3 Rahmenanwendung der ZFXEngine

»Was man bekommt, wenn man nicht bekommt, was man will, ist Erfahrung.« (Désirée Nick)

Kurz überblickt ...

In diesem Kapitel werden die folgenden Themen behandelt:

- Was ist ein Interface?
- Erstellung statischer und dynamischer Bibliotheken
- Anzeigen von Dialogen zur Interaktion mit Programmen
- Entwicklung einer Bibliothek als Renderer für die *ZFXEngine*
- Initialisierung und Starten von Direct3D 9 über eine DLL
- Schreiben eines Rahmenprogramms für die *ZFXEngine*

3.1 Begriffsbestimmung Interface

Die Bezeichnung »Interface« wird im Bereich der Informatik für eine Vielzahl von Dingen verwendet. So ist ein GUI beispielsweise ein *Graphical User Interface*, also nichts anderes als eine Ansammlung von Buttons, Menüs und anderen Steuerelementen, mit deren Hilfe der Benutzer am Bildschirm das Programm bedienen kann. Die Abkürzung API wiederum steht für *Application Programming Interface*. Damit meint man diejenigen Funktionen, die eine Bibliothek von Funktionen einer anderen Applikation für die Erledigung diverser Aufgaben zur Verfügung stellt. Die DirectGraphics-Komponente von DirectX ist beispielsweise eine solche Grafik-API, über deren Funktionen wir in begrenztem Umfang unsere Grafikkarte ansprechen können, ohne dass wir selbst mit dem Treiber der Grafikkarte kommunizieren müssten.

Interfaces

Daher nennt man ein Interface in der deutschen Sprache »Schnittstelle«. Ein Interface ist immer eine Schnittstelle zwischen zwei Objekten, die auf irgendeine Weise miteinander kommunizieren müssen. Wenn der Benutzer am Bildschirm sein Programm dazu bringen möchte, etwas für ihn zu tun, dann muss er das dem Programm über die GUI tun, beispielsweise durch Anklicken eines bestimmten Buttons. Andersherum muss das Programm,

wenn es dem Benutzer etwas mitteilen möchte, ein Dialog-Fenster öffnen, um so über die GUI mit dem Benutzer zu kommunizieren.

Abstrakte Klassen

Im strengen programmiertechnischen Sinne bezeichnet eine API aber mehr oder weniger lediglich eine abstrakte Klasse einer objektorientierten Programmiersprache. Wir verwenden in diesem Buch die Sprache C++, also orientieren wir die folgenden Erklärungen an diesem konkreten Beispiel. Eine abstrakte Klasse ist eine Klasse, die nicht instanziiert werden kann. Man kann also kein Objekt dieser Klasse erzeugen. Das erreicht man, indem man in einer Klasse eine rein virtuelle Member-Funktion deklariert, selbst wenn die Klasse noch andere, nicht rein virtuelle Methoden beinhaltet.

Hm, so kommen wir irgendwie nicht weiter. Ein Fremdwort folgt hier auf das nächste. Vielleicht sollten wir Schritt für Schritt vorgehen.

Schlüsselwort virtual

Durch das Schlüsselwort `virtual` in C++ können wir Member-Funktionen einer Klasse virtuell machen. Das hat nur dann eine Auswirkung, wenn abgeleitete Klassen existieren, die dieselbe Methode implementieren. Normalerweise entscheidet der Typ des Pointers, über den eine Methode aufgerufen wird, aus welcher Klasse die Methode verwendet wird. So kann man Objekte abgeleiteter Klasse durchaus über einen Pointer vom Typ der Basisklasse ansprechen. In diesem Fall würde jedoch die Methode aus der Basisklasse verwendet, was nicht immer erwünscht sein muss. Ist die Methode in der Basisklasse mit dem Schlüsselwort `virtual` deklariert, so wird die Methode aus der Klasse genommen, von der das Objekt instanziiert wurde, auf das der Pointer zeigt und nicht mehr aus der Klasse des Pointers. So kann man namensgleiche Methoden in der Basisklasse und in davon abgeleiteten Klassen haben, alle Objekte dieser Klasse über einen Pointer vom Typ der Basisklasse ansprechen und dennoch die Methoden aus der jeweils richtigen Klasse aufrufen.

Rein virtuelle Member-Funktionen

Als rein virtuell bezeichnet man eine Member-Funktion genau dann, wenn sie einerseits mit dem Schlüsselwort `virtual` definiert wurde, aber andererseits in der Basisklasse gar keine eigene Implementierung hat. Die Methode ist also in der Basisklasse nicht definiert, aber deklariert. Um dies dem Compiler anzuzeigen, erhält der Prototyp der Member-Funktion den Zusatz **= 0** in der Deklaration. Eine rein virtuelle Funktionsdeklaration sieht also wie folgt aus:

```
virtual vector3d GetCenterpoint(void) = 0;
```

Sobald eine Klasse eine rein virtuelle Member-Funktion enthält, wird sie als abstrakte Klasse bezeichnet. Von einer solchen abstrakten Klasse können keine Objekte erzeugt werden, da eine abstrakte Klasse eben auch Methoden anbietet, die gar nicht implementiert sind. Man kann aber natürlich einen Pointer vom Typ dieser Klasse erzeugen.

Wozu Interfaces?

Nun stellt sich die Frage nach dem Sinn einer abstrakten Klasse, wenn man ja quasi gar nicht mit ihr arbeiten kann. Interessant wird es dann, wenn

man eine solche abstrakte Klasse als eine Art Interface-Definition versteht. Man kann beispielsweise eine abstrakte Klasse mit diversen Attributen und rein virtuellen Funktionen definieren. Diese abstrakte Klasse ist dann eine zwingende Vorlage, was für Member-Funktionen eine Ableitung dieser Basisklasse mindestens definieren muss. Bleibt die Frage, warum man das überhaupt machen sollte, und nicht gleich die korrekte Klasse implementiert. Nun sind wir beispielsweise bei der verteilten Arbeit in einem Team angelangt. Unser Ziel ist es, ein schönes 3D-Spiel zu programmieren. Dazu arbeitet ein Teil des Teams an dem Game-Code, also an der Ablauflogik des Spiels, und ein anderer Teil des Teams programmiert die 3D-Engine. Das erstgenannte Teilteam muss aber bereits frühzeitig in der Entwicklung wissen, auf welche Funktionen der 3D-Engine es zurückgreifen kann und welche Parameterlisten diese Funktionen verwenden. Also wird zu Beginn der Arbeit ein Interface in Form einer abstrakten Klasse definiert, das exakt vorschreibt, welche Funktionen die 3D-Engine später zwingend bieten muss. Basierend auf dieser Vorgabe, entwickelt das Teilteam der 3D-Engine eine Ableitung der abstrakten Klasse, in der die entsprechenden Member-Funktionen implementiert werden. Das zweite Teilteam kann aber bereits mit Pointern der abstrakten Klasse im Game-Code arbeiten. So ist sichergestellt, dass die beiden separat entwickelten Teilkomponenten des Spiels später problemlos zusammenpassen werden.

3.2 Unser Interface

Interface für den Renderer

Jetzt dürftet ihr schon eine ganz gute Vorstellung davon haben, wozu man Interfaces verwendet. Im Verlauf dieses Kapitels werden wir lernen, wie man ein solches Interface erzeugen und eine abgeleitete Klasse definieren kann, die dieses Interface implementiert. Unser Ziel ist es, unsere 3D-Engine so unabhängig wie möglich von einer bestimmten Grafik-API oder API-Version zu machen. Wir werden also ein Interface definieren, das uns sämtliche Funktionalität für die Ausgabe von Grafik zur Verfügung stellt, die wir für unsere Engine brauchen. Von diesem Interface können wir dann beliebig viele Klasse ableiten, die die vorgeschriebenen Funktionen implementieren. Auf diese Weise können wir beispielsweise in einer Klasse einen DirectX-Renderer programmieren, in einer anderen Klasse einen OpenGL-Renderer und in einer dritten Klasse einen Software-Renderer.

Dynamische Auswahl eines Renderers

Der große Vorteil ist nun folgender: Wir können die jeweiligen Renderer in einer jeweils eigenen Bibliothek erstellen. Unsere *ZFXEngine* kann nun mittels des Interfaces komplett programmiert werden. Danach können wir beliebig eine der drei oben genannten Bibliotheken in das Projekt einbinden und erhalten somit Zugriff auf einen von drei Renderern. Der Engine selbst ist es vollkommen egal, ob dort DirectX, OpenGL oder eine eigene Software-Implementierung verwendet wird. Und wir werden noch einen Schritt weiter gehen: Die Implementierung unserer Renderer werden wir zu einer

DLL (Dynamic Link Library) kompilieren. Haben wir schließlich die *ZFX-Engine* oder ein ganzes Spiel unter Verwendung der Engine fertig gestellt, können wir es ruhigen Gewissens kompilieren. Die Render-DLL wird ja erst zur Laufzeit des Programms geladen. Ohne die Engine oder das Spiel neu kompilieren zu müssen, können wir einfach die Render-DLL(s) auswechseln und durch neue Implementierungen ersetzen. Die Engine bzw. das Spiel bleibt weiterhin ausführbar, nutzt jedoch sofort die neue Implementierung aus – und das, ohne dass auch nur eine einzige Änderung am Engine- oder Game-Code notwendig ist.

Analog kann man diese Technik auch für jede andere Komponente eines Projekts durchführen. Man könnte also die Musik- und Soundausgabe in einer DLL kapseln, ebenso wie die Inputverarbeitung oder gar die künstliche Intelligenz. Das macht aber relativ wenig Sinn, denn diese Komponenten sind nicht derartigen Veränderungen unterworfen, wie der Renderer das sein wird. Beim Erscheinen einer neuen DirectX-Version oder einer neuen OpenGL-Spezifikation kann es durchaus sehr viel Sinn machen, den Renderer eines laufenden Projekts neu zu schreiben. Zum Beispiel kann so eine neue Version zusätzliche Vertex- und Pixel-Shader bieten oder einfach nur Optimierungen enthalten. Ebenso ist es denkbar, dass man zunächst eine Art Prototyp-Renderer entwickelt, wobei die kurze Entwicklungszeit das wichtigste Kriterium ist. Mit diesem Prototyp kann man die Engine oder ein konkretes Spiel dann bereits in der frühen Entwicklungsphase am Bildschirm testen. Zeitgleich wird dann ein optimierter Renderer entwickelt, der später lediglich die DLL des Prototyps ersetzt.

Selbst definierte Renderer für die ZFXEngine

Für dieses Buch werde ich lediglich eine DLL verwenden, die die Render-Funktionalität mit Hilfe von DirectX implementiert. Die Schritte zur Erstellung einer solchen DLL laufen allerdings immer ganz analog ab. Wer sich einen eigenen Renderer unter Verwendung von OpenGL oder eigener Software-Routinen erstellen will, der muss lediglich die entsprechenden Funktionen in einer eigenen DLL implementieren und kann dann die komplette *ZFXEngine* unverändert mit seinem Renderer betreiben. So kann man natürlich auch die Performance verschiedener Implementierungen direkt vergleichen, indem man ein konkretes Projekt mit verschiedenen Render-DLLs unter identischen Bedingungen (z.B. auf derselben Hardware) testet.

Wichtig ist dabei jedoch eines: Ein solcher selbst definierter Renderer, also die konkrete Implementierung nach einem vorgegebenen Interface einer abstrakten Klasse, darf als öffentliche Methoden lediglich die rein virtuellen Member-Funktionen der abstrakten Basisklasse definieren. Lediglich als `private` oder `protected` deklarierte Member-Funktionen kann die abgeleitete Klasse neu einführen und verwenden, ebenso wie zusätzliche Member-Variablen (Attribute). Zusätzliche öffentliche Funktionen in den abgeleiteten Klassen wären ja nur zugänglich, wenn man einen Pointer des Typs der abgeleiteten Klasse verwenden würde. Sinn und Zweck der ganzen Angele-

genheit ist es aber, in der Engine einen Pointer vom Typ der abstrakten Basisklasse zu verwenden, unabhängig davon, welche Ableitung (DirectX oder OpenGL usw.) man als DLL in sein Projekt lädt und verwendet.

3.3 Der Arbeitsbereich für unsere Implementierung

Nach der langen Vorrede ist es nun so weit. Jetzt können wir uns an die Tasten machen und einen Renderer in einer DLL programmieren. Dafür gibt es im Grunde zwei Möglichkeiten: Wir können die DLL direkt in unserem Projekt verlinken, oder wir laden sie per Funktionsaufruf. Die erste Variante ist sicherlich die bequemere. Sie erfordert jedoch, dass unser Projekt zu der statischen Bibliothek gelinkt wird, die beim Erstellen der DLL erzeugt wird.

Statische und dynamische Bibliotheken

Eine Bibliothek ist ein Programm, das lediglich Klassen und Funktionen beinhaltet, die ein anderes Programm verwenden kann. Im Gegensatz zu einem ausführbaren Programm enthält eine Bibliothek keine Startfunktion und kann daher nicht autark ausgeführt werden.

Kompiliert man sein Projekt zu einer Bibliothek, so wird daraus wahlweise eine Datei mit der Endung `*.lib` (Static Library, statische Bibliothek) oder `*.dll` (Dynamic Link Library, dynamische Bibliothek) erzeugt. Diese Datei kann man dann in anderen Projekten verlinken, um die Klassen und Funktionen aus der Bibliothek verwenden zu können.

Eine statische Bibliothek wird dabei beim Kompilieren des ausführbaren Projekts, das die Bibliothek verwendet, in dessen kompilierte, ausführbare Datei mit eingebunden. Sie ist damit in der Release-Version direkt mit enthalten. Eine DLL hingegen wird nicht in die ausführbare Datei eingebunden. Sie wird vom Programm erst zur Laufzeit geladen und muss daher zusammen mit dem Programm ausgeliefert werden.

Da die fertig kompilierte DLL nicht direkt in ein ausführbares Projekt eingebunden wird, *weiß* dieses Programm auch nicht, welche Klassen und Funktionen die DLL enthält. Beim Kompilieren einer DLL wird daher auch immer eine namensgleiche Datei `*.lib` mit erzeugt. Diese Datei enthält quasi eine Aufstellung all der Klassen und Funktionen, die die DLL enthält. Möchte ein Projekt eine DLL verwenden, so muss das Projekt (nach der oben erstgenannten Methode) nicht zu der DLL gelinkt werden, sondern zu der zugehörigen statischen Bibliothek. Damit entfällt aber der Vorteil einer DLL, weil ein ausführbares Projekt eben doch neu kompiliert werden muss, wenn sich an der DLL etwas geändert hat. Die zugehörige statische Bibliothek muss ja neu in die ausführbare Datei eingebunden werden. Zudem werden auf diese Weise beim Start des Programms sofort alle DLLs geladen, mit deren `*.lib`-Dateien das Programm verlinkt ist. Hierbei verschwendet man unnötig Speicher, wenn man nicht wirklich alle diese DLLs benötigt.

Aus diesem Grunde muss man bei der Programmierung eines DirectX-Projekts beispielsweise immer zu diversen `*.lib`-Dateien linken, obwohl die eigentliche Implementierung der DirectX-Komponenten als DLLs vorliegt.

Dieses Problem der DLLs kann man jedoch umgehen. Es ist möglich, eine DLL erst zur Laufzeit eines Programms an einer beliebigen Stelle im Ablauf des Programms gezielt zu laden, ohne dessen `*.lib`-Datei verlinkt zu haben. Wir schauen uns gleich an, wie das funktioniert. Wichtig dabei ist, dass wir unserem Projekt dann noch irgendwie anders mitteilen müssen, welche Klassen und Funktionen unsere DLL enthält. Dazu kann man den Funktionen den Zusatz `__declspec(dllexport)` mit auf den Weg geben. Eine andere Alternative ist, dass man eine Datei mit der Endung `*.def` erstellt, die die exportierten Funktionen der DLL auflistet. Streng genommen ist es (mit beiden Varianten) unmöglich, eine Klasse aus einer DLL zu exportieren. Man ist hier auf Funktionen im C-Stil beschränkt. Diese Beschränkung kann man aber umgehen, indem man ein Interface verwendet. Das werden wir gleich noch im Quelltext sehen, aber ich werde es hier schon einmal allgemein beschreiben.

Klassen aus einer DLL verwenden

Wir definieren ein Interface, also eine abstrakte Klasse. Dieses Interface binden wir in das ausführbare Projekt, beispielsweise in Form einer Header-Datei, ein. In dem DLL-Projekt erzeugen wir eine Klasse, die von diesem Interface abgeleitet ist und dessen Funktionen implementiert. Nun schreiben wir eine reine C-Funktion für die DLL, die ein Objekt dieser Klasse instanziiert, eine Typumwandlung dieses Objekts in den Typ der abstrakten Klasse des Interfaces durchführt und das gecastete Objekt an den Aufrufer zurückgibt. Diese C-Funktion ist nun das Einzige, was wir von der DLL exportieren müssen. Über sie erhalten wir in unserem ausführbaren Projekt einen Pointer vom Typ des Interfaces, der aber auf ein Objekt der Klasse aus der DLL zeigt. Wir wissen aber, dass dieses Objekt alle Funktionen implementiert, die das Interface vorschreibt. Und genau diese Funktionen können wir aufrufen.

Das klingt vielleicht für den Anfang ein wenig kompliziert, aber wenn wir uns später das gesamte Projekt ansehen, dann wird die Sache schnell klar. Um es aber nicht ganz so einfach zu machen, werden wir hier noch eine statische Bibliothek zusätzlich verwenden. Die ganze Aufruferei von Funktionen für das Laden und Verwenden von DLLs zur Laufzeit ist nicht besonders schön. Vor allem ist es unkomfortabel für die Benutzer unserer DLL. Also schreiben wir den gesamten Code, der das Laden der DLL betrifft, in eine statische Bibliothek. Diese wird zwar fest in das ausführbare Projekt eingebunden, das unsere DLL verwenden möchte, wird sich allerdings nie mehr ändern müssen, selbst wenn sich unsere DLL verändert. Diese statische Bibliothek beinhaltet also nur eine Klasse, die das Handling der DLLs managt. Sie wird auch nicht sonderlich umfangreich sein, aber dennoch eine Menge unansehnlichen Codes vor dem Anwender unserer DLL verstecken. Zusätzlich schreiben wir dann, wie geplant, eine eigene DLL für jeden Renderer, den wir verwenden möchten.

Organisation der Dateien

Im Folgenden werde ich die notwendigen Schritte noch einmal explizit beschreiben, wenn sie notwendig werden. Aber es kann ja nicht schaden, hier schon einmal einen groben Überblick über das zu bekommen, was wir gleich tun werden. Als Erstes benötigen wir einen *Visual C++*-Arbeitsbereich. Zu Beginn erzeugen wir ein Projekt für eine statische Bibliothek. Diese wird unser Manager für das DLL-Handling. Dann fügen wir in diesen Arbeitsbereich ein neues Projekt ein – diesmal jedoch ein Projekt für eine dynamische Bibliothek. Diese wird unseren DirectX-Renderer enthalten, also die Implementierung unseres Interfaces. Für jeden weiteren Renderer müsste man in dem Arbeitsbereich ein weiteres neues DLL-Projekt einfügen. Sehen wir uns das nun etwas genauer an.[1]

ZFXRenderer, eine statische Bibliothek als Manager

Bisher haben wir ja immer von dem Renderer an sich gesprochen. Diese Namensgebung werden wir jetzt ein wenig korrigieren. Dabei orientieren wir uns ein wenig an der Namensgebung von DirectX. Die statische Bibliothek, die das Laden der DLL erledigt, nennen wir **ZFXRenderer**. Dieser Renderer hat nur zwei Aufgaben. Zum einen soll er die DLL laden, die der Benutzer unserer *ZFXEngine* verwenden möchte, also die DirectX-Version oder die OpenGL-Version oder eben jede andere Variante eines Renderers, die wir in einer DLL implementiert haben. Zum anderen benötigen wir den ZFXRenderer dann auch noch, um beim Beenden des Programms die verwendeten Ressourcen wieder freizugeben.

Der Benutzer unserer Engine muss sich mit dem ZFXRenderer entsprechend nur zweimal auseinander setzen. Bei der Initialisierung des Programms muss er ein Objekt dieser Klasse instanziieren und sich von diesem ein Device-Objekt übergeben lassen. Beim Beenden des Programms wird der Renderer einfach wieder per `delete`-Aufruf über den Destruktor gelöscht. Bei dem Device-Objekt handelt es sich um einen Zeiger auf eine Instanz der Klasse aus der DLL. Diese Klasse nennen wir, analog zur DirectX-Namensgebung, **ZFXRenderDevice**. Dies ist also die Bezeichnung, die unser Interface, also die abstrakte Klasse, erhält. Die tatsächliche Klasse in einer DLL erhält einen anderen Namen, wird jedoch, wie oben besprochen, von der abstrakten Klasse ZFXRenderDevice abgeleitet und implementiert deren rein virtuelle Funktionen.

Anlegen des Projekts in Visual C++

Jetzt machen wir uns an die Arbeit. Als Erstes starten wir *Visual C++* und wählen aus dem Menü DATEI den Befehl NEU ... aus. In dem nun erscheinenden Dialog wählen wir die Registerkarte PROJEKTE aus und klicken in der Auswahlliste der verschiedenen Projekttypen einmal auf *Win-32 Bibliothek (statische)*. Auf der rechten Seite wählen wir noch einen entsprechen-

1 Der folgende Quellcode basiert auf dem hervorragenden Tutorial *Striving for Graphics API Independence* auf GameDev.net von Erik Yuzwa, `www.gamedev.net/reference/articles/article1672.asp`.

den Pfad aus, in dem wir unseren Arbeitsbereich anlegen wollen. Im Feld PROJEKTNAME geben wir die Bezeichnung *ZFXRenderer* ein. Ein Klick auf den Button OK führt zu einem weiteren Dialog-Fenster von *Visual C++*. Hier können wir über diverse Checkboxen weitere Optionen für das Projekt festlegen. Wir achten darauf, dass keine dieser Checkboxen angewählt ist, erstellen den Arbeitsbereich nun endgültig mit einem Klick auf den Button FERTIGSTELLEN und bestätigen noch einmal per Mausklick auf die Schaltfläche OK im neu auftauchenden Dialog.

Einfügen der Dateien

Jetzt haben wir den Arbeitsbereich vor uns. Dieser ist jedoch gähnend leer. Also wählen wir aus dem Menü PROJEKT den Menüpunkt DEM PROJEKT HINZUFÜGEN und dort die Option NEU aus dem Untermenü. Jetzt erscheint der Dialog für das Einfügen von Objekten in den Arbeitsbereich. Dort wählen wir die Registerkarte DATEIEN aus. Aus der Liste suchen wir den Eintrag *C/C++-Header-Datei* beziehungsweise *C++-Quellcodedatei* und geben auf der rechten Seite im Dialog den Namen für die neue Datei an. Auf diese Weise fügen wir dem Projekt die folgenden neuen drei Dateien hinzu:

- `ZFXRenderer.cpp`
- `ZFXRenderer.h`
- `ZFXRenderDevice.h`

Die ersten beiden Dateien sind für die Implementierung der Klasse ZFXRenderer gedacht. Die letztgenannte Datei ist die Definition des Interfaces, wird also eine abstrakte Klasse enthalten, von der sich dann jeweils die Klassen der DLLs ableiten. Die eigentliche Implementierung der Funktionen und Klassen in den Dateien schauen wir uns gleich an. Zunächst erweitern wir unseren Arbeitsbereich noch ein wenig, um auch eine DLL verwenden zu können. Im Anschluss definieren wir noch das Interface, damit wir überhaupt wissen, was wir eigentlich implementieren müssen.

ZFXD3D, eine dynamische Bibliothek als Render-Device

Mehrere Projekte in einem Arbeitsbereich

In unserem eben erzeugten Arbeitsbereich haben wir nun ein Projekt namens ZFXRenderer. Viele werden bisher ihre Arbeitsbereiche auch immer nur so verwendet haben, dass sie ein einziges Projekt darin verwendeten. Ein Arbeitsbereich in *Visual C++* kann aber beliebig viele Projekte enthalten. Insbesondere bei logisch zusammenhängenden Projekten von Bibliotheken (statischen und dynamischen) macht es Sinn, diese in einem Arbeitsbereich zu halten. Das Projekt für die statische Bibliothek ist ja die Basis, über die wir auf unsere DLLs zugreifen werden. Also fügen wir in denselben Arbeitsbereich für jede DLL, die wir zum Rendern erstellen, ein eigenes Projekt ein. Im Verlauf dieses Buches arbeiten wir nur mit einer DLL, und zwar auf Basis der Direct3D-Komponente von DirectX. Für jede eigene Kapselung einer anderen API (OpenGL, Software-Renderer usw.) erzeugt man einfach analog ein eigenes DLL-Projekt in diesem Arbeitsbereich.

Wir gehen also wieder genauso vor wie eben bei dem Einfügen von Dateien in das Projekt ZFXRenderer. Diesmal wählen wir jedoch nicht die Registerkarte DATEIEN, sondern die Registerkarte PROJEKTE. Hier wählen wir den Eintrag *Win32 Dynamic-Link Library*. Bei der Pfadangabe auf der rechten Seite hängen wir noch ein Unterverzeichnis namens *ZFXD3D* an den Pfad des bisherigen Projekts an. Auch als Namen des Projekts geben wir *ZFXD3D* an. Ein Klick auf den OK-Button zeigt einen neuen Dialog an, in dem wir einfach den Eintrag *Ein leeres DLL-Projekt* wählen und wieder durch OK bestätigen. Den daraufhin auftauchenden Informationsdialog klicken wir mit OK schnell weg. Nun haben wir ein zweites Projekt, diesmal eine DLL, in unserem Arbeitsbereich.

Aktives Projekt im Arbeitsbereich festlegen

Wenn man in einem Arbeitsbereich von Visual C++ *mehrere verschiedene Projekte angelegt hat, dann muss man immer darauf achten, welches davon gerade aktiv ist. In der Dateiübersicht von* Visual C++ *erkennt man das daran, dass der Name des aktiven Projekts in einer fett formatierten Schrift dargestellt wird. Alle projektbezogenen Aktionen von* Visual C++ *beziehen sich auf dieses eine aktive Projekt, insbesondere das Einfügen von Dateien und auch das Kompilieren.*

Welches Projekt gerade aktiv ist, kann man über das Menü PROJEKT *festlegen. Dort gibt es den Menüpunkt* AKTIVES PROJEKT FESTLEGEN, *wo man alle Projekte des Arbeitsbereichs in einer Auswahlliste zur Verfügung hat. Sobald man Änderungen an einem Projekt vorgenommen hat, sollte man darauf achten, dass man auch das entsprechende Projekt als aktives Projekt festgelegt hat, bevor man die Dateien kompiliert.*

Da wir es ja nicht anders wollten, enthält das DLL-Projekt noch keine Dateien. Aber das ändern wir nun sofort. Analog zu den Arbeitsschritten beim Einfügen von Dateien in das Projekt der statischen Bibliothek fügen wir nun die folgenden Dateien in das Projekt der DLL ein:

- `ZFXD3D_init.cpp`
- `ZFXD3D_enum.cpp`
- `ZFXD3D_main.cpp`
- `ZFXD3D.h`
- `ZFXD3D.def`

Die ersten vier Dateien dienen zur Implementierung der Klasse ZFXD3D. Dabei wird es sich um eine von dem Interface abgeleitete Klasse handeln, in der wir Direct3D als Grafik-API verwenden. Zur besseren Übersicht werden wir jedoch die Funktionen der Klasse nicht in eine einzige Datei quetschen, sondern ein wenig trennen. Die Funktionen für die Initialisierung und das

Beenden von Direct3D schreiben wir in die Datei mit dem Suffix `_init`. Die Enumeration vorhandener Grafikhardware ist aber, wie man es von DirectX gewohnt ist, auch recht komplex, daher kapseln wir diese Funktionen in einer eigenen Klasse in der Datei mit dem Suffix `_enum`. Die eigentliche Funktionalität steckt dann in der Datei mit dem Suffix `_main`. Eventuell kommen bei Bedarf später noch andere Dateien hinzu, aber jetzt kommen wir erst mal damit aus. Die letztgenannte Datei, `ZFXD3D.def`, gibt an, welche Funktionen die DLL exportiert und für externe Benutzer zugänglich macht.

Nun haben wir unseren Arbeitsbereich so eingerichtet, dass wir mit der Arbeit beginnen können. Um uns noch einmal zu vergewissern, was wir hier tun, sollten wir erneut einen Blick auf den Arbeitsbereich werfen. Wir haben dort eine statische Bibliothek, die die Auswahl und Verwendung einer DLL für ein anderes Projekt ermöglichen wird. Dann haben wir im Arbeitsbereich für jede Grafik-API, die wir unterstützen wollen (bisher nur Direct3D), ein eigenes DLL-Projekt.

:-)
TIPP

Per Voreinstellung erzeugt Visual C++ die kompilierten Dateien, in diesem Fall die `*.lib`- und `*.dll`-Dateien, in den Ordnern Debug bzw. Release – je nach gewählter Kompilierungsart. Im Menü PROJEKT unter dem Menüpunkt EINSTELLUNGEN ... kann man auf der Registerkarte ALLGEMEIN diese Verzeichnisse bzw. den ganzen Pfad ändern. So kann man beispielsweise einen Ordner auf der Festplatte erstellen, in den man alle Bibliotheken einer Engine hineinkompiliert.

Fügt man diesen Verzeichnispfad in Visual C++ für Bibliotheken-Verzeichnisse hinzu, so kann man andere Projekte zu seinen Bibliotheken linken, ohne nach dem Ändern der Bibliotheken an den anderen Projekten etwas ändern zu müssen, was nötig wäre, wenn wir die kompilierten Bibliotheken einfach nur in das Verzeichnis dieser Projekte kopieren würden.

Bevor wir uns daran machen, unsere Bibliotheken zu implementieren, müssen wir uns noch ein paar Gedanken über das Interface machen.

ZFXRenderDevice, ein Interface als abstrakte Klasse

Sämtliche Funktionen, die wiederum Methoden einer spezifischen API verwenden, müssen wir für die *ZFXEngine* in einer DLL kapseln. Nur so erreichen wir wirkliche Unabhängigkeit von einer bestimmten API oder API-Version. Dazu ist es aber nötig, dass wir uns zunächst überlegen, welche Funktionen wir überhaupt brauchen. Aus diesen Überlegungen heraus konstruieren wir dann ein Interface, also eine abstrakte C++-Klasse, an die sich unsere DLLs sozusagen *halten* müssen.

Für dieses Kapitel werden wir das Interface sehr übersichtlich halten. Hier geht es schließlich um die Technik, wie wir unsere *ZFXEngine* unabhängig von einer Grafik-API halten können, indem wir die entsprechenden Funk-

tionen in einer beliebig austauschbaren DLL kapseln. Am Ende dieses Kapitels wird unsere DLL nicht viel mehr können, als den Bildschirm in einer beliebigen Farbe löschen zu können – das dafür aber sehr komfortabel. Intern wird in der DLL aber wesentlich mehr stecken. Insbesondere soll die DLL dazu in der Lage sein, die vorhandene Grafikhardware zu erkennen und ihre Eigenschaften abzufragen. Per Auswahldialog kann der Benutzer dann einen verfügbaren Modus (Bildschirmauflösung, Farbtiefe usw.) wählen und die Engine starten.

Schauen wir uns erst einmal die Definition des Interfaces an:


```
// in der Datei: ZFXRenderDevice.h
#define MAX_3DHWND 8

class ZFXRenderDevice { class ZFXRenderDevice {
   protected:
      HWND       m_hWndMain;         // Hauptfenster
      HWND       m_hWnd[MAX_3DHWND]; // 3D-Fenster
      UINT       m_nNumhWnd;         // Anzahl Fenster
      UINT       m_nActivehWnd;      // aktives Fenster
      HINSTANCE  m_hDLL;             // DLL-Modul
      DWORD      m_dwWidth;          // Screen-Breite
      DWORD      m_dwHeight;         // ScreenHöhe
      bool       m_bWindowed;        // Windowed Mode?
      char       m_chAdapter[256];   // Name der GaKa
      FILE      *m_pLog;             // Logfile
      bool       m_bRunning;

   public:
      ZFXRenderDevice(void) {};
      virtual ~ZFXRenderDevice(void) {};

      // INIT/RELEASE STUFF:
      // ===================
      virtual HRESULT Init(HWND, const HWND*, int,
                           int, int, bool)=0;
      virtual void Release(void)      =0;
      virtual bool IsRunning(void)    =0;


      // RENDERING STUFF:
      // ================
      virtual HRESULT UseWindow(UINT nHwnd)=0;
      virtual HRESULT BeginRendering(bool bClearPixel,
                                     bool bClearDepth,
                                     bool bClearStencil)
                                     =0;
      virtual void EndRendering(void)=0;
      virtual HRESULT Clear(bool bClearPixel,
                            bool bClearDepth,
```

```
                                bool bClearStencil) =0;

    virtual void SetClearColor(float fRed,
                               float fGreen,
                               float fBlue)=0;
}; // class

typedef struct ZFXRenderDevice *LPZFXRENDERDEVICE;
```

Virtuelle Destruktoren

An Attributen definiert das Interface schon einen ganzen Satz wichtiger Variablen, insbesondere die Höhe, Breite und Farbtiefe der Auflösung. Diese Attribute stehen den abgeleiteten Klassen natürlich auch zur Verfügung. Interessanter sind hier aber die rein virtuellen Funktionen. Als Erstes beachte man, dass auch der Destruktor der Klasse als virtuell definiert ist. Auf diese Weise verhindern wir, dass wir das Aufrufen des Destruktors verpassen. Normalerweise entscheidet ja der Typ des Pointers auf ein Objekt, von welcher Klasse der Destruktor verwendet wird. Später arbeiten wir aber mit Pointern vom Typ der Klasse ZFXRenderDevice. Diese zeigen allerdings auf Objekte vom Typ der abgeleiteten Klasse ZFXD3D. Definieren wir nun auch den Destruktor virtuell, so entscheidet der Typ des Objekts, von welcher Klasse wir den Destruktor verwenden.

Dann folgen ein paar Funktionen für die Initialisierung, die Freigabe und die Prüfung auf erfolgreiche Initialisierung der Klasse. Auch für das Rendern durch unsere DLL haben wir bisher nur vier Funktionen. Die Funktion `ZFXRenderDevice::BeginnRendering` dient dazu, den Render-Vorgang in einem Frame zu starten. Über die Parameterliste der Funktion wird gesteuert, ob und welche Buffer (Pixel-Buffer, Depth-Buffer, Stencil-Buffer) dabei gelöscht werden sollen. Dazu haben wir noch eine weitere Funktion, mit der lediglich der oder die angegebenen Buffer gelöscht werden, ohne jedoch die Szene zu starten. Dies kann hilfreich sein, wenn man mitten in einer Szene beispielsweise den Depth-Buffer löschen muss. An grafischen Funktionen ist im Interface bisher nur eine Methode vorgesehen, mit der man die Hintergrundfarbe ändern kann, mit der der Pixel-Buffer gelöscht wird. In den späteren Kapiteln werden wir die Funktionalität des Interfaces natürlich noch um einiges erweitern. Für dieses Kapitel reichen diese Funktionen aber aus.

Zu guter Letzt definieren wir noch die Struktur `LPZFXRENDERDEVICE`, die nichts weiter als ein Pointer auf eine Instanz der Klasse ist.

3.4 Implementierung der statischen Bibliothek

Der Ankerpunkt für die Arbeit mit unseren DLLs zum Rendern ist die statische Bibliothek ZFXRenderer. Ihre Aufgabe ist es zu entscheiden, welche DLL geladen wird. Eine DLL repräsentiert dabei ein Render-Device, über das dann tatsächlich Grafik ausgegeben werden kann. Die Implementierung

der statischen Bibliothek ist recht kurz und bündig und wird im Verlauf des Buches nicht mehr wirklich geändert. Neu kompilieren muss man sie nur in zwei Fällen: Zum einen natürlich dann, wenn man an ihren Funktionen doch noch etwas ändert. Das ist immer dann der Fall, wenn man beispielsweise eine komplett neue DLL unterstützen möchte. Dann muss man ja den Namen der DLL angeben und die Bedingung, unter der diese spezifische DLL geladen werden soll. Zum anderen müssen wir die statische Bibliothek neu kompilieren, wenn wir an unserem Interface ZFXRenderDevice etwas ändern, da die Bibliothek den Header `ZFXRenderDevice.h` auch mit einbindet.

Wo wir gerade bei Headern sind, schauen wir uns den Header gleich einmal an, in dem die Definition der Klasse `ZFXRenderer` steht.


```
// in der Datei: ZFXRenderer.h
#include "ZFXRenderDevice.h"

class ZFXRenderer {
   public:
      ZFXRenderer(HINSTANCE hInst);
      ~ZFXRenderer(void);

      HRESULT            CreateDevice(char *chAPI);
      void               Release(void);
      LPZFXRENDERDEVICE  GetDevice(void)
                             { return m_pDevice; }
      HINSTANCE          GetModule(void)
                             { return m_hDLL;    }

   private:
      ZFXRenderDevice  *m_pDevice;
      HINSTANCE         m_hInst;
      HMODULE           m_hDLL;
   }; // class

typedef struct ZFXRenderer *LPZFXRENDERER;
```

Wie versprochen, ist die Klasse nicht sehr umfangreich. Dem Konstruktor der Klasse geben wir den Instanzzähler des Windows-Programms an, das die Klasse später verwendet. Diesen Zähler speichern wir in einem Attribut der Klasse, falls wir ihn später einmal brauchen sollten. Viel wichtiger ist das Attribut `m_hDLL`, das später auf den Instanzzähler der geladenen DLL hinweisen wird. Das Attribut `m_pDevice` ist dann endlich der ersehnte Pointer auf die von dem Interface abgeleitete Klasse `ZFXD3D` aus der DLL. Das zugehörige Objekt erstellen wir mit der Methode `ZFXRenderer::CreateDevice`.

Aber schauen wir uns erst mal den Konstruktor und den Destruktor an. In diesen werden lediglich die Attribute initialisiert bzw. die `Release()`-Methode beim Löschen des Objekts aufgerufen.

```
ZFXRenderer::ZFXRenderer(HINSTANCE hInst) {
   m_hInst   = hInst;
   m_hDLL    = NULL;
   m_pDevice = NULL;
   }

ZFXRenderer::~ZFXRenderer(void) {
   Release();
   }
```

Gehen wir also gleich zu den interessanten Dingen dieser Manager-Klasse. Und das ist definitiv die Erstellung eines Device-Objekts in der entsprechenden Funktion. Dafür benötigen wir aber noch schnell die folgende Definition in der Header-Datei des Interfaces:

```
// in der Datei: ZFXRenderDevice.h
extern "C" {
  HRESULT CreateRenderDevice(HINSTANCE hDLL,
                        ZFXRenderDevice **pInterface);
  typedef HRESULT (*CREATERENDERDEVICE)(HINSTANCE hDLL,
                        ZFXRenderDevice **pInterface);

  HRESULT ReleaseRenderDevice(ZFXRenderDevice
                              **pInterface);
  typedef HRESULT(*RELEASERENDERDEVICE)(ZFXRenderDevice
                **pInterface);
   }
```

Export von Klassen aus einer DLL

Auf diese Weise definieren wir die Symbole `CREATERENDERDEVICE` und `RELEASERENDERDEVICE` für die entsprechenden Funktionen `CreateRenderDevice()` und `ReleaseRenderDevice()` im C-Stil, die die DLL später bereitstellen muss. Und das sind nämlich genau die Funktionen, die als einzige von unseren DLLs exportiert werden. Weiter oben hatten wir ja schon besprochen, dass wir aus einer DLL keine Klassen exportieren können. Daher verwenden wir diese Funktionen, um die DLL einen Pointer vom Typ des Interfaces ZFXRenderDevice auf ein Objekt ihrer eigenen Klasse, beispielsweise ZFXD3D, setzen zu lassen. Diese Klasse ist ja von der Klasse ZFXRenderDevice abgeleitet. So erhalten wir dann doch Zugriff auf die Klasse aus der DLL. Entsprechend muss jede DLL, die hier als Render-Device verwendet werden soll, diese beiden Funktionen definieren und exportieren. Aber so weit sind wir ja noch nicht.

Nun schauen wir uns an, wie die Klasse ZFXRenderer ein Render-Device erzeugen kann. Die entsprechende Funktion nimmt als Parameter einen String auf, der den Namen einer API angeben soll. Im Verlauf dieses Buches entwickeln wir ja nur eine DLL für Direct3D, also akzeptiert diese Funktion bisher auch nur den String »`Direct3D`«. Enthält der Parameter etwas anderes, bricht die Funktion mit einer Fehlermeldung ab, weil sie ja weiß, dass sie nur die DLL für Direct3D zur Auswahl hat.


```
HRESULT ZFXRenderer::CreateDevice(char *chAPI) {
   char buffer[300];

   if (strcmp(chAPI, "Direct3D") == 0) {
      m_hDLL = LoadLibraryEx("ZFXD3D.dll",NULL,0);
      if(!m_hDLL) {
         MessageBox(NULL,
            "Loading ZFXD3D.dll from lib failed.",
            "ZFXEngine - error", MB_OK | MB_ICONERROR);
         return E_FAIL;
         }
      }
   else {
      sprintf(buffer, "API '%s' not supported.", chAPI);
      MessageBox(NULL, buffer, "ZFXEngine - error",
                 MB_OK | MB_ICONERROR);
      return E_FAIL;
      }

   CREATERENDERDEVICE _CreateRenderDevice = 0;
   HRESULT hr;

   // Zeiger auf die dll Funktion 'CreateRenderDevice'
   _CreateRenderDevice = (CREATERENDERDEVICE)
                         GetProcAddress(m_hDLL,
                                "CreateRenderDevice");
   // aufurf der dll Create-Funktion
   hr = _CreateRenderDevice(m_hDLL, &m_pDevice);
   if(FAILED(hr)){
      MessageBox(NULL,
         "CreateRenderDevice() from lib failed.",
         "ZFXEngine - error", MB_OK | MB_ICONERROR);
      m_pDevice = NULL;
      return E_FAIL;
      }

   return S_OK;
   } // CreateDevice
```

Wenn die Funktion als Parameter einen String übergeben bekommt, mit dem sie etwas anfangen kann, dann macht sie sich an die Arbeit. Im ersten Schritt versucht die Funktion, die entsprechende DLL zu laden. Im dem Fall, in dem Direct3D verwendet werden soll, ist das die DLL `ZFXD3D.dll`, für die wir ja bereits ein Projekt angelegt haben. Dazu stellt die WinAPI die folgende Funktion zur Verfügung:

```
HINSTANCE LoadLibraryEx(LPCTSTR lpLibFileName,
                        HANDLE  hFile,
                        DWORD   dwFlags);
```

Für den ersten Parameter dieser Funktion müssen wir den Namen der zu ladenden DLL als String angeben. Der zweite Parameter dieser Funktion ist für interne Zwecke reserviert und muss `NULL` sein. Im dritten Parameter können diverse Flags angegeben werden, das ist aber in unserem Fall unnötig. Daher setzen wir diesen Wert auf 0. Der Rückgabewert dieser Funktion hingegen ist sehr wichtig. Er ist das Handle von Windows auf die geladene DLL. Dieses brauchen wir beispielsweise dann, wenn wir in der DLL mit weiteren Funktionen der WinAPI arbeiten wollen, die als Parameter den `HINSTANCE`-Wert verlangen.

Funktionen in der DLL finden

War das Laden der DLL erfolgreich, dann möchten wir nun gern einen Pointer auf ein Objekt von der Klasse in der DLL. Dazu soll die DLL ja die Funktion `CreateRenderDevice()` exportieren. Anders als bei einer statischen Bibliothek können wir diese Funktion nicht direkt aufrufen, weil die dynamische Bibliothek eben nicht in das Projekt mit hineinkompiliert wird. Der Compiler weiß also zur Kompilierungszeit gar nicht, an welcher Stelle sich diese Funktion später, nach dem Laden der DLL zur Laufzeit, befinden wird. Glücklicherweise gibt es aber eine Funktion der WinAPI, mit der wir zur Laufzeit die Adresse einer Funktion auffinden können:

```
FARPROC GetProcAddress(HMODULE hModule,
                       LPCTSTR lpProcName);
```

Als ersten Parameter müssen wir dieser Funktion das Handle auf die geladene DLL angeben, also genau den Wert, den die Funktion `LoadLibraryEx()` zurückgeliefert hat. Im zweiten Parameter geben wir dann den Namen der Funktion an, die wir in der DLL suchen. Der Rückgabewert der Funktion ist dann die Adresse der Funktion, sofern sie gefunden wurde. Über diese Adresse, die wir im Pointer mit der Bezeichnung `_CreateRenderDevice` speichern, können wir nun die Funktion in der DLL aufrufen. Damit haben wir, sofern die Funktion keinen Fehler liefert, einen gültigen Pointer auf ein Objekt der Klasse aus der DLL. Keine Panik, diese DLL sehen wir uns auch gleich noch im Detail an, um zu sehen, was die exportierten Funktionen, die wir hier verwenden, überhaupt machen.

Ganz genauso arbeitet auch die Funktion für die Freigabe des RenderDevice-Objekts. Dafür verwenden wir nur die zweite Funktion, die die DLL exportiert.

{} LIST

```
void ZFXRenderer::Release(void) {
   RELEASERENDERDEVICE _ReleaseRenderDevice = 0;
   HRESULT hr;

   if (m_hDLL) {
      // Zeiger auf dll-Funktion 'ReleaseRenderDevice'
      _ReleaseRenderDevice = (RELEASERENDERDEVICE)
                              GetProcAddress(m_hDLL,
```

```
                                "ReleaseRenderDevice");
    }
// call dll's release function
if (m_pDevice) {
    hr = _ReleaseRenderDevice(&m_pDevice);
    if(FAILED(hr)){
       m_pDevice = NULL;
       }
    }
} // release
```

Soll das Objekt aus der DLL, also das Render-Device, freigegeben werden, dann suchen wir zuerst die Adresse der entsprechenden Funktion, die die DLL für eben diesen Zweck für uns exportiert, also die Funktion `ReleaseRenderDevice()`. Haben wir diese Adresse gefunden, so speichern wir sie in einem passenden Pointer mit der Bezeichnung `_ReleaseRenderDevice` und rufen sie mit dem freizugebenden Objekt auf. Dieses ist ja als Attribut der Klasse ZFXRenderer gespeichert.

Am Anfang erscheint das ein wenig verwirrend, insbesondere die beiden ominösen Typdefinitionen im Header `CREATERENDERDEVICE` und `RELEASERENDERDEVICE`. Diese sind dazu da, dass der Compiler weiß, welche Parameterliste sich an der gefundenen Adresse zu den gesuchten Funktion in der DLL befindet. Nur so können wir einen korrekten Pointer auf diese Adresse setzen, um die Funktion richtig aufrufen zu können.

Die gute Nachricht ist, dass das schon alles war, was wir in der statischen Bibliothek implementieren müssen. Die einzige Änderung, die man hier später vornehmen muss, ist die Auswahl der zu ladenden DLL, wenn man mehrere zur Auswahl hat.

3.5 Implementierung der dynamischen Bibliothek

Als dynamische Bibliothek haben wir ja bereits das Projekt ZFXD3D erzeugt. In dieser dynamischen Bibliothek wollen wir eine Kapselung für Direct3D schreiben. In diesem Kapitel fällt unsere Implementierung noch ein wenig rudimentär aus, aber im Verlauf dieses Buches werden wir die Implementierung ständig um weitere Funktionen bereichern. Betrachten wir nun zuerst einmal die Definition der Klasse. Ein besonderes Augenmerk richten wir hier auf das Attribut `m_pChain[MAX_3DHWND]`, wobei es sich um ein Array von Swap-Chain-Elementen handelt. Selbst wer schon öfter mit Direct3D gearbeitet hat, dem sind die so genannten Swap Chains von DirectX eventuell immer noch neu. Diese brauchen wir, um mit Direct3D komfortabel in beliebig viele Child-Windows rendern zu können. Aber das werden wir später noch sehen. Der Wert von `MAX_3DHWND` ist übrigens als Definition in der Interface-Datei zu finden.

LIST

```
// in der Datei: ZFXD3D.h
class ZFXD3D : public ZFXRenderDevice {
   public:
      ZFXD3D(HINSTANCE hDLL);
      ~ZFXD3D(void);

      // Initialisierungsfunktionen
      HRESULT Init(HWND, const HWND*, int, int, int,
                   bool);
      BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

      // Interface-Funktionen
      void    Release(void);
      bool    IsRunning(void) { return m_bRunning; }
      HRESULT BeginRendering(bool,bool,bool);
      HRESULT Clear(bool,bool,bool);
      void    EndRendering(void);
      void    SetClearColor(float, float, float);
      HRESULT UseWindow(UINT nHwnd);

   private:
      ZFXD3DEnum            *m_pEnum;
      LPDIRECT3D9            m_pD3D;
      LPDIRECT3DDEVICE9      m_pDevice;
      LPDIRECT3DSWAPCHAIN9   m_pChain[MAX_3DHWND];
      D3DPRESENT_PARAMETERS  m_d3dpp;
      D3DCOLOR               m_ClearColor;
      bool                   m_bIsSceneRunning;
      bool                   m_bStencil;

      // startet die API
      HRESULT Go(void);

      // Protokollieren des Ablaufs
      void Log(char *, ...);
   }; // class
```

In dieser Definition finden wir neben dem Konstruktor und Destruktor zunächst acht öffentliche Member-Funktionen. Sieben davon sind Funktionen, die durch das Interface zwingend vorgeschrieben sind. Hinzugekommen ist hier lediglich die Funktion `ZFXD3D::DlgProc`. Da wir nur mit Pointern vom Typ des Interfaces arbeiten, ist diese Funktion von außen nicht sichtbar. Wir benötigen sie aber innerhalb der Klassen-Implementierung von ZFXD3D als öffentlich zugängliche Funktion. Das sehen wir aber gleich, sobald wir zu dem Dialog dieser Klasse kommen.

Member-Funktionen in abgeleiteten Klassen, die die Implementierung von rein virtuellen Funktionen einer abstrakten Klasse darstellen, werden von C++ automatisch als virtuelle Funktionen behandelt. Die explizite Angabe des Schlüsselwortes `virtual` *ist hier nicht nötig.*

Enumeration aus dem DirectX SDK

An geschützten Membern verfügt die Klasse über eine Reihe von Attributen, die unter anderem die wichtigsten Direct3D-Objekte und Informationen über die Grafikkarte speichern. Dazu benötigen wir einen Satz an Funktionen, die die Informationen der Grafikkarte abfragen (Enumeration) und auflisten. Im Folgenden werde ich voraussetzen, dass die Enumeration der vorhandenen Grafikhardware unter DirectX bekannt ist.[2] Als Attribut verwende ich in dieser Klasse unter anderem `m_pEnum` als Objekt der Klasse `ZFXD3DEnum`. Diese Klasse führt die Enumeration der verfügbaren Grafikkarten und deren Modi durch. Das Abdrucken dieser Klasse spare ich mir hier, da diese Enumeration eine vereinfachte Variante der Common Files aus dem DirectX SDK ist, die ich als bekannt voraussetze. Die Implementierung dieser Klasse findet ihr natürlich auf der CD-ROM zu diesem Buch. Wer auf diesem Gebiet noch ein wenig unsicher ist, der sollte sich diese noch einmal zu Gemüte führen.

Ablauf des Hochfahrens von Direct3D

Besprechen wir erst einmal den groben Plan von dem, was wir vorhaben. Unser Interface besteht zunächst nur darauf, dass wir eine `Init()`-Funktion implementieren, durch die die DLL die verwendete Grafik-API hochfährt. Es ist aber durchaus empfehlenswert, dieses Hochfahren flexibel zu gestalten. In der Implementierung der ZFXD3D-DLL werde ich es so machen, dass die DLL bei Aufruf der Initialisierungsfunktion eine Dialogbox anzeigt. Diese Dialogbox listet alle gefundenen Grafikkarten auf dem Zielcomputer auf, ebenso wie sämtliche Bildschirmauflösungen, die die entsprechende Grafikkarte fahren kann. Ebenso kann der Benutzer dann auswählen, ob er im Fullscreen-Modus oder im Windowed-Modus starten möchte. So erhält der Benutzer unserer Engine in einem einzigen Funktionsaufruf von außen die Möglichkeit, flexibel eine Grafikkarte auszuwählen (falls mehrere in einem Computer vorhanden sind) und den Bildschirmmodus anzugeben oder die Anwendung im Fenster zu starten. Bequemer geht es gar nicht mehr.

Aber bevor wir dazu kommen, müssen wir uns erst noch ansehen, wie wir über die oben besprochenen exportierten Funktionen im C-Stil auf die Klasse in der DLL zugreifen können.

Exportierte Funktionen

Bei der Erstellung des Projekts ZFXD3D hatten wir dem Projekt ja auch gleich ein paar leere Dateien hinzugefügt – unter anderem die Datei `ZFXD3D.def`, in der die Exportinformationen für die DLL stehen werden. Das klingt viel aufwändiger, als es ist, denn die Datei erhält als Inhalt nur die folgenden Zeilen:

2 Siehe auch das Tutorial unter `www.zfx.info`.

```
;ZFXD3D.def
;Die folgenden Funktionen werden von der DLL exportiert
LIBRARY   "ZFXD3D.dll"
EXPORTS
   CreateRenderDevice
   ReleaseRenderDevice
```

Zum einen wird der Name der DLL angegeben, damit wir sie beim Laden auch eindeutig identifizieren können. Zum anderen sind dort die beiden Funktionen im C-Stil aufgelistet, über die wir ja bereits in der statischen Bibliothek ZFXRenderer auf die DLL zugegriffen haben. Wichtig sind hierbei nur die Namen der Funktionen, nicht aber die Parameterlisten. Die Implementierung dieser Funktionen schreiben wir dann in eine `*.cpp`-Datei, die zu dem DLL-Projekt gehört.

!! STOP

Wenn man das Visual Studio .NET verwendet, dann funktioniert die Verwendung eines `*.def` *Files nicht immer korrekt. Manchmal verweigert das Programm dann einfach den Dienst, weil die exportierten Funktionen in der DLL nicht gefunden werden – die Funktion* `GetProcAddress()` *liefert lediglich* `NULL` *zurück. Das kann man aber leicht beheben, indem man auf das* `*.def` *File verzichtet und die Funktionen anders exportiert.*

Aus der Interface-Header-Datei entfernt man die Deklarationen der exportierten Funktionen, hier wären das also `CreateRenderDevice()` *und* `ReleaseRenderDevice()` *in der Datei* `ZFXRenderDevice.h`*. Die beiden* `typedef` *Anweisungen bleiben dort aber stehen. Die Deklarationen schreibt man statt dessen in eine Header-Datei in der DLL, und fügt sowohl den Deklarationen als auch den Funktionsköpfen bei der Funktions-Definition das folgende Präfix hinzu:*

```
extern "C" __declspec(dllexport)
```

Auf diese Weise sind die Funktionen in der DLL auch als Exporte gekennzeichnet, und werden nun auch in einem Build von Visual Studio .NET korrekt erstellt.

Wenn wir uns den Quellcode dieser Funktionen einmal ansehen, so sind sie beinahe lächerlich kurz. In diesen Funktionen muss ja auch nichts anderes geschehen als das Erzeugen bzw. das Freigeben eines Objekts der Klasse. Das sieht dann so aus:

{} LIST

```
// in der Datei: ZFXD3D_init.cpp
#include "ZFX.h"

HRESULT CreateRenderDevice(HINSTANCE hDLL,
                           ZFXRenderDevice **pDevice) {
   if(!*pDevice) {
      *pDevice = new ZFXD3D(hDLL);
```

```
      return ZFX_OK;
      }
   return ZFX_FAIL;
   }


HRESULT ReleaseRenderDevice(ZFXRenderDevice **pDevice) {
   if(!*pDevice) {
      return ZFX_FAIL;
      }
   delete *pDevice;
   *pDevice = NULL;
   return ZFX_OK;
   }
```

Bei der Erstellung eines Render-Device-Objekts unserer Klasse müssen wir das Handle auf die geladene DLL angeben, da wir diese im Konstruktor noch brauchen. Dann erstellen wir einfach ein Objekt unserer Klasse und setzen den Pointer vom Typ des Interfaces auf dieses Objekt. Damit können wir über den Interface-Pointer auf das Objekt zugreifen. Bei der Freigabe löschen wir dann dieses Objekt einfach wieder, wodurch natürlich der Destruktor der Klasse ZFXD3D aufgerufen wird.

Eigene Fehlerwerte definieren

In der Datei `ZFX.h` habe ich diverse Werte definiert. Insbesondere handelt es sich dabei um Rückgabewerte vom Typ `HRESULT` für unsere Funktionen. Später, wenn wir nicht mehr nur mit `ZFX_FAIL` oder `ZFX_OK` arbeiten, werden wir differenziertere Fehler ausgeben, anhand derer der Benutzer sehen kann, warum eine Funktion beispielsweise fehlgeschlagen ist. Diese Fehlerwerte liste ich hier auch nicht alle auf. Aufgrund ihres Präfixes sind sie eindeutig zu erkennen, und die Definitionen stehen in der angesprochenen Header-Datei, die wir in allen unseren Projekten verwenden werden.

Jetzt haben wir es tatsächlich geschafft, unsere Klasse in der DLL von außen verfügbar zu machen. Nun können wir endlich daran gehen, die Funktionalität unseres Render-Device-Objekts zu programmieren.

Komfort durch einen Dialog

Einen Dialog erzeugen

Bevor wir die eigentliche Enumeration der vorhandenen Grafik-Hardware implementieren, benötigen wir auch eine Art Container, der die später gesammelten Informationen darstellen und zur Auswahl stellen kann. Hierfür verwenden wir eine Dialogbox. Wir aktivieren also das ZFXD3D-Projekt als aktives Projekt und fügen ihm eine Dialogbox hinzu. Dazu gehen wir in das Menü EINFÜGEN und wählen den Menüpunkt RESSOURCE ... aus. In der nun erscheinenden Dialogbox wählen wir aus der Liste den Eintrag *Dialog* und klicken auf den Button NEU. Jetzt haben wir ein neues Ressourcen-Objekt mit *Visual C++* erzeugt, und das Programm wechselt automatisch zu dem integrierten Editor für Dialogboxen.

Hier können wir durch einen Doppelklick mit der Maus auf ein Steuerelement oder den Dialog selbst ein Eigenschaften-Menü aufrufen. Unter dem Reiter ALLGEMEIN findet sich hier ein Feld mit der Beschriftung *ID*. Dies ist besonders wichtig für uns, weil wir hier eine eindeutige Bezeichnung für ein Steuerelement des Dialoges oder den Dialog selbst vergeben können. Dem Dialog geben wir auf diese Weise die Bezeichnung »dlgChangeDevice«, wobei es hier besonders wichtig ist, dass wir die ID in dem Feld auch mit den Anführungszeichen eingeben. Über diese ID werden wir den Dialog später aufrufen können. Die folgende Übersicht zeigt, welche Steuerelemente unser Dialog neben ein paar statischen Textfeldern noch enthält (die IDs werden hier aber ganz normal ohne Anführungszeichen eingegeben):

- Combobox mit der Bezeichnung `IDC_ADAPTER`
- Combobox mit der Bezeichnung `IDC_MODE`
- Combobox mit der Bezeichnung `IDC_ADAPTERFMT`
- Combobox mit der Bezeichnung `IDC_BACKFMT`
- Combobox mit der Bezeichnung `IDC_DEVICE`
- Radiobutton mit der Bezeichnung `IDC_FULL`
- Radiobutton mit der Bezeichnung `IDC_WND`
- Buttons mit der Bezeichnung `IDOK` und `IDCANCEL`

Die Comboboxen dienen dazu, die verfügbaren Grafikkarten und deren Betriebsmodi zur Auswahl anzubieten. Im DirectX-Jargon bezeichnet man eine solche Grafikkarte als *Adapter*. Jeder Adapter verfügt über eine gewisse Anzahl Modi, auch Bildschirmauflösung genannt, die er darstellen kann, beispielsweise 800x600, 1024x768 usw. Dann gibt es zwei weitere Comboboxen für die Farbformate, denn seit DirectX 9 ist es möglich, den Back-Buffer mit einer anderen Auflösung zu betreiben als den Front-Buffer, wenn das Programm im Fenstermodus betrieben wird. Die letzte Combobox dient dazu, zwischen den verschiedenen Direct3D-Device-Typen auszuwählen. Hier kommen eigentlich nur zwei Typen in Betracht: einmal das HAL-Device, also die Grafikkarte selbst, und zum anderen das REF-Device, also der *Software Reference Rasterizer*. Wir erinnern uns daran, dass DirectX beinahe sämtliche Funktionalität in Software emulieren kann, wenn diese durch die Hardware nicht unterstützt wird. Das ist zwar quälend langsam, kann zu Testzwecken aber sehr sinnvoll sein.

Die beiden Radiobuttons dienen zur Auswahl, ob die Anwendung im Fullscreen-Modus oder im Fenster gestartet werden soll. Abbildung 3.1 zeigt, wie unser fertiger Dialog jetzt aussehen sollte.

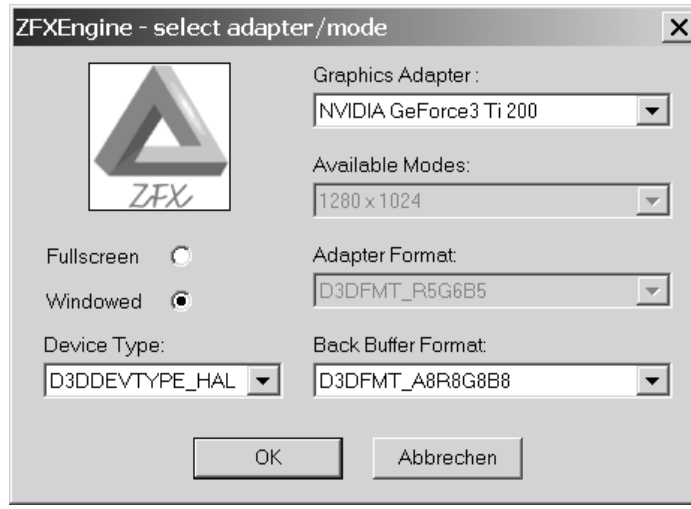

Abbildung 3.1: Der Dialog, wie er später im Programm erscheinen wird

Dialog im Projekt verfügbar machen

Klicken wir nun auf den Button zum Speichern unseres gesamten Projekts in *Visual C++*, dann werden wir aufgefordert, unsere neue Ressource, also die Dialogbox, zu speichern. Dazu wählen wir das Verzeichnis unseres Projekts ZFXD3D aus und speichern die Ressource unter dem Namen `dlgChangeDevice.rc` ab, wobei automatisch die Datei `resource.h` miterzeugt wird. In der *Visual C++*-Projektübersicht haben wir auch schon einen Ordner namens *Ressourcendateien*, der für jedes Projekt automatisch angelegt wird. Auf diesen klicken wir nun mit der rechten Maustaste und wählen aus dem erscheinenden Menü den Eintrag *Dateien zu Ordner hinzufügen ...* aus. Jetzt fügen wir die beiden eben abgespeicherten Dateien in diesen Ordner ein. Damit können wir nun die Dialogbox in unserem Projekt verwenden.

Anzeigen von Dialogen

Ich gehe davon aus, dass die Arbeit mit selbst erstellten Dialogen nichts neues für euch ist, daher behandle ich die entsprechenden Schritte eher knapp. Ein Dialog ist unter Windows nichts anderes als eine besondere Art von Fenster. Daher wird er auch ganz genauso wie ein Fenster behandelt. Wir haben in dem Dialog bereits einige Steuerelemente und deren IDs. Wenn wir nun diese Steuerelemente abfragen wollen, dann machen wir das genauso, wie wir es in jedem anderen Fenster machen würden, nämlich über eine Callback-Funktion, die die Nachrichten verarbeitet, die an den Dialog geschickt werden. Um einen Dialog aufzurufen und ihm eine Callback-Funktion zuzuweisen, verwenden wir die folgende Funktion der WinAPI:

```
int DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate,
              HWND hWndParent, DLGPROC lpDialogFunc);
```

Der erste Parameter der Funktion verlangt nach dem Instanzzähler der Anwendung. Dies ist der Wert, den uns die Funktion `LoadLibraryEx()` für die DLL zurückgeliefert hat. Für den zweiten Parameter geben wir die ID für den anzuzeigenden Dialog an. Wenn wir hier eine ID in Form eines Strings

angeben wollen, müssen wir die entsprechende ID im Ressourcen-Editor auf alle Fälle auch mit Anführungszeichen angegeben haben. Für den dritten Parameter geben wir das Handle auf das Fenster an, zu dem der Dialog gehört. Dies ist in der Regel das Hauptfenster der Anwendung, die unsere DLL verwendet. Als letzten Parameter geben wir den Namen der Callback-Funktion an, die die Nachrichten des Dialogs bearbeitet. Hier haben wir allerdings ein kleines Problem, weil wir so ohne weiteres keine Funktion einer Klasse als Callback-Funktion angeben können. Um das zu umgehen, erzeugen wir eine normale C-Funktion, die dann über ein Objekt der Klasse auf die entsprechende öffentliche Funktion zur Verarbeitung des Dialogs verweist. Und das ist eben jene Funktion `ZFXD3D::DlgProc`. Die Funktion zur Anzeige eines Dialogs hat aber noch einen Rückgabewert vom Typ `int`. Dieser Rückgabewert kann über die Funktion zum Beenden eines Dialogs angegeben werden. Dann kommen wir darauf zurück.

Den Aufruf des Dialogs und die Zuweisung der Callback-Funktion erledigen wir in der Initialisierungsfunktion der Klasse. Hier schauen wir uns erst einmal nur die Callback-Funktion an und besprechen die Funktionen, die sie zu erfüllen hat.

{} LIST

```
ZFXDEVICEINFO  g_xDevice;
D3DDISPLAYMODE g_Dspmd;
D3DFORMAT      g_fmtA;
D3DFORMAT      g_fmtB;

BOOL CALLBACK ZFXD3D::DlgProc(HWND hDlg, UINT message,
                              WPARAM wParam,
                              LPARAM lParam) {
   DIBSECTION dibSection;
   BOOL       bWnd=FALSE;

   // hole die Handles
   HWND hFULL       = GetDlgItem(hDlg, IDC_FULL);
   HWND hWND        = GetDlgItem(hDlg, IDC_WND);
   HWND hADAPTER    = GetDlgItem(hDlg, IDC_ADAPTER);
   HWND hMODE       = GetDlgItem(hDlg, IDC_MODE);
   HWND hADAPTERFMT = GetDlgItem(hDlg, IDC_ADAPTERFMT);
   HWND hBACKFMT    = GetDlgItem(hDlg, IDC_BACKFMT);
   HWND hDEVICE     = GetDlgItem(hDlg, IDC_DEVICE);

   switch (message) {

      // Fenster-Modus vorselektieren
      case WM_INITDIALOG: {
         SendMessage(hWND, BM_SETCHECK, BST_CHECKED, 0);
         m_pEnum->Enum(hADAPTER, hMODE, hDEVICE,
                       hADAPTERFMT, hBACKFMT,
                       hWND, hFULL, m_pLog);
```

```
      return TRUE;
      }

   // Logo rendern (g_hBMP in Init() initialisiert)
   case WM_PAINT: {
      if (g_hBMP) {
         GetObject(g_hBMP, sizeof(DIBSECTION),
                   &dibSection);
         HDC      hdc = GetDC(hDlg);
         HDRAWDIB hdd = DrawDibOpen();
         DrawDibDraw(hdd, hdc, 50, 10, 95, 99,
                     &dibSection.dsBmih,
                     dibSection.dsBm.bmBits, 0, 0,
                     dibSection.dsBmih.biWidth,
                     dibSection.dsBmih.biHeight, 0);
         DrawDibClose(hdd);
         ReleaseDC(hDlg, hdc);
         }
      } break;

   // ein Control hat eine Meldung
   case WM_COMMAND: {
      switch (LOWORD(wParam)) {
         // Okay-Button
         case IDOK: {
            m_bWindowed = !SendMessage(hFULL,
                           BM_GETCHECK, 0, 0);
            m_pEnum->GetSelections(&g_xDevice,
                                   &g_Dspmd,
                                   &g_fmtA,
                                   &g_fmtB);
            GetWindowText(hADAPTER,m_chAdapter,256);
            EndDialog(hDlg, 1);
            return TRUE;
            } break;

          // Cancel-Button
         case IDCANCEL: {
            EndDialog(hDlg, 0);
            return TRUE;
            } break;

          case IDC_ADAPTER: {
            if(HIWORD(wParam)==CBN_SELCHANGE)
              m_pEnum->ChangedAdapter();
            } break;
         case IDC_DEVICE: {
            if(HIWORD(wParam)==CBN_SELCHANGE)
              m_pEnum->ChangedDevice();
            } break;
```

```
            case IDC_ADAPTERFMT: {
                if(HIWORD(wParam)==CBN_SELCHANGE)
                  m_pEnum->ChangedAdapterFmt();
                } break;
            case IDC_FULL: case IDC_WND: {
                m_pEnum->ChangedWindowMode();
                } break;

            } // switch [CMD]
          } break; // case [CMD]
        } // switch [MSG]
    return FALSE;
    }
```

Bei der Initialisierung des Dialogs

Wenn der Dialog initialisiert wird, müssen wir bereits ein paar Aktionen durchführen. Insbesondere rufen wir zuerst die Funktion `ZFXD3DEnum::Enum` auf. Als Parameter erwartet diese Funktion die Handles auf sämtliche Steuerelemente des Dialogs, damit sie die Einträge der Comboboxen füllen kann. Die Klasse `ZFXD3DEnum` speichert diese Handles in Member-Variablen, um immer Zugriff auf die Comboboxen zu haben. Die Funktion fährt dann fort und führt die Enumeration durch. Die Comboboxen werden mit entsprechenden Einträgen betankt.

Im zweiten Abschnitt der Callback-Funktion des Dialogs malen wir das ZFX-Logo in die Dialogbox. Dazu muss die Bibliothek `vfw32.lib` (Video for Windows) gelinkt und der Header `vfw.h` eingebunden werden. Beachtet, dass die Bitmap-Datei bereits in der Funktion `ZFXD3D::Init` geladen wird.

Nachrichten der Steuerelemente des Dialogs

Die Nachricht `WM_COMMAND` wird von Windows an den Dialog geschickt, wenn eines seiner Steuerelemente an Windows gemeldet hat, dass ein Event stattgefunden hat. Wir suchen also in der Callback-Funktion im unteren Wort (`LOWORD`) des `wParam`-Parameters der Nachricht nach der ID des Steuerelementes, das die Nachricht ausgelöst hat. Handelt es sich dabei um eine der Comboboxen, deren gewählter Eintrag sich geändert hat, oder um einen der beiden Radiobuttons, dann rufen wir die entsprechende Funktion der Klasse `ZFXD3DEnum` auf:

- `ZFXD3DEnum::ChangedAdapter`
- `ZFXD3DEnum::ChangedDevice`
- `ZFXD3DEnum::ChangedAdapterFmt`
- `ZFXD3DEnum::ChangedWindowMode`

Diese Funktionen dienen dazu, die Listen der Enumeration zu durchlaufen, und die Einträge der Comboboxen zu aktualisieren. Wählt der Benutzer beispielsweise einen anderen Adapter, so müssen die Einträge der anderen Comboboxen mit den verschiedenen Betriebsarten dieses Adapters gefüllt

werden usw. Die entsprechenden Funktionen sind wiederum größtenteils analog zu denen des DirectX SDK. Sehen wir uns also noch die beiden verbleibenden Steuerelemente an, nämlich die Buttons für »OK« und »Abbrechen«.

Beenden eines Dialogs

```
BOOL EndDialog(HWND hWnd, int nResult);
```

Der erste Parameter der Funktion ist natürlich das Handle auf den Dialog, der beendet werden soll. Der zweite Parameter ist der Wert, den die Funktion, durch die der Dialog erzeugt wurde, als Rückgabewert liefern soll. In diesem Fall war das die Funktion `DialogBox()`. Wurde also der ABBRECHEN-Button im Dialog angeklickt, dann geben wir den Wert 0 zurück, um ein Abbrechen des Dialogs ohne Fehler anzuzeigen. Mehr müssen wir beim Anklicken des ABBRECHEN-Buttons nicht tun.

Wurde nun aber der OK-Button im Dialog angeklickt, dann haben wir etwas mehr zu tun. Jetzt ist davon auszugehen, dass der Benutzer in dem Dialog die Grafikkarte, die Bildschirmauflösung und alle weiteren Einstellungen gewählt hat und das Render-Device starten möchte. In diesem Fall holen wir uns die Einträge und Zustände aller Steuerelemente des Dialogs über die Funktion `ZFXD3DEnum::GetSelections` und speichern ihre aktuellen Einstellungen in den entsprechenden globalen Variablen bzw. Attributen der Klasse `ZFXD3D`. Die Struktur `ZFXDEVICEINFO` beinhaltet dabei ein paar Werte aus der Enumeration, beispielsweise die Eigenschaften des Devices, zu welchem Adapter es gehört und welche Modi es verwenden kann. Diese Struktur ist aber ebenfalls fast analog zu dem Äquivalent aus der Enumeration des DirectX SDK.

Dann beenden wir den Dialog und geben den Wert 1 dabei zurück, der andeuten soll, dass der Dialog erfolgreich beendet wurde. Wie das Programm nun auf das Beenden des Dialogs reagiert, sehen wir gleich in der Initialisierungsfunktion.

Initialisierung, Enumeration und Shutdown

Zunächst einmal rufen wir in der aus der DLL exportierten Funktion `CreateRenderDevice()` den Konstruktor der ZFXD3D-Klasse auf. Dieser sieht wie folgt aus:


```
ZFXD3D *g_ZFXD3D=NULL;

ZFXD3D::ZFXD3D(HINSTANCE hDLL) {
   m_hDLL               = hDLL;
   m_pEnum              = NULL;
   m_pD3D               = NULL;
   m_pDevice            = NULL;
```

```
    m_pLog              = NULL;
    m_ClearColor        = D3DCOLOR_COLORVALUE(
                          0.0f, 0.0f, 0.0f, 1.0f);
    m_bRunning          = false;
    m_bIsSceneRunning   = false;

    m_nActivehWnd       = 0;

    g_ZFXD3D = this;
    }
```

Es handelt sich hierbei um einen vollkommen durchschnittlichen Konstruktor, der nur dazu genutzt wird, ein paar Startwerte für die Attribute festzulegen. Beachtet, dass wir hierbei das Handle auf die geladene DLL in einem Attribut speichern. Bemerkenswert ist hier lediglich, dass wir uns in einem globalen Pointer namens `g_ZFXD3D` noch die Adresse des durch den Konstruktor erstellten Objekts merken. Diese benötigen wir später noch einmal in unserem Programm, und zwar für die Callback-Funktion des Dialogs. Der Destruktor der Klasse sieht wie folgt aus:


```
ZFXD3D::~ZFXD3D() {
    Release();
    }

void ZFXD3D::Release() {
    if (m_pEnum) {
        m_pEnum->~ZFXD3DEnum();
        m_pEnum = NULL;
        }
    if(m_pDevice) {
        m_pDevice->Release();
        m_pDevice = NULL;
        }
    if(m_pD3D) {
        m_pD3D->Release();
        m_pD3D = NULL;
        }
    fclose(m_pLog);
    }
```

Wir müssen nur darauf achten, dass wir auch brav wieder alle Objekte freigeben, die initialisiert worden sein könnten.

Sehen wir uns die Funktion für die Initialisierung an. Dort werden also die spannenden Sachen passieren, wenn das schon im Konstruktor nicht der Fall war. Diese Funktion können wir später aus anderen Anwendungen heraus separat aufrufen, nachdem wir über die exportierten Funktionen ein Objekt der Klasse `ZFXD3D` erhalten haben.


```
HBITMAP g_hBMP;

HRESULT ZFXD3D::Init(HWND hWnd, const HWND *hWnd3D,
                     int nNumhWnd, int nMinDepth,
                     int nMinStencil, bool bSaveLog) {
   int nResult;

   m_pLog = fopen("log_renderdevice.txt", "w");
   if (!m_pLog) return ZFX_FAIL;

   // sollen wir Child-Windows verwenden?
   if (nNumhWnd > 0) {
      if (nNumhWnd > MAX_3DHWND) nNumhWnd = MAX_3DHWND;
      memcpy(&m_hWnd[0], hWnd3D, sizeof(HWND)*nNumhWnd);
      m_nNumhWnd = nNumhWnd;
      }
   // sonst speichern wir das Handle des Hauptfensters
   else {
      m_hWnd[0] = hWnd;
      m_nNumhWnd = 0;
      }
   m_hWndMain = hWnd;;

   if (nMinStencil > 0) m_bStencil = true;

   // Erzeuge das Enum-Objekt
   m_pEnum = new ZFXD3DEnum(nMinDepth, nMinStencil);

   // Lade das ZFX-Logo
   g_hBMP = (HBITMAP)LoadImage(NULL, "zfx.bmp",
                               IMAGE_BITMAP,0,0,
                               LR_LOADFROMFILE |
                               LR_CREATEDIBSECTION);

   // Öffne den Auswahl-Dialog
   nResult = DialogBox(m_hDLL, "dlgChangeDevice", hWnd,
                       DlgProcWrap);

   // Ressource wieder freigeben
   if (g_hBMP) DeleteObject(g_hBMP);

   // Fehler im Dialog
   if (nResult == -1)
      return ZFX_FAIL;
   // Dialog vom Benutzer abgebrochen
   else if (nResult == 0)
      return ZFX_CANCELED;
   // Dialog mit OK-Button beendet
   else
      return Go();
   }
```

Wohin mit der Grafikausgabe?

Dieser Funktion können wir sechs Parameter übergeben. Zuerst einmal ist dies ein Handle auf das Hauptfenster der Anwendung. Dies benötigen wir einerseits, um den Dialog als Kind dieses Fensters anzuzeigen, und andererseits wird dieses Fenster für die Grafik-Darstellung von unserem Render-Device verwendet, wenn wir eine Applikation im Fullscreen-Modus ausführen. Möchten wir jedoch die Anwendung im Fenster starten, so können wir im zweiten Parameter optional ein ganzes Array von Handles auf Child-Windows angeben, die für die Grafikausgabe verwendet werden sollen. Das ist beispielsweise bei Editoren sinnvoll, bei denen die Grafikausgabe nur einen Teil des Programm-Fensters einnimmt. Über dieses Array können wir also beliebig viele (na gut, nur `MAX_3DHWND` Stück) Child-Windows angeben. Diese können dann alle zum Rendern von Grafik über unser Device benutzt werden. Die Umschaltung zwischen diesen verschiedenen Fenstern (beim Rendern kann ja immer nur eins aktiv sein) erledigen wir über die Funktion `ZFXD3D::UseWindow`.

Der dritte Parameter gibt schließlich an, wie viele Handles auf Child-Windows in dem Array zu finden sind. Die folgenden beiden Parameter geben an, wie viele Bits der Depth-Buffer und der Stencil-Buffer jeweils mindestens haben sollen. Der letzte Parameter dient dazu, ein abgesichertes Log zu schreiben, aber das brauchen wir erst in späteren Kapiteln.

Anzeige des Dialogs und Callback-Funktion

Wie wir aber unschwer erkennen können, macht die Initialisierungsfunktion eigentlich auch nicht viel. Nach dem Umspeichern von Handles in Member-Variablen und dem Laden unseres Eye-Candy, des ZFX-Logos aus einer Bitmap-Datei, ruft sie lediglich die WinAPI-Funktion `DialogBox()` auf, um unseren Dialog namens *dlgChangeDevice* anzuzeigen. Als Rückgabewerte sind hier, durch die Callback-Funktion, drei Werte möglich. Falls der Benutzer den Dialog korrekt über den OK-Button beendet hat, rufen wir die Funktion `ZFXD3D::Go` auf. Diese wird dann Direct3D mit den im Dialog gewählten Einstellungen starten. Aber dazu kommen wir gleich. Zuerst schauen wir uns die Callback-Funktion näher an. Wir verwenden hier die Funktion `DlgProcWrap()`, weil wir als Callback-Funktion keine normale Member-Funktion einer Klasse angeben können. Also brauchen wir ein kleines Stützkorsett, das uns dabei hilft. Und das ist eben diese Funktion:

LIST

```
BOOL CALLBACK DlgProcWrap(HWND hDlg,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam) {
   return g_ZFXD3D->DlgProc(hDlg,
                            message,
                            wParam,
                            lParam);
   }
```

Aufgabe der Callback-Funktion

Über das globale Objekt unserer Klasse, das wir im Konstruktor gespeichert haben, leiten wir den Aufruf der Callback-Funktion einfach auf die entsprechende Member-Funktion unserer Klasse um. Das ist deshalb notwendig, weil wir in der Callback-Funktion auf diverse Member-Attribute der Klasse zugreifen wollen. Die Implementierung der Funktion `ZFXD3D::DlgProc` haben wir ja vorhin schon gesehen. In dieser Funktion geht es hauptsächlich um die Enumeration und die Auflistung der vorhandenen Grafikkarten und Bildschirmmodi. Die entsprechenden Funktionen, die die Callback-Funktion dafür aufruft, werden dann über die Klasse `ZFXD3DEnum` implementiert.

Direct3D starten

Nun können wir endlich dazu kommen, wie wir Direct3D starten. Wird die Dialogbox durch Anklicken des OK-Buttons beendet, liest die Callback-Funktion die gewählten Einträge aus den Steuerelementen und kehrt zu der Initialisierungsfunktion zurück. Dort wird dann die Funktion `ZFXD3D::Go` aufgerufen.


```
HRESULT ZFXD3D::Go(void) {
   ZFXCOMBOINFO   xCombo;
   HRESULT        hr;
   HWND           hwnd;

   // Erzeuge das Direct3D-Hauptobjekt
   if (m_pD3D) {
      m_pD3D->Release();
      m_pD3D = NULL;
      }
   m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );
   if(!m_pD3D)
      return ZFX_CREATEAPI;

   // Finde die passende Combo
   for (UINT i=0; i<g_xDevice.nNumCombo; i++) {
      if ( (g_xDevice.d3dCombo[i].bWindowed  ==
            m_bWindowed)
        && (g_xDevice.d3dCombo[i].d3dDevType ==
            g_xDevice.d3dDevType)
        && (g_xDevice.d3dCombo[i].fmtAdapter ==
            g_fmtA)
        && (g_xDevice.d3dCombo[i].fmtBackBuffer ==
            g_fmtB) )
         {
         xCombo = g_xDevice.d3dCombo[i];
         break;
         }
      }

   // Betanke die PresentParameters-Struktur
   ZeroMemory(&m_d3dpp, sizeof(m_d3dpp));
   m_d3dpp.Windowed               = m_bWindowed;
```

```
m_d3dpp.BackBufferCount        = 1;
m_d3dpp.BackBufferFormat       = g_Dspmd.Format;
m_d3dpp.EnableAutoDepthStencil = TRUE;
m_d3dpp.MultiSampleType        = xCombo.msType;
m_d3dpp.AutoDepthStencilFormat =
                               xCombo.fmtDepthStencil;
m_d3dpp.SwapEffect             =
                              D3DSWAPEFFECT_DISCARD;

// ist ein Stencil-Buffer aktiv??
if (  (xCombo.fmtDepthStencil == D3DFMT_D24S8)
   || (xCombo.fmtDepthStencil == D3DFMT_D24X4S4)
   || (xCombo.fmtDepthStencil == D3DFMT_D15S1) )
   m_bStencil = true;
else
   m_bStencil = false;

// Fullscreen-Modus
if (!m_bWindowed) {
   m_d3dpp.hDeviceWindow    = hwnd = m_hWndMain;
   m_d3dpp.BackBufferWidth  = g_Dspmd.Width;
   m_d3dpp.BackBufferHeight = g_Dspmd.Height;
   ShowCursor(FALSE);
   }
// Windowed-Modus
else {
   m_d3dpp.hDeviceWindow     = hwnd = m_hWnd[0];
   m_d3dpp.BackBufferWidth =
                        GetSystemMetrics(SM_CXSCREEN);
   m_d3dpp.BackBufferHeight =
                        GetSystemMetrics(SM_CYSCREEN);
   }

// Erstelle das Direct3D-Device
hr = m_pD3D->CreateDevice(g_xDevice.nAdapter,
                          g_xDevice.d3dDevType,
                          m_hWnd, xCombo.dwBehavior,
                          &m_d3dpp, &m_pDevice);

// Swap Chains erstellen, falls nötig
if ( (m_nNumhWnd > 0) && m_bWindowed) {
   for (UINT i=0; i<m_nNumhWnd; i++) {
      m_d3dpp.hDeviceWindow = m_hWnd[i];
      m_pDevice->CreateAdditionalSwapChain(
                     &m_d3dpp, &m_pChain[i]);
      }
   }

m_pEnum->~ZFXD3DEnum();
m_pEnum = NULL;
```

```
if(FAILED(hr))
   return ZFX_CREATEDEVICE;

m_bRunning        = true;
m_bIsSceneRunning = false;
return ZFX_OK;
} // go
```

Struktur erstellen und Device initialisieren

In dieser Funktion tun wir nichts anderes, als alle Combos vom Typ `ZFXCOMBOINFO` für das ausgewählte Device zu durchlaufen. Mit den entsprechenden Werten der Combo befüllen wir dann die `D3DPRESENT_PARAMETERS`-Struktur und initialisieren das Direct3D-Device durch den Aufruf von `IDirect3D8::CreateDevice`. Wir müssen dabei lediglich ein paar Unterscheidungen der beiden Fälle Fullscreen- oder Fenster-Modus beachten. Ab diesem Moment ist unser Render-Device dann einsatzbereit, immer vorausgesetzt, es sind keine Fehler aufgetreten. Eine Combo ist hier allerdings nicht mit einem Steuerelement *Combobox* zu verwechseln. Microsoft verwendet diese Bezeichnung seit DirectX 9 dafür, um eine Kombination von Front-Buffer- und Back-Buffer-Format, Adapter, Device-Typ, Vertex-Processing-Art und Depth-Stencil-Buffer-Format zusammenzufassen. Bei der Enumeration wurden entsprechend alle diese Combos für jedes vorhandene Device aller Adapter erzeugt.

Swap Chains

Am Ende der Funktion lüften wir dann das Geheimnis, wie wir durch Direct3D in beliebig viele Child-Windows rendern können. Falls der Benutzer das Programm im Fenster-Modus startet und die Applikation ein oder mehrere Child-Window-Handles angegeben hat, dann erstellen wir für jedes Child-Window ein eigenes Direct3D-Swap-Chain-Objekt. So können wir das Hauptfenster der Applikation weiterhin als normales Fenster nutzen und die Grafikausgabe in ein oder mehrere seiner Child-Windows realisieren.

Im bisherigen Design würden wir den Anwender unserer DLL immer dazu zwingen, den Dialog aufzurufen, wenn er das Render-Device nutzen möchte. Dies ist sicherlich für Programme sinnvoll, die für den Fullscreen-Modus entwickelt wurden. Hier kann der Benutzer frei die Bildschirmauflösung bestimmen oder das Programm eben doch im Fenster-Modus laufen lassen.

Wenn man allerdings ein Programm entwickelt, das nur im Fenster-Modus laufen soll, beispielsweise einen Editor, dann macht das wenig Sinn. Hierzu habe ich in das Interface zusätzlich noch die Funktion `ZFXRenderDevice::InitWindowed` *als Alternative zu dem Aufruf von* `ZFXRenderDevice::Init` *eingefügt. Diese ist eine Mischung aus der* `Init()`*- und der* `Go()`*-Methode und initialisiert das Device ohne Aufruf des Dialogs gleich im Fenstermodus. Für die Parameterliste ist die Angabe der Bytes für den Depth- und Stencil-Buffer daher unnötig.*

Zwischen Child-Windows wechseln

Haben wir der `ZFXD3D::Init`-Methode tatsächlich ein Array von mehreren Handles auf Child-Windows angegeben, so müssen wir natürlich auch jederzeit in der Lage sein, eines der Child-Windows als aktives Render-Ziel für Direct3D einzustellen. Per Voreinstellung nutzen wir im Fenstermodus das erste Array-Element mit dem Index 0 (oder eben das Hauptfenster, falls gar keine Child-Windows angegeben sind).

Swap Chains wechseln

Mit Hilfe der folgenden Funktion können wir also zwischen den Child-Windows umschalten, sofern mehrere vorhanden sind. Dazu muss man wissen, dass Direct3D für jedes Child-Window, für das wir ein Swap-Chain-Objekt erzeugt haben, einen eigenen Back-Buffer erstellt hat. Möchten wir jetzt das aktive Swap-Chain-Objekt (also das Child-Window) ändern, dann müssen wir von dem zu aktivierenden Swap-Chain-Objekt einen Pointer auf dessen Back-Buffer abfragen und genau diesen Back-Buffer dann als neues Render-Target für das Direct3D-Device setzen. An den Aufrufen zum Rendern, Löschen des Back-Buffers usw. brauchen wir dabei überhaupt nichts zu verändern.

{} LIST

```
HRESULT ZFXD3D::UseWindow(UINT nHwnd) {
   LPDIRECT3DSURFACE9 pBack=NULL;

   if (!m_d3dpp.Windowed)
      return ZFX_OK;
   else if (nHwnd >= m_nNumhWnd)
      return ZFX_FAIL;

   // versuche, den richtigen Back-Buffer zu holen
   if (FAILED(m_pChain[nHwnd]->GetBackBuffer(0,
             D3DBACKBUFFER_TYPE_MONO, &pBack)))
      return ZFX_FAIL;

   // aktiviere ihn für das Device
   m_pDevice->SetRenderTarget(0, pBack);
   pBack->Release();
   m_nActivehWnd = nHwnd;
   return ZFX_OK;
   }
```

Einfacher geht's nicht

Damit sind wir nun in der Lage, beliebig viele Ansichten mit 3D-Grafik für unsere Applikationen zu erzeugen, und zwar ganz einfach dadurch, dass wir die Handles der Child-Windows der `ZFXD3D::Init`-Funktion übergeben und per Aufruf der Methode `ZFXD3D::UseWindow` ein beliebiges dieser Fenster aktivieren und dann Grafik rendern.

Demo-Applikation

Abbildung 3.2 zeigt einen Screenshot des Beispielprogramms dieses Kapitels. Dort sind vier Child-Windows im Hauptfenster der Applikation angelegt, die der Initialisierungsfunktion unseres Devices als zu verwendende

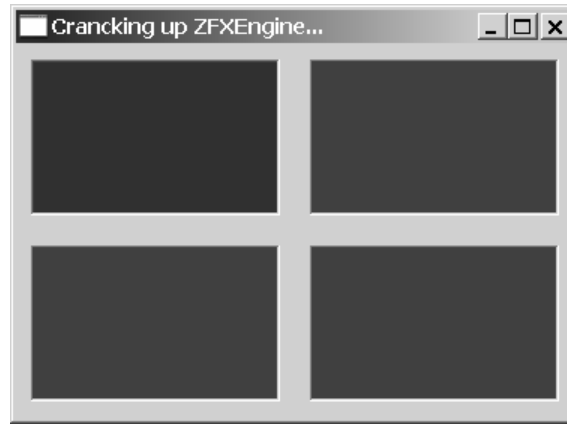

Abbildung 3.2: Vier Child-Windows im Hauptfenster

Fenster bekannt gemacht wurden. Startet man die Engine durch Auswahl im Dialog im Fenster-Modus, dann kann man diese vier Child-Windows verwenden. Startet man über den Dialog das Programm jedoch im Fullscreen-Modus, dann werden alle vier Child-Windows ignoriert, und das Hauptfenster der Anwendung wird zum Rendern verwendet. Den kompletten Quelltext der Demo-Applikation findet ihr am Ende dieses Kapitels.

Render-Funktionen

Jetzt haben wir es geschafft, unser Render-Device aus der DLL komfortabel zu initialisieren und können es auch wieder freigeben. Allerdings – fehlt da nicht noch etwas? Na klar, wir müssen ja auch irgendwie mit dem Ding arbeiten können! Wirklich etwas rendern werden wir jetzt noch nicht, lediglich den Bildschirm mit der Hintergrundfarbe löschen. Unter Direct3D benötigt man auch noch Funktionen, die zu Beginn und am Ende eines Render-Vorgangs aufgerufen werden müssen, um interne Strukturen für das Rendern einzustellen. Wir werden hier das Löschen der Buffer gleich mit diesem *Starten der Szene* vor dem Rendern verbinden. Ebenso müssen wir eine im Pixel-Buffer existierende gerenderte Szene auf den Bildschirm bringen (also vom Back-Buffer in den Front-Buffer flippen). Diesen Aufruf koppeln wir in unserer Implementierung an das Beenden der Szene nach dem Rendern.

Hier sind die Funktionen unseres Render-Devices, mit denen wir das alles umsetzen:

{} LIST

```
HRESULT ZFXD3D::BeginRendering(bool bClearPixel,
                               bool bClearDepth,
                               bool bClearStencil) {
   DWORD dw=0;
```

```
   // soll irgendetwas gelöscht werden?
   if (bClearPixel || bClearDepth || bClearStencil) {
      if (bClearPixel)   dw |= D3DCLEAR_TARGET;
      if (bClearDepth)   dw |= D3DCLEAR_ZBUFFER;

      if (bClearStencil && m_bStencil)
         dw |= D3DCLEAR_STENCIL;

      if (FAILED(m_pDevice->Clear(0, NULL, dw,
                                  m_ClearColor,
                                  1.0f, 0)))
            return ZFX_FAIL;
      }

   if (FAILED(m_pDevice->BeginScene()))
      return ZFX_FAIL;

   m_bIsSceneRunning = true;
   return ZFX_OK;
   }
/*-----------------------------------------------------*/

HRESULT ZFXD3D::Clear(bool bClearPixel,
                      bool bClearDepth,
                      bool bClearStencil) {
   DWORD dw=0;

   if (bClearPixel)   dw |= D3DCLEAR_TARGET;
   if (bClearDepth)   dw |= D3DCLEAR_ZBUFFER;

   if (bClearStencil && m_bStencil)
      dw |= D3DCLEAR_STENCIL;

   if (m_bIsSceneRunning)
      m_pDevice->EndScene();

   if (FAILED(m_pDevice->Clear(0, NULL, dw,
                               m_ClearColor,
                               1.0f, 0)))
         return ZFX_FAIL;

   if (m_bIsSceneRunning)
      m_pDevice->BeginScene();
   }
/*-----------------------------------------------------*/

void ZFXD3D::EndRendering(void) {
   m_pDevice->EndScene();
   m_pDevice->Present(NULL, NULL, NULL, NULL);
```

```
    m_bIsSceneRunning = false;
    }
/*-------------------------------------------------*/

void ZFXD3D::SetClearColor(float fRed,
                           float fGreen,
                           float fBlue) {
    m_ClearColor = D3DCOLOR_COLORVALUE(fRed,
                                       fGreen,
                                       fBlue,
                                       1.0f);
    }
```

Löschen der Buffer

Die Funktion für das Starten der Szene ist auch so flexibel gehalten, dass wir angeben können, welche der vorhandenen Buffer gelöscht werden sollen. Bei vielen Anwendungen, die den gesamten Pixel-Buffer in einem Frame neu rendern, ist es beispielsweise unnötig, den Pixel-Buffer zu löschen. Ebenso können wir auf das Löschen des Stencil-Buffers verzichten, wenn wir diesen gar nicht verwenden oder er eine konstante Maske enthält, die sich im Verlauf des Programms nicht ändert. Hierbei ist aber zu beachten, dass das Flag für den Stencil-Buffer auf keinen Fall gesetzt sein darf, wenn unser Programm diesen nicht verwendet. Das würde zu einem Fehlschlag der DirectX-Funktion `Clear()` führen. Ebenfalls gilt es zu beachten, dass wir die Szene sozusagen kurz unterbrechen müssen, wenn der Benutzer einen oder mehrere der Buffer löschen möchte, während die Szene noch nicht beendet wurde. Um diese Möglichkeit zu überwachen dient das Attribut `m_IsSceneRunning`.

Die letzte der vier gezeigten Funktionen dient dazu, die Hintergrundfarbe für den Löschvorgang zu ändern, sollte dies nötig werden. Normalerweise ist das aber nur einmal zu Beginn einer Anwendung nötig, um die gewünschte Farbe der Anwendung einzustellen.

Nun haben wir alles für dieses Kapitel komplett, was die Implementierung unseres Interfaces über eine DLL betrifft. Im verbleibenden Teil dieses Kapitels entwerfen wir eine Rahmenanwendung, die die Verwendung unserer bisherigen Arbeit an einem konkreten, lauffähigen Beispiel demonstriert.

3.6 Testlauf der Implementierung

Da wir alle schon einmal mit einer 3D-API gearbeitet haben, wissen wir, wie viel Arbeit es sein kann, diese zu starten, alle möglichen Fehler abzufangen, Bildschirmmodi auszuwählen usw. Halt der ganze Kram, den wir gerade in die DLL gestopft haben. Umso mehr wird uns der Quelltext der nun folgenden Anwendung erschrecken, weil er durch Verwendung der DLL so kurz und einfach gehalten werden kann.

Ein neuer Arbeitsbereich

Wir öffnen also Visual C++ und legen einen neuen Arbeitsbereich für eine *Win32-Anwendung* an. Diese nennen wir schlicht und einfach *Demo* und fügen ihr die neuen Dateien `main.cpp` und `main.h` hinzu. Um Verwirrung zu vermeiden, kopieren wir die nötigen Dateien aus dem Verzeichnis *ZFXRenderer* in das neue Verzeichnis *Demo*. Das ist nur deshalb nötig, weil ich hier keinen festen Ordner auf unserer Festplatte voraussetze, in den wir unsere statische und dynamische Bibliothek sowie die notwendigen Header hineinerstellt bzw. kompiliert haben. Das wäre aber die praktischere Methode, die jeder zu Hause bei sich anwenden sollte. Kopieren müssen wir also die folgenden Dateien:

- `ZFXRenderer.h`
- `ZFXRenderDevice.h`
- `ZFXRenderer.lib`
- `ZFXD3D.dll`

Nun können wir loslegen. In unserem Programm benötigen wir nur vier kurze Funktionen. Eine davon ist natürlich die `WinMain()`-Funktion, also der Eintrittspunkt eines jeden Windows-Programms. Diese geht natürlich Hand in Hand mit einer Callback-Funktion für die der Anwendung automatisch von Windows zugeteilten Nachrichten. Die anderen beiden Funktionen schreiben wir für die Initialisierung und das Beenden des Programms, um unser Render-Device zu erstellen und freizugeben. Das komplette Programm sieht wie folgt aus:

Listing 3.1: Demo-Applikation für die Verwendung der ZFXD3D-DLL

```
//////////////////////////////////////////////////////
// FILE: main.h
LRESULT WINAPI MsgProc(HWND, UINT, WPARAM, LPARAM);
HRESULT ProgramStartup(char *chAPI);
HRESULT ProgramCleanup(void);
//////////////////////////////////////////////////////


//////////////////////////////////////////////////////
// FILE: main.cpp

#define WIN32_MEAN_AND_LEAN

#include "ZFXRenderer.h"  // Unser Interface
#include "ZFX.h"          // Rückgabewerte
#include "main.h"         // Prototypen

// Unsere statische Bibliothek einbinden
#pragma comment(lib, "ZFXRenderer.lib")

// Windows-Kram
HWND      g_hWnd  = NULL;
```

```
HINSTANCE g_hInst = NULL;
TCHAR     g_szAppClass[] = TEXT("FrameWorktest");

// Anwendungskram
BOOL  g_bIsActive = FALSE;
bool  g_bDone     = false;
FILE *pLog        = NULL;

// ZFX-Render- und -RenderDevice-Objekte
LPZFXRENDERER     g_pRenderer = NULL;
LPZFXRENDERDEVICE g_pDevice   = NULL;


/**
 * WinMain-Funktion als Startpunkt.
 */
int WINAPI WinMain(HINSTANCE hInst,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow) {
   WNDCLASSEX      wndclass;
   HRESULT         hr;
   HWND            hWnd;
   MSG             msg;

   // Fenster-Attribute initialisieren
   wndclass.hIconSm    = LoadIcon(NULL,IDI_APPLICATION);
   wndclass.hIcon      = LoadIcon(NULL,IDI_APPLICATION);
   wndclass.cbSize        = sizeof(wndclass);
   wndclass.lpfnWndProc   = MsgProc;
   wndclass.cbClsExtra    = 0;
   wndclass.cbWndExtra    = 0;
   wndclass.hInstance     = hInst;
   wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW);
   wndclass.lpszMenuName  = NULL;
   wndclass.lpszClassName = g_szAppClass;
   wndclass.style         = CS_HREDRAW | CS_VREDRAW |
                            CS_OWNDC | CS_DBLCLKS;

   if(RegisterClassEx(&wndclass) == 0)
      return 0;

   if (!(hWnd = CreateWindowEx(NULL, g_szAppClass,
                  "Cranking up ZFXEngine...",
                  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                  GetSystemMetrics(SM_CXSCREEN)/2 -190,
                  GetSystemMetrics(SM_CYSCREEN)/2 -140,
                  380, 280, NULL, NULL, hInst, NULL)))
      return 0;
```

```
   g_hWnd  = hWnd;
   g_hInst = hInst;

   pLog = fopen("log_main.txt", "w");

   // Starte die Engine
   if (FAILED( hr = ProgramStartup("Direct3D"))) {
      fprintf(pLog, "error: ProgramStartup() failed\n");
      g_bDone = true;
      }
   else if (hr == ZFX_CANCELED) {
      fprintf(pLog, "ProgramStartup() canceled\n");
      g_bDone = true;
      }
   else
      g_pDevice->SetClearColor(0.1f, 0.3f, 0.1f);

   while (!g_bDone)
      {
      while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }

      if (g_bIsActive)
         {
         if (g_pDevice->IsRunning()) {
            g_pDevice->BeginRendering(true,true,true);
            g_pDevice->EndRendering();
            }
         }
      }

   // Cleanup-Stuff
   ProgramCleanup();

   UnregisterClass(g_szAppClass, hInst);
   return (int)msg.wParam;
   } // WinMain
/*-----------------------------------------------------*/

/**
 * MsgProc zur Nachrichtenverarbeitung.
 */
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg,
                       WPARAM wParam,
                       LPARAM lParam) {
   switch(msg) {
```

```
      // Anwendungsfokus
      case WM_ACTIVATE: {
         g_bIsActive = (BOOL)wParam;
         } break;

      // Tastaturereignis
      case WM_KEYDOWN: {
         switch (wParam) {
            case VK_ESCAPE: {
               g_bDone = true;
               PostMessage(hWnd, WM_CLOSE, 0, 0);
               return 0;
               } break;
            }
         } break;

      // Zerstöre das Fensterobjekt
      case WM_DESTROY: {
         g_bDone = true;
         PostQuitMessage(0);
         return 1;
         } break;

      default: break;
      }
   return DefWindowProc(hWnd, msg, wParam, lParam);
   }
/*-----------------------------------------------------*/

/**
 * Erstelle ein RenderDevice-Objekt.
 */
HRESULT ProgramStartup(char *chAPI) {
   HWND hWnd3D[4];
   RECT rcWnd;
   int  x=0,y=0;

   // Wir haben noch keine OpenGL-DLL ...
   if (strcmp(chAPI, "OpenGL")==0) return S_OK;

   // Erstelle einen Renderer
   g_pRenderer = new ZFXRenderer(g_hInst);

   // Erstelle ein RenderDevice
   if (FAILED( g_pRenderer->CreateDevice(chAPI) ))
      return E_FAIL;

   // Speichere einen Pointer auf das Device
   g_pDevice = g_pRenderer->GetDevice();
   if(g_pDevice == NULL) return E_FAIL;
```

```
   // Größe des Fensterbereichs abfragen
   GetClientRect(g_hWnd, &rcWnd);

   for (int i=0; i<4; i++) {
      if ( (i==0) || (i==2) ) x = 10;
      else x = rcWnd.right/2 + 10;

      if ( (i==0) || (i==1) ) y = 10;
      else y = rcWnd.bottom/2 + 10;

      hWnd3D[i] = CreateWindowEx(WS_EX_CLIENTEDGE,
                  TEXT("static"), NULL, WS_CHILD |
                  SS_BLACKRECT | WS_VISIBLE, x, y,
                  rcWnd.right/2-20, rcWnd.bottom/2-20,
                  g_hWnd, NULL, g_hInst, NULL);
      }

   // Device initialisieren (Dialogbox anzeigen)
   return g_pDevice->Init(g_hWnd,   // Hauptfenster
                          hWnd3D,   // Child-Windows
                          4,        // 4 Children
                          16,       // 16-Bit-Z-Buffer
                          0,        // 0-Bit-Stencil
                          false);
   } // ProgramStartup
/*------------------------------------------------------*/

/**
 * Freigabe der verwendeten Ressourcen.
 */
HRESULT ProgramCleanup(void) {
   if (g_pRenderer) {
      delete g_pRenderer;
      g_pRenderer = NULL;
      }
   if (pLog)
      fclose(pLog);
   return S_OK;
   } // ProgramCleanup
/*------------------------------------------------------*/
```

Mit dem Render-Device arbeiten

Hierbei handelt es sich ja um wenig mehr als eine typische minimale Windows-Anwendung. Das kleine Bisschen mehr, was wir hier haben, findet fast alles in der Funktion `ProgramStartup()` statt. Als Erstes erzeugen wir eine Instanz der Klasse `ZFXRenderer` aus unserer statischen Bibliothek. Dieser Konstruktor macht noch nicht viel. Dann rufen wir aber die Initialisierungsfunktion für unser Render-Device, also `ZFXRenderer::CreateDevice`, des Renderers auf. Der Renderer lädt daraufhin die passende DLL (hier `ZFXD3D.dll`) und erzeugt eine Instanz der Klasse `ZFXD3D`, also das eigentliche

Render-Device. Auf diese Instanz lassen wir uns dann einen Pointer geben, den wir global speichern, um ihn überall im Programm verfügbar zu haben.

Child-Windows

Dann basteln wir uns automatisiert vier Child-Windows, die gleichmäßig im Hauptfenster angeordnet sind (siehe Abbildung 3.2). Im letzten Schritt rufen wir die Initialisierungsfunktion für das Render-Device auf. Diese Funktion bringt den Dialog für die Auswahl der Grafikkarte und deren Modi zur Anzeige.

Sobald der Benutzer den Dialog abgehandelt hat, läuft das Programm weiter. In der Hauptschleife löschen wir einfach so lange den Bildschirm in jedem Frame, bis der Benutzer das Programm durch Drücken von Esc beendet oder im Fenster-Modus auf das kleine Kreuzchen oben rechts im Fenster klickt.

Wenn man das Programm nun startet, erscheint zunächst das Fenster, dann der Dialog. Dieser benötigt eine kleine Weile, um die vorhandene Hardware zu untersuchen, um sie in den Steuerelementen anzuzeigen. Jetzt kann der Benutzer einen Bildschirmmodus und eine Grafikkarte (falls mehrere vorhanden sind) auswählen und das Programm starten. Im Fenster-Modus werden die vier Child-Windows angezeigt, im Fullscreen-Modus natürlich lediglich ein nackter Bildschirm, der das Hauptfenster verwendet.

3.7 Ein Blick zurück, zwei Schritt nach vorn

Solide Basis

Rückblickend kann man sagen, dass wir in diesem Kapitel unsere Grundkenntnisse in verschiedenen Bereichen der Programmierung wieder haben aufleben lassen. Zudem hoffe ich, dass wir unsere Kenntnisse in einigen Bereichen, speziell was den Umgang mit DLLs angeht, auch erweitern konnten. Hier haben wir es geschafft, uns die Basis für eine DLL zu schaffen, in der wir die Verwendung einer API kapseln können. Diese DLL, die nun unsere eigene ZFX-Grafik-API ist, beliebig zu erweitern stellt auch kein Problem mehr dar. An dem Code zum Laden und zur Verwendung der DLL ändert sich nichts mehr.

Flexibilität

Ein weiterer Punkt, der für viele neu gewesen sein wird, ist das flexible Handling von beliebig vielen Child-Windows, die wir gleich im Initialisierungsaufruf unseres Devices angeben können. Durch diese eine Zeile Quelltext erhalten wir die Möglichkeit, beispielsweise die vier typischen Fenster (von rechts, von vorne, von oben und 3D-Ansicht) eines Modell/Level-Editors anzulegen. Zusätzlich können wir auch noch ein Child-Window anlegen, in dem wir dann ausgewählte Texturen anzeigen oder eine Preview eines einfügbaren 3D-Objekts usw. Später werden wir diesen flexiblen Ansatz noch weiter ausbauen, indem wir in jedem Child-Window beliebig viele Viewports zulassen werden. Während die Child-Windows lediglich

nebeneinander angeordnet sein können, bieten die Viewports die Möglichkeit, Fenster in einem Fenster anzulegen, also beispielsweise einen kleinen Bereich in der 3D-Ansicht, der eine andere Ansicht der Szene zeigt – beispielsweise die Sicht eines vorgeschobenen Aufklärers oder einen Rückspiegel.

Vorteile durch Verwendung einer eigenen Grafik-API

Welchen großen Vorteil wir durch die Verwendung einer eigenen Grafik-API, hier basierend auf Direct3D, erhalten, das zeigt unsere kleine Demo-Applikation. Wir benötigen lediglich vier kurze Funktionsaufrufe, um beispielsweise Direct3D zu starten. Dabei sind wir über unseren Dialog noch immer variabel genug, die API im Fenster-Modus oder im Fullscreen-Modus bei beliebiger Auflösung zu starten, ohne das irgendwo im Programm *hart verdrahten* zu müssen. Damit verschwindet der ganze hässliche Initialisierungs- und Auswahl-Kram komplett aus dem Quelltext einer Anwendung, was deutlich zu deren Übersichtlichkeit beiträgt. Und wenn eine neue DirectX-Version auf den Markt kommt? Auch das ist jetzt kein Problem mehr für uns. Die Zeiten, in denen wir Tausende Zeilen von Quellcode nach API-abhängigem Code durchsuchen mussten, sind endgültig vorbei. Wir laden einfach den Arbeitsbereich mit der DLL und ändern deren Code entsprechend, ohne die Funktionsprototypen anzufassen, die durch das Interface vorgeschrieben sind. Dann kompilieren wir die DLL neu und können sie sofort mit unseren fertig kompilierten anderen Projekten verwenden, ohne diese irgendwie verändern zu müssen.

Und jetzt?

Nun ist es an der Zeit, dass wir die Welt der Grafik-APIs verlassen. Dort draußen wartet noch etwas anderes Unheimliches auf uns, und zwar die 3D-Mathematik. Wenn wir unabhängig von einer Grafik-API bleiben wollen, müssen wir natürlich auch unsere eigenen Funktionen für 3D-Mathematik implementieren. Aus dieser Not werden wir jedoch eine Tugend machen, und diese Implementierung so umfangreich und komfortabel machen, dass wir uns die Arbeit an einer 3D-Engine ohne sie gar nicht mehr werden vorstellen können.