_pLayout11);
```

Listing 1.36: Anlegen des Vertexlayouts

In `OnD3D11FrameRender` werden die Variablen dann mit Werten versorgt:

```
D3DXMatrixTranslation(&g_mWorld, 0, 0, 0);

D3DXMATRIX mView = *g_Camera.GetViewMatrix();
D3DXMATRIX mProj = *g_Camera.GetProjMatrix();
D3DXMATRIX mWorldViewProjection = g_mWorld * mView * mProj;

g_pmWorldViewProj->SetMatrix((float*)&mWorldViewProjection);
g_pmWorld->SetMatrix((float*)&mWorld);

//Licht
D3DXVECTOR3 LightDir(1.0f, 0.0f, 0.0f);
if(g_pLightDirVariable)
   g_pLightDirVariable->SetFloatVector(LightDir);

D3DXVECTOR3 LightDiffuse(1.0f, 0.0f, 0.0f); //Rot
if(g_pLightDiffuseVariable)
   g_pLightDiffuseVariable->SetFloatVector(LightDiffuse);

//Material
D3DXVECTOR3 Ambient(0.0f, 0.0f, 1.0f); //Blau
if(g_pAmbientVariable)
   g_pAmbientVariable->SetFloatVector(Ambient);

D3DXVECTOR3 Diffuse(1.0f, 1.0f, 1.0f); //Weiss
if(g_pDiffuseVariable)
   g_pDiffuseVariable->SetFloatVector(Diffuse);

//Textur
if(g_ptxDiffuseVariable)
  g_ptxDiffuseVariable->SetResource(g_pRV);
```

Listing 1.37: `OnD3D11FrameRender`

Statt Vertexshader, Pixelshader und Sampler State an den Device Context zu übergeben, brauchen wir jetzt vor dem Rendern nur noch die Technique anzuwenden:

```
g_pTechnique->GetPassByIndex(0)->Apply(0, pd3dImmediateContext);
```

Denken Sie daran, beim Rendern des Mesh nicht die interne, sondern die eigene Textur zu verwenden (indem Sie als zweiten Parameter an `Render` nicht `0`, sondern `-1` übergeben oder die letzten drei Parameter ganz weglassen):

```
g_Mesh.Render(pd3dImmediateContext);
```

Aufräumarbeiten

Zuletzt müssen wir noch in `OnD3D11DestroyDevice` den Effekt wieder freigeben:

```
SAFE_RELEASE(g_pEffect);
```

Da das Effekt-Framework den Programmcode doch um einiges übersichtlicher macht, werden wir dieses in den folgenden Kapiteln überwiegend verwenden. Solange wir immer nur den Standardeffekt verwenden (bis zum Kapitel 8), können Sie die oben vorgestellte Effektdatei einfach für jedes neue Programm kopieren. Sie können auch die Datei in ein gemeinsames Verzeichnis legen und in allen Projekten wiederverwenden. Die Beispiele in den folgenden Kapiteln benutzen, sofern sie nicht eine eigene Effektdatei mitbringen, die Datei `SimpleSample11.fx` im Verzeichnis `MyEngine/Effects` auf der Buch-CD, die weitgehend derjenigen aus diesem Beispiel gleicht.