CHAPTER 3

Engineering the Engine

What you get if you don't get what you want is experience.

Desire Nick

You are probably tired of listening to design issues and general game programming and engine programming topics. If you think like me then you want to finally get down to work on some code. If so, this is the chapter for you. From now on, you will develop the ZFXEngine step-by-step, chapter-by-chapter. Don't worry. I will guide you.

This chapter covers the following objectives:

- Creating static and dynamic linked libraries (DLLs)
- Using Windows dialogs to interact with programs
- Developing a renderer for the ZFXEngine as a static library
- Initializing and cranking up Direct3D using a custom DLL
- Writing a framework program using the ZFXEngine

What Is an Interface?

An interface is a pretty smart concept related to the design of code. Note that we are talking about interfaces in object-oriented design here as opposed to user interfaces, for example. Just think about it: the bigger a software project becomes, the more problems arise from multiple different people screwing around with the code and sometimes screwing it up badly. Interfaces are just a way of insulating yourself from a particular implementation. In an object-oriented software engineering world, big projects should be made up of separate code objects that are reusable whenever possible and that are independent from your colleagues' work. However, how would you prevent a hard-core coder from changing the object's implementation, forcing you to rewrite the code that is related to this object?

An interface protects your work from this type of hard-core programmer. Interfaces are a concept independent of any particular language. Generally speaking, an interface sits between two objects or persons, and its task is to provide a means of communication between them. Using C++ you can define a class as an interface that uses a bunch of abstract methods. You might ask how this helps you to organize software projects.

It's actually fairly easy. Let's suppose your super-coder colleague has to implement an object A, and it's your job to implement object B that uses some of object A's public methods. You can define an interface for object A, and the implementation your colleague plays with has to inherit from the interface class you both agree on or that was given to you by the project's lead programmer. No matter how much the super-coder colleague messes around in his code, he has to provide only the methods declared in the interface—or risk being fired, as there is always more than one option.

Finally, you are in a quite secure position. Even if the implementation of object A stalls, you can do your implementation of object B based on the interface definition; the program will work with any and all objects that implement the interface. This is the basic interchangeability of interfaces. For different platforms you can have different implementations of the same interface.

Note

Basically you can define the layout of a class by just writing a specification document. But using interfaces is a means of enforcing the specification.

Abstract Classes

There are several ways to create interfaces. The most common way in object-oriented programming is to create an interface as a class that has no implementations. In C++, you create a class that declares only public member functions but provides no implementation for them. For those of you who have not yet dealt with C++, this might sound new. However, hang on, it gets even more difficult. The point is that you cannot instance an object from such a class because the compiler would complain that there is at least one implementation missing that such an object would need. However, it is not our intention to get objects from this interface class. Its only purpose is to act as the base class from which other classes inherit. Then those derived classes provide the implementations prescribed by the interface.

Now for the freaky stuff. Such functions you don't provide an implementation for are called *pure virtual functions*. A C++ class containing at least one pure virtual function is called an *abstract class* because you can't create instances of this class. Finally, such an abstract class is what we call an *interface*. So there is one more new thing to learn for you C coders who have avoided C++ until now. How do I declare a pure virtual function, and what is *virtual* supposed to mean?

Virtual Member Functions

Actually, from a coder's point of view it's pretty easy. You just have to use the C++ keyword `virtual` while declaring the function. I don't want to dive too deeply into C++ right now, but at least you should know the meaning of this keyword. To keep it simple, let me put it this way: If a class inherits from another class, it can declare and implement member functions featuring the same name as functions in the class they are derived from. Normally, an object would call the function from the base class if the derived class does not declare a function with the same name even if the object is of the derived class's type. However, if it does declare the same function, you can quickly mess up the code. Because with normal, non-virtual functions, the type of the pointer to your object, rather than the type of the object, is used to decide which function to call.

This means you can cast the pointer to your object of the derived class and use a pointer of the base class type instead. This way, your code would call the base class's function, which is normally not what you want. By declaring the function as virtual, it is now the type of the object that determines which implementation to call and not the type of the pointer you use to access the object. Take a look at the following line of code, which shows a sample of an abstract class:

```
class CMyClass
   {
   public:
      virtual void MyPureVirtualFunction( void ) = 0;
   };
```

This is as simple as it looks. The `=0` in the function declaration is the part responsible for making the function pure virtual: it tells the compiler that there must not be an implementation for this function in this class. Furthermore it forces each non-abstract class deriving from this base class to provide an implementation for this function. Otherwise the code could not be compiled. Guess what? You have now mastered what you need to know about declaring pure virtual functions and creating abstract classes.

Defining the Interface

You should have a quite good understanding of why a big software project needs interfaces and what their benefits are. And as you already might suppose, I designed the ZFXEngine to use interfaces. This engine is not a very big project, but it is larger than what you would call a small one. My goal in designing the engine is simply to prevent

API-dependency, as discussed previously. I will define an interface that exposes all functionalities that output graphics to the screen using the engine. After that, I will derive a class from this interface and implement the functions using Direct3D. Note that you can also derive a class of your own and implement the functions using OpenGL. Finally, I will put the implementation into a dynamic link library (DLL).

Note

Normally your compiler compiles your code into object files, which are then linked together to a program executable file by the linker. However, you can also link those object files together to a static library, which is in turn linked by any program that uses this library. If you link your object files as DLLs, each program using the library is not linked to it, but needs to load it at runtime.

About DLLs

The big advantage of DLLs should be clear. You can write a bunch of programs using the engine's interface, and you only need to load my implementation of the interface featuring Direct3D. Okay, okay. You can also load your own implementation featuring OpenGL or a software renderer. The point is that you can load the DLL at runtime. There is no need to recompile your code each time you make changes in the implementation of the engine because your code that uses the engine does not use the derived class in the DLL. The implementation is used via the interface. As long as the interface does not change, you can change whatever you want inside the DLL and provide the newly compiled version of it to your software project. You could even unload the old version during runtime and load the new one. If a new DirectX or OpenGL version comes along, you won't have a problem. You just need to rewrite your interface implementation inside the library and distribute the new library to your engine users. They don't need to rebuild their programs to use the new version. It is just used. This process is opposed to that with static libraries, which are effectively copied right into the executable file during the compilation and linking of your project.

Tip

On Linux systems, there are also libraries loaded at runtime just like DLLs on Windows systems. However, Linux folks call these *shared objects* (`.so`).

You can also use this technique for every component in your engine. You can have a DLL for the audio component featuring sound effects and music, you can encapsulate your input system in a DLL, and so on. Can you see where this path leads us? We will rebuild a system similar to DirectX itself, but in a high-level manner. Actually, this is ultimately what we do in the course of this book. You can also use other strategies, such as putting everything into one very big DLL.

Another advantage of using such DLLs is that you can build prototype implementations. The game programmers or application programmers can have a full working version of the engine ready to go in a short amount of time. This is not to speak of performance, however. This version is definitely not very good. However, it will work, so the other coders can do their job without waiting for you to get the engine on track. When the optimized version of the engine is done, you can hand over the new DLL and it will power the game or application code immediately without rebuilding the project or making changes to the code.

More About the Renderer

For the purpose of this book, I implement a DLL for the renderer using only Direct3D. In Chapters 9–11 I add libraries for audio, input, and network capabilities to the engine. If you feel more comfortable with OpenGL, you can provide your own implementation in your own DLL and run the engine using OpenGL for rendering. If you happen to use Linux, you can create a shared object using OpenGL and also implement the interfaces.

Caution

Actually, I'm not providing platform-independent code here. I want to keep the code as straightforward and simple as possible, so some interface functions use plain Windows structures. With a careful design, you can come up with an interface that will do the job on both systems.

Also of importance is the fact that the class that inherits from the interface and implements it cannot provide additional public member functions. Well, it can, but you must remember that the user of your engine does not see this derived class. The user sees only the functions declared in the interface because he or she only has a pointer of the interface type, even if that points to an object built from the derived class. There is no way for the user to access other functions of the object except those from the interface.

Workspace for the Implementations

Fire up the development environment of your choice, preferably Microsoft Visual C++, and get ready to hack some lines of code. Oh, but wait. A few other issues come to mind that you might find interesting. We spoke only about loading the DLL at runtime. There are two ways to do that. If you create a DLL using Visual C++, you will end up with the `.dll` file itself and an additional `.lib` file (a static library). The first way of loading a DLL is to link to this static library, while the second way is to load the DLL with a specific function call. The next section shows you how both ways work.

Loading DLLs

Huh? What is that static library doing there? Again, it is one way to load the DLL. However, it comes with the disadvantage that all applications that use your engine need to link to that static library. Thus, any time you make changes to the DLL code that also affect this `.lib` file, you also need to recompile the application's code using the DLL to relink the new static library. Such a change is, for example, changing the public interface of the DLL or any other exported function. That is not good, so we will use the second way to load a DLL without the accompanying static library. Windows provides a function you will see in action shortly that can be used to load DLLs. Here is its prototype:

```
HINSTANCE LoadLibrary( LPCTSTR plLibFileName );
```

This seems too easy, doesn't it? If you agree, you are right. It's never this easy. By linking to the static library, you tell Windows that you want that DLL automatically loaded at program startup, and you tell the linker about all the functions contained in the DLL—so that when you use them like regular functions in your program, it doesn't complain that it can't find them. Without this kind of list, you just don't know what is inside the DLL. If you use the Windows function to load the DLL at runtime without the static library, your application is not able to find the functions or classes inside the DLL.

This way of dynamically loading a DLL would be stupid if there weren't a way to get around the obstacles. Indeed there is a workaround. You have to declare the functions you want the application to access as `__declspec(dllexport)`. The application can then get a pointer to these functions and can them. There is another way. You can write a def file that lists the functions inside the DLL that should be exported. Then you have the list of exported functions in a single place to check, rather than distributed over several header files, for example. This is the way we will go, so more on this is coming soon.

Note

If you use the Windows function to load a DLL at runtime, you will not be able to use the functions inside the DLL. You first have to use another function called `GetProcAddress()` to get a pointer to a function you know the name of.

Exporting Classes from DLLs

There is one final catch. Who said it would be easy? The problem is that there is no easy way to export a class from a DLL if you load the DLL yourself and don't use the static library it came with. I can provide you with an easy solution. First, we will define a class derived from the interface and implement this class inside the DLL. Then we will write a function inside the DLL that creates an object of this class, casts this object to the interface type, and returns it to the caller. You might have already guessed that this function is exported from the DLL. In addition, there is no need to export anything other than this function from the DLL. The user knows about the interface and he can do everything with the returned object the interface allows him to do.

ZFXRenderer: A Static Library As Manager

Are you still with me? Now we will implement a static library called `ZFXRenderer`. The only purpose of this static library is to do the loading of our DLL. Please note that this is *not* the static library that the compiler will auto-generate for our DLL project. I implement this static library only to take the workload of loading the DLL from the user. You can still make changes inside the DLL without recompiling.

There is another job the static library needs to do. I will show you how to implement the renderer using Direct3D. However, to do an OpenGL implementation, you need to decide somewhere in the program which DLL to load. This task is also done by the `ZFXRenderer`, which you will see in action later in this chapter.

You will need to deal with that `ZFXRenderer` library only two times. First, you need it during initialization to create and obtain a render device. You see that I'm trying to adhere to DirectX naming conventions, so the thing actually sitting in my DLL and doing all that rendering of things is the `ZFXRenderDevice`. To finally make the puzzle pieces come together I will show that the `ZFXRenderDevice` is actually the abstract class we will use as the interface for our rendering implementation. The second time you need to deal with the `ZFXRenderer` is during the shutdown of the engine. You are required to delete that object in order to free its allocated resources.

Next, you need to learn how to set up a project in Visual C++. You also need to know which files to add and what kind of code to hack into those files to make this kind of system run. Don't panic if you feel a bit overwhelmed with the perhaps unfamiliar terms flying around in your head like slow motion bullets on weird paths. Well, I should not play Max Payne that often I guess, but I'm somehow connected to Finland, having lived and studied there for half a year.

Note

The description of how to set up the projects in this book is meant to explain how it's done using Visual C++ 6.0. If you use a newer version, check your software's manual. Also note that DirectX 9 no longer supports versions of Visual C++ or Visual Studio older than version 6.0.

Open Visual C++. In the File menu, select the New option. A dialog comes up; select the Project tab and select Win 32 Library (static) from the list of options. In the field on the right, enter the name for the project, which is ZFXRenderer. This confirms for Visual C++ that you are done with your settings. (For Visual C++ .NET 2003, select Project from the New option in the File menu. Chose Visual C++ Projects as Win32 Project, then click OK.

Now click on Application Settings, select Static Library, uncheck Precompiled Header, and finally click on Finish.)

You should now have an empty project space to work in. You can add files to that project in Visual C++ 6.0 by navigating to the Project menu and selecting the Add to Project option and the New suboption. In the dialog box that appears, select the File tab. In the list, you can select either C/C++ Header File or C/C++ Source Code File, depending on which kind of file you want to add. After that you can name the file that Visual Studio should create and add for you to the project space. (In Visual C++ 2003, select the Add New Item option from the File menu, select Visual C++, and select C++ File (.cpp) or Header File (.h). Then type the name of the file and click on Open.) Now, please add the following files:

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

The first two files will contain the implementation of the class `ZFXRenderer`. The third file is used to take the definition of the interface, that is, the abstract class that the implementation in our DLL will be derived from. Now I will show you how to integrate the abstract class into the workspace that is open on your screen.

Tip

You can find all the source code on the CD-ROM that comes with this book, so you don't necessarily have to write all the files from scratch and you don't even have to set up the workspace and projects. I think it's always better to know what is going on and how you can set up a workspace of your own.

ZFXD3D: A DLL As Render Device

Currently, you have a Visual C++ workspace that contains one project named ZFXRenderer. Most of you have worked with only one project in one workspace. It is time to change that. A workspace in Visual C++ can hold more than just one single project. Otherwise, there is no need to separate the project from the workspace. Add a second project to your workspace.

Tip

Since the advent of Visual Studio .NET (7.0 and higher), the workspaces were renamed with a fancier name. They are called *solutions* (`*.sln`) now.

Whenever you have two or more projects that are somehow related, it is a good idea to keep them all in the same workspace. Chances are that you will work on several of them in parallel. In one workspace, you can easily change between projects. Ask yourself how your projects will be related. It is pretty easy. You have one static library acting as a manager that is responsible for loading a DLL that implements the `ZFXRenderDevice` interface. In addition to the static library project, you have to add a project into the workspace for each implementation you will provide for the interface. There needs to be at least one such implementation, which I will show you in this chapter using Direct3D. If you want to implement the interface using OpenGL or a software renderer, you need to add another project to the workspace similar to the ZFXD3D project.

ZFXD3D is the name of the DLL project and also of the class inside this DLL project that implements the render interface `ZFXRenderDevice`. To add this project to the workspace, the steps necessary are quite similar to the ones for adding new components, such as files, to the workspace. However, for Visual C++ 6.0, instead of choosing files, select the Projects tab. From the list, select Win 32 Dynamic-Link Library. In the Name field, add a subdirectory called ZFXD3D and also use the same name for the project itself, such as ZFXD3D/ZFXD3D. Click the OK button and select that you want a plain, empty project. In Visual C++ 2003, select the Add Project / New Project option from the File menu, select Visual C++ Projects / Win32, type in the name (ZFXD3D in this case), and click OK. Then click on Application Settings, select DLL, check the Empty Project option, and click Finish.

Tip

Using the Active Project option in the Project menu, or from the Context menu, you can decide which of the projects inside the workspace is to be the active one. This is the one to be compiled when you hit the Build button and the one where all project-related settings and options are being used. If you want to add new files, for example, they can be added to the active project in Visual C++ 6.0. The active project is displayed in bold letters in the workspace treeview, so make sure that when you compile, add new files, and so on, you have the correct project set as active.

You should now have a second project sitting inside the workspace. This new project ZFXD3D is still missing files, but you will change that. You know how to add new files to a project, so take advantage of this knowledge and add the following files to the project:

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

I use the first four of these files to implement the class `ZFXD3D`. As discussed previously, this class implements the interface `ZFXRenderDevice`. We have not defined this interface, but we will come to that soon. The name `ZFXD3D` gives away the fact that we use Direct3D as the means of communication with the hardware. This class creates quite a bit of work so we just get started in this chapter. We will revisit the class in Chapter 5, "Materials, Textures, and Transparency." I personally don't like my files to get too long so I put the implementation of this class into several files to maintain structure in the project. All functions that are needed to initialize (and shut down for that matter) are defined in the file with the suffix `_init`. The enumeration of Direct3D needs a chapter of its own; I put everything related to that enumeration of available video and buffer modes into a separate file with the suffix `_enum`. This leaves only the file with the suffix `_main` for all the other things needed at runtime.

Finally, there is the mysterious file `ZFXD3D.def`. If you use a DLL, you need to tell the application which functions sit inside the DLL and which are meant to be called by an application. You can either export those functions directly by using `declspec(dllexport)` or you can use a def file containing the names of the exported functions. There is nothing magical about doing this.

Now that you have set up workspaces and added files to the project you can load a DLL that implements the render interface. We will discus this in the next section. After that, we will implement the ZFXD3D dynamic library project that implements the render device interface.

ZFXRenderDevice: An Abstract Class As Interface

Before you can start to implement anything, you first need to think about the interface. The interfaces of an engine need to be defined and fixed before anything else can take place. This is because everything in the engine is connected to these interfaces in some way. The programmers ordered to implement the interfaces naturally need to know the complete interface definition before they can get started. Those who write the programs that will use the interface to render something to the screen will also need the complete specification of the interfaces to see what they can or cannot perform—and more importantly, how to perform things with the interface.

All functions using functions from a particular API, such as Direct3D or OpenGL, must be hidden by interfaces. This way you end up with an engine that is not strictly dependent on a certain graphics library, and so not dependent on a certain operating system such as Windows or Linux. I already said that platform-independence is not that easy to achieve. There are always some things pestering you, starting with opening up a simple window of an operating system's specified data types. Therefore, as discussed previously, you must strive for platform-independence. The code in this book is graphics-API–independent, although it is still slightly bound to the operating system Windows.

The code in this chapter is fairly simple. I use this chapter only to demonstrate the idea of interfaces and API-independence, not to show off fancy rendering effects. We will get to the fancier code later.

You will not see a lot on the front end, but on the back end, the engine will do a lot more, such as detect underlying hardware and initialize Direct3D, providing a comfortable dialog where the user can select the settings he chooses. But now I have talked enough. Following is the definition of the interface:

```
// File: ZFXRenderDevice.h
#define MAX_3DHWND 8

class ZFXRenderDevice
   {
   protected:
      HWND       m_hWndMain;          // main window
      HWND       m_hWnd[MAX_3DHWND]; // render window(s)
      UINT       m_nNumhWnd;          // number of render-windows
      UINT       m_nActivehWnd;      // active window
      HINSTANCE  m_hDLL;              // DLL module
      DWORD      m_dwWidth;           // screen width
      DWORD      m_dwHeight;          // screen height
      bool       m_bWindowed;         // windowed mode?
      char       m_chAdapter[256];   // graphics adapter name
      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;
```

CH03_p.041-090.qxd 6/1/04 2:39 PM Page 52

```
      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;
```

Caution

Be warned: Do not show this code to hard-core C++ object-oriented programming gurus. These programmers would criticize this code because when you define an interface, you should not define attributes in the abstract class. This is a code design issue. In this book, I am less strict about design matters, saving the derived classes from having something close to a billion attributes.

If you are new to C++, you will notice the *virtual destructor* of the interface. The same reasoning that leads to virtual functions also leads to virtual destructors. No matter what pointer you use to point to an object, make sure that the correct destructor of the object is called by making the destructor virtual. Normally, every constructor is virtual unless you can guarantee that no one will ever cast a pointer to an object into something else. Believe me, you can't do that.

In addition to the constructor and destructor, there are several other functions that are defined as purely virtual. You will see the meaning of each function when we implement the functions, but I think the names are already hint at what the functions are meant to do later on. From a graphical point of view there is not much to do yet. You can use the method `ZFXRenderDevice::SetClearColor` to change the screen color and clear the screen explicitly using the appropriate function. Clearing the color is already implicitly contained in the `ZFXRenderDevice::BeginRendering` call. But don't worry if you think there is not much we can use this interface for. For now you are right, but later you will be able to use most of Direct3D's functionality through this interface. In addition, you will have learned how to add other things not explicitly covered in this book.

Implementing the Static Library

The anchor for our work with a DLL is the static library `ZFXRenderer`. Its task is to decide which DLL to load. The DLL in turn represents a render device that can be used to output graphics on the screen. The implementation of this static library is quite short and will not change over the course of this book. You need to recompile it only in one of two cases. The first case is if you get into a situation where you indeed need to change something in the implementation. This will happen, for example, in a situation where you add another DLL to the workspace and want the static library to load this new DLL. The second case where you need to recompile the static library is when you are making changes to the `ZFXRenderDevice` interface definition, because the static library uses its header and handles pointers of the interface type.

That made clear, I show you now the header file for the static library's only class. It is, not surprisingly, called `ZFXRenderer`.

```
// file: 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;
```

This is the whole class that will not change anymore, correct? Well, yes, as I told you, this class is short and easy to implement. The constructor simply takes the Windows instance handle from the application using the ZFXEngine and stores it in one of its attributes. The class will use it later on. Even more important is the attribute `m_hDLL`, which will receive the handle from Windows for the loaded DLL. The third attribute `m_pDevice` is the most important one around. It is a pointer to an object that implements the `ZFXRenderDevice` interface. That object will be created using the `ZFXRenderer::CreateDevice` method. This method takes a string as the input parameter specifying the DLL to be loaded.

You will be even more disappointed when you look at the constructor and destructor of this class. They are quite short, as there is not really much to do:

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

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

Okay, before we move on to the more interesting stuff of this class, namely, the creation of the interface-implementing object, we need to take a look at some other functions that are also contained in the header file `ZFXRenderDevice.h` but that are not part of the class itself:

```
// file: 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);
   }
```

Here, I define the symbols `CREATERENDERDEVICE` and `RELEASERENDERDEVICE` for pointers to the given functions `CreateRenderDevice()` and `ReleaseRenderDevice()`. Those functions are declared as `extern` to indicate that we do not implement them here. This must be done in another part of the source code, but it is stated here that we need to work with those functions in our static library. As discussed previously, there is no direct way to export a class straight from a DLL without knowing its declaration. However, you can use the functions you see in the previous code to get a pointer of the interface type to an object implementing the interface. These functions will be implemented in the DLL in a moment. This static library knows only that those functions are somewhere around to be used and it uses them.

Oh, and the `"C"` means that the functions should be exported in plain C style without C++ name mangling. The overhead of object orientation forces C++ to twist around the function names and parameter lists a bit, but we don't want to worry about that and so we enforce plain C usage of those functions here.

Loading DLLs

The following function is responsible for creating an object that implements the interface. It takes a string as a parameter that specifies the name identifying the DLL to be used. To keep things simple here, I included one possible option for this string, namely `"Direct3D"`. This will identify that the caller wants to load the implementation of the interface that features Direct3D. Any other string will result in an error message thrown out by the function, as you can see for yourself in the following code:

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

   if (strcmp(chAPI, "Direct3D") == 0)
      {
      m_hDLL = LoadLibrary ("ZFXD3D.dll");
      if (!m_hDLL)
         {
         MessageBox(NULL,
            "Loading ZFXD3D.dll failed.",
            "ZFXEngine - error", MB_OK | MB_ICONERROR);
         return E_FAIL;
         }
      }
   else
      {
      _snprintf(buffer, 300, "API '%s' not supported.", chAPI);
      MessageBox(NULL, buffer, "ZFXEngine - error",
                 MB_OK | MB_ICONERROR);
      return E_FAIL;
      }

   CREATERENDERDEVICE _CreateRenderDevice = 0;
   HRESULT hr;

   // pointer to DLL function 'CreateRenderDevice'
   _CreateRenderDevice = (CREATERENDERDEVICE)
                            GetProcAddress(m_hDLL,
                                   "CreateRenderDevice");
   if ( !_CreateRenderDevice ) return E_FAIL;

   // call DLL function to create the device
   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
```

If the function gets a string that specifies a DLL that it recognizes, then the function will do all the things necessary to load the DLL and create an object of the interface implementation. If the caller uses the Direct3D API, the function will load the `ZFXD3D.dll`. You will see this library's implementation shortly. The function uses the following Windows API function to load a DLL at runtime:

```
HINSTANCE LoadLibrary( LPCTSTR lpLibFileName );
```

For the single parameter of this thing, pass in the name of the DLL you want to load. Now for the guts of this function. It will ensure that the DLL is loaded into memory only once. If another application is already using the DLL, the function ensures that the memory is mapped in a way that this application will also have access to the DLL. This is another advantage of using DLLs. They are loaded only one time when they are needed.

Now the return value gets interesting. If the call succeeds, you will get an instance handle from Windows. Whenever you need to call Windows API functions inside the dynamic link library that are requesting an instance handle as parameter, you must not use the application's instance handle but the handle of the library itself—that is, the return value of this function.

Sniffing for Exported Functions inside DLLs

Let's suppose now that the call was successful and we loaded the DLL. Now we want an object from the class `ZFXD3D` inside the DLL, but you can't see this class inside the DLL. For this reason, I define the external function `CreateRenderDevice()`, which must be implemented in the DLL. The problem now is to catch up with this function implementation. Unlike static libraries, you don't know the address of a function inside a DLL at compile time so you cannot just call the function. The linker would not be able to find it because it does not have the compiled version of the DLL at hand.

There is a way around this potential disaster. You can just sniff inside a DLL and seek the address of any exported function at runtime. This is done using the following Windows API function:

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

You have to pass in the handle of the loaded DLL as the first parameter to this function. Don't get freaked by Microsoft messing around with different handle names, as they are all more or less the same type. The `HMODULE` handle you need to use here is the same handle returned by the `LoadLibrary()` function, even if it was called `HINSTANCE` when it was returned.

You can see how this all is connected. You can use this function to get the address of the `CreateRenderDevice()` inside the DLL and store this address in the pointer called `_CreateRenderDevice`. Then you can call the function, and if everything goes well, you have a valid object of the class `ZFXD3D` from the DLL stored in the attribute `m_pDevice`. Please note that this attribute is the interface type `ZFXRenderDevice`. The static library does not need to know anything about the class `ZFXD3D` at all.

That's how interfaces work. Neat, isn't it? To make this clearer, I will now show you the `ZFXRenderer::Release` method. It does the very same thing, except that it sniffs for and calls the `ReleaseRenderDevice()` function from the DLL.

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

   if (m_hDLL)
      {
      // pointer to dll function '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;
         }
```

CH03_p.041-090.qxd 6/1/04 2:39 PM Page 58

```
      }
   } // Release
```

As you will see in a moment, the object that implements the interface is created inside the DLL. Therefore, I let the DLL do the release job of this object as well because the object's memory has been allocated on the DLL's heap. To do this, you have to get the pointer to the release function that is exported. If you have that pointer, call the function and hand to it the object that should be deleted. Again, there's no magic behind this.

Maybe the definitions `CREATERENDERDEVICE` and `RELEASERENDERDEVICE` are a bit confusing for you, but they actually tell the compiler which parameter list belongs to that function pointer. Without that, the compiler would not be able to verify if you called the function in a way that doesn't blow up the call stack.

Now for the good news. I'm glad you made it so far. As far as I can see, we just suffered slight losses in taking that hill here. Good job, Soldier. You're done with the static library loading a DLL now. The bad news? Well, there is another hill to be taken tonight. Now, move it!

Implementing the DLL

Change the active project to the ZFXD3D project. I'll show you now how to encapsulate Direct3D inside this project. The benefit of the encapsulation is that anyone using our engine does not need to know a single piece of information about Direct3D. He does not need to have the DirectX SDK (*Software Development Kit*) installed and there is no need to mess around with Direct3D data types or classes.

Note the attribute `m_pChain[MAX_3DHWND]` in the following class definition. Even if the engine we implement in this book is a small demo project, I want to make it as complete as possible. One of the frequently asked questions in bulletin boards about game programming goes like this: "I want to render graphics into multiple child views for my editor. How do I do that?"

There are a number of ways to achieve this goal using DirectX, but the way it's meant to be done is to use the so-called Direct3D swap chains, which are basically used to take one rendered image each. I will go into more detail about that later. For now, note that our engine supports up to `MAX_3DHWND` different child windows that you can render your graphics into. The handles to those different windows are stored in the attribute `m_pChain`.

```
// file: ZFXD3D.h

#define MAX_3DHWND 8

class ZFXD3D : public ZFXRenderDevice
   {
```

```
public:
   ZFXD3D(HINSTANCE hDLL);
   ~ZFXD3D(void);

   // initialization
   HRESULT Init(HWND, const HWND*, int, int, int,
                bool);

   BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);

   // interface functions
   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;

   // start the API
   HRESULT Go(void);

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

As you can see, there are eight public functions in this class in addition to the constructor and destructor. Seven of these functions need to be there as they stem from the interface that this class inherits from. The other function is called `ZFXD3D::DlgProc`. Although this function is public, don't forget that the class `ZFXD3D` itself cannot be seen from outside the DLL so the public attribute resolves effectively to "can be seen by all objects inside the DLL." This function is needed to process the messages connected to the dialog box that the engine uses to let the user select some settings for the display modes and stuff.

Note

Member functions in derived classes that override virtual member functions from the parent class(es) are automatically tagged as virtual by the compiler. There is no way around being virtual for those functions even if you don't use the keyword `virtual` explicitly. Naturally, it is advisable to use the keyword anyway so one can see at first glance that those functions are virtual.

Enumeration from the DirectX SDK

Several attributes in the `ZFXD3D` class store the most important Direct3D objects and information about the graphics adapter capabilities or active settings. You need some functions to get the graphics adapter to reveal its deepest secrets, namely, its capabilities. This is all part of the enumeration of available graphics adapters and their modes. In this implementation, I use a class `ZFXD3DEnum` to do this job, and its methods are quite similar to the stuff provided by the DirectX SDK common files. I don't want to bore you by explaining the DirectX framework to you.

If you are unsure about those enumeration things and want to learn them first, feel free to dig through the common files and see how enumeration is done there. My implementation is nothing more than a boiled-down version of it. So you can take a look at my class instead, because I think it is easier to grasp. You can find the class implementation on the CD-ROM accompanying this book.

What Is Coming Next?

Before we get started implementing the DLL's `ZFXD3D` class, I want to make sure that you know what is coming next. The interface demands that we provide an `Init()` function to crank up the graphics API our DLL wants to use. It is advisable to make this powering-up process as flexible as possible and use as little hard coding as possible. One thing people like to do to achieve that flexibility is write an init file containing all the values for screen width and height and things and so on. During powering up, the engine parses the init file. But I want to do it another way. I will show you how to implement a Windows dialog box that pops up during initialization and lets the user select some settings to influence these values in the most flexible way.

Still, our engine is very easy to work with. You only need to call one initialization function and that will kick off an internal process that includes enumeration and the dialog box to let the user decide which settings should be used (such as fullscreen or windowed, and so on). If you take a look at the DirectX SDK samples, you see that you can switch all those settings at runtime. This comes along with some overhead of resetting the Direct3D device, so I'm not going to include this feature in the ZFXEngine. I want to keep the code as simple to follow as possible. You can always refer to the DirectX SDK common files to see how to switch at runtime.

Okay, but before we can start implementing our class I want to show you the implementation of the exported functions. These are the functions the static library sniffs for via `GetProAddress()`.

Exported Functions

You still remember the empty files we created for the ZFXD3D project, don't you? If you do, then you also remember that mysterious `ZFXD3D.def` file, which is somehow related to exporting functions from a DLL. I already talked about exporting functions from a DLL and how you can do it. The following "code" is what you need to write into the def file to export the functions the static library `ZFXRenderer` will sniff for:

```
; file: ZFXD3D.def
; DLL exports the following functions
LIBRARY "ZFXD3D.dll"
EXPORTS
   CreateRenderDevice
   ReleaseRenderDevice
; end of file :-)
```

As you might have already guessed, the semicolon defines a comment in a def file. Then you need to write the name of the library after the keyword `LIBRARY` so that the file can be identified as belonging to the named DLL and to define exports for that library. After that you use the keyword `EXPORTS` to name your wishes, that is, to name the symbols that should be exported from the DLL and that are therefore visible and accessible to other applications using this DLL. Here you only have to define the two functions we already talked about by their names. The parameter lists for the functions are not needed when you export them.

Caution

If you are using Visual C++ or Visual Studio .NET (version 7.0 or newer), you will sometimes get into trouble with the def files. The trouble is that they don't work and as a result of this, the functions are not exported. The `GetProcAddress()` function then returns a `NULL` pointer because it cannot find what you want it to look for.

When this happens, you need to remove the function declarations `CreateRenderDevice()` and `ReleaseRenderDevice()` from the file `ZFXRenderDevice.h`, but the two `typedef` instructions must remain in place there. Now move the two declarations into a header file inside the DLL project ZFXD3D and add the following prefix to the declarations and function definitions: `extern "C" __declspec(dllexport)`

Now these two functions are marked as exports inside the DLL and will be found by the `GetProcAddress()` call.

Now get ready to see how cute these two exported functions really are. There is not much behind them other than creating an object or deleting it. But please note that though the object created is from the class ZFXD3D, it is returned as a reference with a pointer of the interface type ZFXRenderDevice:

```
// file: 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;
   }
```

If you want to create such an object, you need to know the handle to the loaded DLL, as this is used by the constructor of the ZFXD3D class. You need this handle to open up a dialog from inside the DLL, for example, or to access any other kind of resource stored in the DLL, such as icons. The release of the object is just a plain delete call to the object, which forces its destructor to go to work and then free all memory used for the object itself. The interesting thing here is that you don't need to cast the interface type pointer to a ZFXD3D pointer. Because of the virtual destructor, the delete call correctly calls the ZFXD3D class's constructor, as this is the class the object originally stems from. There are times when object-oriented programming makes life easier, as you might know.

To recap, you made the class ZFXD3D available from outside the DLL without the need for the user to know this class at all. If you just know the interface definition, you can use the render device object.

Comfort By Dialogs

To keep the engine as flexible as possible to demonstrate what you can do to satisfy your customers, I'll now show you how to use a dialog to make settings during the startup phase of the engine. Then I'll talk about that nasty enumeration stuff Direct3D comes along with, but hey, OpenGL has its extensions, which you need to verify before using them.

Assume you are done with an enumeration, so you have a whole list of modes and capabilities the current hardware provides. So what do you do with it? Should you as engine programmer select what you think is the best possible option for the user? Definitely do not do this. Instead, use a container that displays the most important findings of the enumeration process. This container is a simple dialog in which the user can choose the settings he prefers.

Creating the Dialog

You need to activate the `ZFXD3D` project now. Go to the Insert menu and select Resource. From the upcoming dialog, select Dialog, and then select the New button. You just created a new resource object and Visual C++ changes to its resource editor automatically.

By double-clicking on a control element (button, drop down list, and so on) or on the dialog box itself, you make the Attributes dialog box for this item show up. The most important field here is the one named ID. Here you need to provide a unique identifier. Try it out and identify the dialog box as `"dlgChangeDevice"`. It is very important here to include the quotation marks in the ID field as well. Otherwise, you have to resolve the ID to the actual name of the dialog if you want to show it.

You are probably familiar with creating custom dialogs in the resource editor. If not, play around with it for a few minutes. It's not complicated. Then insert the controls from the list below into the dialog. Give them the IDs shown in the list. This time, don't use quotation marks because for the controls of a dialog it is much easier to use a macro to resolve a plain ID that the resource editor can connect to. This is a plain number as opposed to a real string.

- Combo box named `IDC_ADAPTER`
- Combo box named `IDC_MODE`
- Combo box named `IDC_ADAPTERFMT`
- Combo box named `IDC_BACKFMT`
- Combo box named `IDC_DEVICE`
- Radio button named `IDC_FULL`
- Radio button named `IDC_WND`
- Buttons named `IDOK` and `IDCANCEL`

The combo box controls are used to offer the available graphics adapters and their video modes for selection to the user. Most adapters are capable of using several video modes or screen resolutions (800x600 or 1024x768, for example). Then there are two combo boxes for the color format because, since Direct3D 9, it is possible to use different formats for the front buffer and the back buffer if the program is running in windowed mode. You are not bound to the current desktop format the user is running.

The last combo box is used for the type of the Direct3D device. There are only two types available. First is the HAL device, the Hardware Abstraction Layer. This is the graphics adapter. The other one is the REF, the Reference Rasterizer. I hope you already know that Direct3D can emulate almost everything in software if the hardware is not able to do it. The REF enables you to run most of the Direct3D features on ancient hardware that does not even provide hardware transformation and lighting.

Caution

Using the REF is not recommended, except for testing scenarios in which you don't have the hardware available to do the features you want to test. It is incredibly slow and not meant for real-time applications. Even the poorest hardware can beat a software implementation on features it actually provides. Furthermore, the REF device isn't even guaranteed to be available in a non-debug install of DirectX.

The two radio buttons in the dialog can be used to select whether to start the engine in windowed or fullscreen mode. Figure 3.1 shows how the dialog box will look in the running engine program later on.

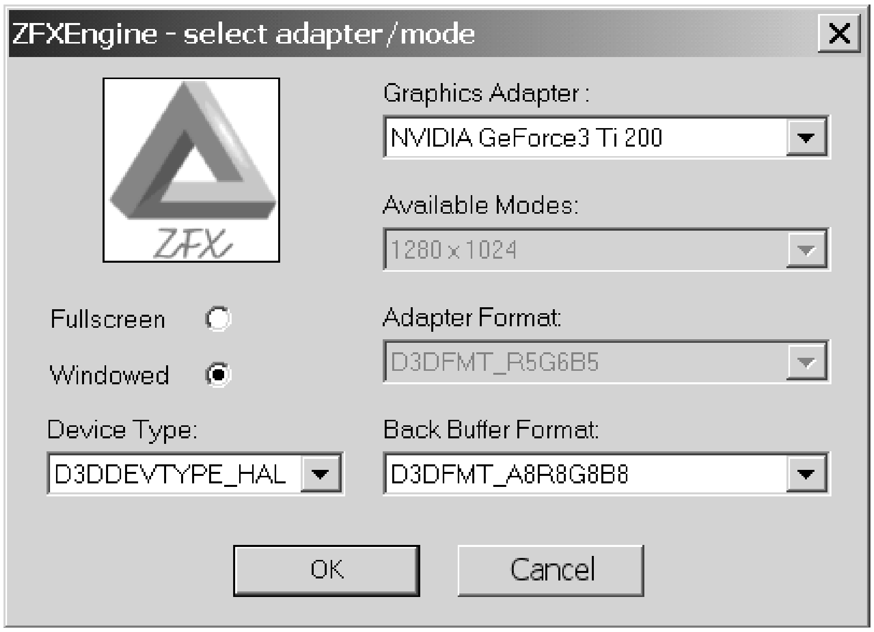
If you are still with me and have followed all these steps, select the Save button in Visual C++ to save the project. Note that a dialog box asks you to save your resource first. Save the list, and then select the `ZFXD3D` directory; name the file `dlgChangeDevice.rc`. This autogenerates a file called `resource.h`, which contains some ID stuff and definitions in a script form. Visual C++ needs this to build the dialog when you compile the project.

Add the two files I just mentioned to the project into a resource directory. This enables you to use the dialog you just created. The resource header must be included in every file where you want to use something from the dialog. Mostly this is when you want to access the control elements in the dialogs and need their IDs for Windows function calls.

But first, let's review dialog boxes and how to call them from a program. A dialog box in Windows is basically the same kind of object as a window. The same is true for the control elements that sit inside the dialog. So if you want to get messages from these controls or send messages to them, you need to have a callback procedure for the dialog like you would have for a simple window.

The following Windows API function lets you show a dialog box and name the callback procedure that should handle the messages coming from the open dialog:

CH03_p.041-090.qxd 6/1/04 2:39 PM Page 65


ZFXEngine - select adapter/mode

Graphics Adapter : NVIDIA GeForce3 Ti 200

Available Modes: 1280 x 1024

Fullscreen ○

Windowed ●

Adapter Format: D3DFMT_R5G6B5

Device Type: D3DDEVTYPE_HAL

Back Buffer Format: D3DFMT_A8R8G8B8

Figure 3.1 The selection dialog as it will appear in the final program version

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

The first parameter has to be the instance handle of the module containing the resource. As you work inside a DLL, it needs to be the DLL's handle. You remember the value we retrieved as return value from the `LoadLibraryEx()` call? And the dialog here is the only reason we need to know this handle in this project. This is why the constructor of the `ZFXD3D` class asks for it. Next, you need to understand the meaning of the parameters.

The second parameter must be the ID of a template used for the dialog. This is the dialog box you designed using the resource editor. If you had given it a plain ID you would need to resolve this now by using the `MAKEINTRESOURCE()` macro. However, you were smart enough to use a real string as the ID so you can enter that string here to load the dialog. The next parameter is the handle of the parent window. You see, the dialog is just a window, a child to its dad.

The last parameter is the name of the callback procedure that should handle the window's messages. This is a problem because object-oriented programmers want a member function as a callback routine. That is not possible because the existence of the callback function must be ensured at compile time, but a normal member function's existence is bound to the existence of an object of the according class. Now you could use a static member function. This would be okay, but it comes at the price of not having access to member variables of the class.

Message Procedure for the Dialog

The thing is, you want this kind of access. In this instance, you want to read the user's selections right into some member variables of the class. I will return to this problem in a moment, but first I want to show you what the callback function should look like:

```
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;

   // get 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)
     {
      // preselect windowed mode
      case WM_INITDIALOG:
         {
         SendMessage(hWND, BM_SETCHECK, BST_CHECKED, 0);
         m_pEnum->Enum(hADAPTER, hMODE, hDEVICE,
                       hADAPTERFMT, hBACKFMT,
                       hWND, hFULL, m_pLog);

         return TRUE;
         }

      // render logo ( g_hBMP is initialized in Init() )
      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;

  // a control reports a message
  case WM_COMMAND:
     {
     switch (LOWORD(wParam))
        {
        // OK button
        case IDOK:
           {
           m_bWindowed = SendMessage(hFULL,
                            BM_GETCHECK, 0, 0) != BST CHECKED;
           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;
}
```

On initialization of the dialog, there are already some actions that need to take place. The most important is that you have to call the `ZFXD3DEnum::Enum` function. This kicks off all those enumeration things I talked about earlier, which you should know from the DirectX SDK common files. As parameters, the function wants to get the handles to all the controls of the dialog to fill them with the findings of the enumerations. This way an instance of the `ZFXD3DEnum` class always has access to the control element of the dialog and can read or write the dialog's contents. When this call is done, the control elements of the dialog, especially the combo box lists, are filled with entries about the hardware capabilities.

The next part of the procedure renders a logo image into the dialog box. This is just to show off. You need only basic functionalities from the Windows API. If this is not the case, please refer to the MSDN Library (*Microsoft Developer Network*) for the functions used in this case. To make this work, you need to link the library `vfw32.lib` (video for Windows) and include its header `vfw.h`. Note that the bitmap file with the logo is already loaded in the function `ZFXD3D::Init`.

Finally, the callback procedure handles the message `WM_COMMAND`. This message is sent by Windows to the dialog when an event occurs from one of the control elements. You can now check the lower part of the `WPARAM` parameter, which contains the plain ID of the control that is involved. Windows provides the macro `LOWORD` to evaluate only the lower parts of a given parameter.

You can see that the procedure calls one of the following functions according to the event that happened. If the user changes the selection in the Device combo box, for example, then the function `ChangedDevice()` from the class `ZFXD3DEnum` is called.

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

All these functions have the same purpose. If the user changes one selection, it might influence the possible options for the other settings. The HAL might not have as many possible formats as the REF, for example. The `ZFXD3DEnum` class then walks through its enumerated lists and fills the dialog combo boxes with the possible choices available for the setting the user just selected. This is fairly simple.

Shutting Down the Dialog

The only two controls that are handled a bit differently are the OK button and the Cancel button. The following Windows API function makes the dialog disappear from the screen and your mind by destroying it.

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

Again you see that the dialog is just a window because for the first parameter you have to hand over the handle of the dialog you want to make quit. The second parameter is more interesting because it allows you to control a crucial return value we have not yet talked about. This return value is given to you by the `DialogBox()` function. Does this sound strange? It is not. You call `DialogBox()` somewhere inside your application. This brings up the dialog box and makes its callback procedure active, as long as you call `EndDialog()` from somewhere. As this dialog is modal, which means that the application is stalled as long as the dialog and its callback procedure are active, this call normally takes place in the callback procedure. From here, you can define what value should be returned to the point where the "normal" application waits for the dialog to end. This is the place where it all began: the `DialogBox()` call itself.

Now you can see that the dialog returns `0` in the case that the user canceled the dialog and `1` in the case that the user hit the OK button. It is also important to know that `DialogBox()`returns a value of `-1` if the call itself fails because of a missing dialog resource or something similar. So you'd better not interfere with negative return values via the `EndDialog()` call.

CH03_p.041-090.qxd 6/1/04 2:39 PM Page 70

But back to the OK Button case. You have to call the ZFXD3DEnum::GetSelections function there to make the class save the current settings from the dialog to global variables or ZFXD3D class variables. The structure ZFXDEVICEINFO contains a value from the enumeration that describes the device capabilities that should be used to initialize the Direct3D device, such as its modes and with which adapter it belongs. Again this structure is very similar to the one used in the DirectX SDK common files.

End the dialog and return 1 to signal a successful termination of the dialog. You can now look at how the initialization of the engine is done and how it uses the dialog. You still remember the problem of using callback functions in a class, don't you?

Initialization, Enumeration, and Shutdown

Everything starts with the constructor of the class ZFXD3D that is called from the exported function CreateRenderDevice(), which in turn is called by the static library class ZFXRenderer if the user wants to initialize the render device. So here is the constructor:

```
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;
   }
```

The only interesting thing in this constructor is the global variable g_ZFXD3D. As you can see, we set it to the this pointer. Given that the user creates only one object of this class we now have a global pointer to that object. I agree that is not a smart way to build something like a singleton, but then it does its job. If you look at the DirectX SDK common files, you will find a big software company from Redmond does the very same thing, so I guess you and I can live with that quick 'n dirty solution in this book.

Have a quick look at the destructor and then follow me to further explanations about global variables and callback procedures.

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

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

Initializations Process Chain

Using the render device interface, you can call the following function to initialize the render device. (This function is short, which means you will need more functions to write and call.) But first dig your way through it.

```
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;

   // should I use child windows??
if (nNumhWnd > 0)
  {
   if (nNumhWnd > MAX_3DHWND) nNumhWnd = MAX_3DHWND;
   memcpy(&m_hWnd[0], hWnd3D, sizeof(HWND)*nNumhWnd);
   m_nNumhWnd = nNumhWnd;
   }
// else use main window handle
else
  {
   m_hWnd[0] = hWnd;
   m_nNumhWnd = 0;
   }
m_hWndMain = hWnd;;

if (nMinStencil > 0) m_bStencil = true;

// generate enum object
m_pEnum = new ZFXD3DEnum(nMinDepth, nMinStencil);

// load ZFX logo
g_hBMP = (HBITMAP)LoadImage(NULL, "zfx.bmp",
                             IMAGE_BITMAP,0,0,
                             LR_LOADFROMFILE |
                             LR_CREATEDIBSECTION);

// open up dialog
nResult = DialogBox(m_hDLL, "dlgChangeDevice", hWnd,
                    DlgProcWrap);

// free resources
if (g_hBMP) DeleteObject(g_hBMP);

// error in dialog
if (nResult == -1)
   return ZFX_FAIL;
// dialog canceled by user
else if (nResult == 0)
   return ZFX_CANCELED;
// dialog ok
else
   return Go();
}
```

There are several parameters to this function, so let's take them one by one. The first parameter is the handle of the application's main window. Easy. The second parameter is an array of handles with the number of entries following in the third parameter. You only need those if you don't want to render into the main application window and use one or more child windows instead. The function does nothing more than copy the data it needs from the parameters to the member variables.

The next two parameters specify the minimum bit depth you want to use for the depth buffer and the stencil buffer. The last parameter specifies whether the caller wants a secure log file that streams each entry at once, thereby making this secure even if the application crashes.

Okay, now back to the callback problem. The function quickly loads the bitmap file used for the logo and creates a `ZFXD3DEnum` object. Then it calls the dialog box. But instead of providing our `ZFXD3D::DlgProc` defined previously, which wouldn't work for reasons stated, it uses a callback procedure named `DlgProcWrap()`. This is a plain C function and looks like this:

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

Again, this is not pretty, but it is a quick (and dirty) workaround to use the callback function from the `ZFXD3D` class. This callback procedure delegates the call and that is why the global object points to the last (and only) created instance of the `ZFXD3D` class.

Now if everything goes right, you will have filled some global and member variables with the settings based on the user's selections from the dialog. The only thing remaining is to crank up the Direct3D device (finally!). As this is a quite lengthy process, I provided its own function for it. It is called `ZFXD3D::Go` and it looks like this:

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

```
// create Direct3D main object
if (m_pD3D)
   {
   m_pD3D->Release();
   m_pD3D = NULL;
   }
m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );

if (!m_pD3D) return ZFX_CREATEAPI;

// get fitting 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;
      }
   }

// fill in present parameters structure
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;

// stencil buffer active?
if (  (xCombo.fmtDepthStencil == D3DFMT_D24S8)
   || (xCombo.fmtDepthStencil == D3DFMT_D24X4S4)
   || (xCombo.fmtDepthStencil == D3DFMT_D15S1) )
   m_bStencil = true;
```

```
else
   m_bStencil = false;

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

// create direct3d device
hr = m_pD3D->CreateDevice(g_xDevice.nAdapter,
                          g_xDevice.d3dDevType,
                          m_hWnd, xCombo.dwBehavior,
                          &m_d3dpp, &m_pDevice);

// create swap chains if needed
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]);
      }
   }

delete m_pEnum;
m_pEnum = NULL;

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

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

In this function, you don't need to do anything other than run through the objects of the `ZFXCOMBOINFO` structures and find the one that fits the user's wishes. From this object, you can then retrieve the values you need to fill in the present parameters structure from Direct3D, and you can initialize the Direct3D device—finally!

There are some minor issues involved in running in either windowed or fullscreen mode. After that, the device is ready to go and ready to pump polygons to the graphics adapter. Please note that the name `combo` has nothing to do with a combo box control. This name was introduced in the DirectX 9 common files, and it describes a combination of front buffer and back buffer format, an adapter, a device type, a vertex processing type, and the depth-stencil format. I want to show you what is happening with the common files.

Caution

In this current design, you force the users of the engine to accept the dialog that pops up each time an end user runs his application. This might be useful for applications such as games that can run either windowed or fullscreen. But it is nonsense for applications meant to run in windowed mode only, such as tools and editors. Therefore, I added a method `ZFXRenderDevice::InitWindowed` as an alternative. This method cranks up the engine in windowed mode without querying the user for his wishes.

About Swap Chains

Let's reveal the magic of how you can render to multiple child windows using Direct3D. It's not that difficult. The Direct3D device uses so-called *swap chains* for this purpose. You say you've never heard of these? If you ever used Direct3D before, then you have actually used a swap chain.

Basically, a swap chain is nothing more than one or more back buffers. The standard Direct3D device has at least one built-in render target—the front buffer. If you use a back buffer, then you suddenly end up with two buffers you can render to that are swapped automatically if you call the `IDirect3DDevice::Present` function. There it is, the implicitly existing swap chain of the Direct3D device. The device simply cannot exist without a swap chain.

Now you can add as many swap chains as you like if you have enough memory. In the previous listing, you saw the function that can do exactly this, creating a swap chain and adding it to the device: `IDirect3DDevice::CreateAdditionalSwapChain`. As parameters you need to provide only the Direct3D present parameters structure and a pointer of the interface type `IDirect3DSwapChain9`.

I don’t want to go into detailed descriptions here of how to use Direct3D interfaces, as it is not the topic of this book. However, I will point out all the functions you need to deal with those swap chains and let you explore in the DirectX SDK reference whatever you cannot grasp from my explanations. What I show you here is more than enough to get a grip on those swap chains.

After creating a new swap chain, which is connected to another window over the window handle you put into the present parameters structure, you can later change the active swap chain if you want to use another child window to render into. Next, I show you how to do this.

Changing between Child Views

Let’s suppose the user provided an array of window handles to the `ZFXRenderDevice::Init` call because he wants to use multiple child views to render into. You have to make your engine capable of switching to any of those child windows whenever the user wants you to. To do this is actually quite easy, but there are some nasty details connected to it. Each time you render something to Direct3D, the pixel output is rendered onto what is called a render target. The back buffer is such a render target, but a texture could be a render target as well. I told you that a swap chain is basically nothing more than a separate back buffer, so the only thing you need to do is change the render target to a swap chain’s back buffer if you want to render to another child window.

And that sounds even more complicated than it is. You have to pick the swap chain object and retrieve its back buffer object as a Direct3D surface object. The following function lets you do this: `IDirect3DSwapChain9::GetBackBuffer`. It awaits three parameters. First, it waits for the index of the back buffer, as there could be more than one. Second, it waits for the flag `D3DBACKBUFFER_TYPE_MONO`. In a later version of DirectX, there might be a support for stereo rendering, but for now there is only the flag I mentioned. The third parameter is a pointer to a Direct3D surface to be filled with the back buffer you requested.

After you get what you came for, you can set this surface as a render target for the Direct3D device. *Voilá*, you successfully changed the active swap chain of the device to another one. This means that until the next switch, Direct3D will render into the window connected with the swap chain that is now active.

It is that simple. The following function takes care of doing this job:

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

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

```
      return ZFX_FAIL;

   // try to get the appropriate back buffer
   if (FAILED(m_pChain[nHwnd]->GetBackBuffer(0,
                                             D3DBACKBUFFER_TYPE_MONO,
                                             &pBack)))
      return ZFX_FAIL;

   // and activate it for the device
   m_pDevice->SetRenderTarget(0, pBack);
   pBack->Release();
   m_nActivehWnd = nHwnd;
   return ZFX_OK;
   }
```

Caution

Things are never easy. There is a slight catch concerning the change of a render target. The depth-stencil surface attached to the Direct3D device must not be smaller than the dimensions of the render target. Otherwise the whole screen will look pretty messed up. To keep things simple, I use the desktop size for the implicit Direct3D swap chain, and it will also be taken for the depth-stencil surface. You can also create a depth-stencil surface for each render target and change the depth-stencil surface if you change the render target.

We have completed the initialization stuff. Our engine is now able to initialize Direct3D in a comfortable way. You can also fire up a lot of child windows you might want to render into, and you can switch the active window in the blink of an eye.

Figure 3.2 shows you what the final demo in this chapter looks like. This is nothing spectacular at all. There is, however, a lot going on behind the scenes, as you learned in this chapter. The figure shows the engine running in windowed mode with four child windows. We are not yet able to render anything but we can already clear the client area of all the child windows, as you will see in a moment.

If you select fullscreen mode in the dialog during the startup phase of the engine, the engine is smart enough to ignore the list of child windows and use the main application window handle instead. It will also block the calls to `ZFXD3D::UseWindow` without an error. This way you cannot mess up the engine by making a wrong call. This intelligent logic sits inside the `ZFXD3D::Init` and `ZFXD3D::UseWindow` functions, as you might have noticed.

There are still some pieces missing from the puzzle. We have not yet implemented the last few functions the `ZFXRenderDevice` interface wants us to implement. These are the topic of the next section.

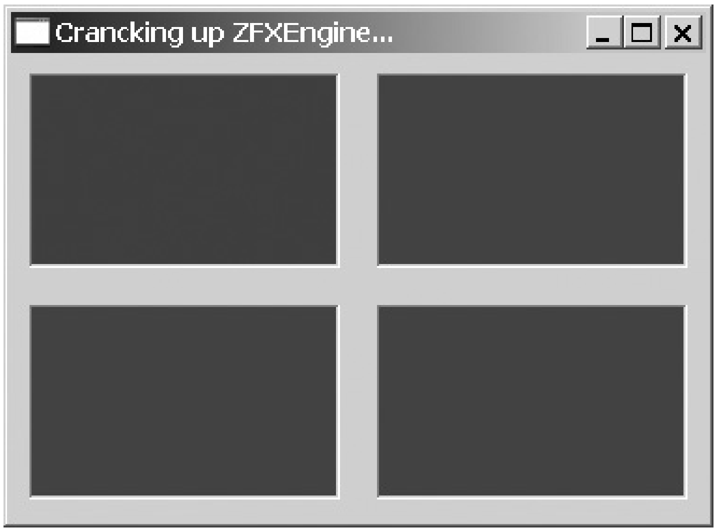


Figure 3.2 The engine running in windowed mode featuring four child windows

Render Functions

We did it. We initialized the render device that sits inside the DLL, and we used Direct3D. We can also release it and just clean up after our party. This does not take us anywhere if we cannot do anything at all between its initialization and its release. However, the main purpose of a render device should be to render something.

I guess you already have a lot to consider from this chapter and I don't want to throw a lot more on you. You will soon see that rendering is not very easy if you want to do it right and encapsulate it comfortably in an engine.

However, I can't just let you initialize and destroy render devices without seeing anything on your screen. Therefore, I will include the basic functionality to clear the screen using the render device. Take a look at the following interface functions or, better, at how they are now implemented in the `ZFXD3D` class using Direct3D.

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

   // anything to be cleared?
   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;
   } // BeginRendering
/*----------------------------*/


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)
```

CH03_p.041-090.qxd 6/1/04 2:39 PM Page 81

```
      m_pDevice->BeginScene();
   } // Clear
/*--------------------------*/


void ZFXD3D::EndRendering(void)
   {
   m_pDevice->EndScene();
   m_pDevice->Present(NULL, NULL, NULL, NULL);
   m_bIsSceneRunning = false;
   } // EndRendering
/*--------------------------*/


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

Before you can render, you have to begin the *scene,* as Direct3D calls it. Each API uses naming conventions as it likes and so I called this one `BeginRendering()`, which looks more native to me and more or less explains itself. (Try to explain to someone what exactly a scene is and why you have to begin a scene.) This step itself is necessary to instruct the video adapter to prepare for rendering.

You have to provide three parameters of type `bool` to control which buffers will be cleared before you begin rendering. You will almost always want to wipe out the depth buffer—but only if you use one. The same applies for the stencil buffer. Ever tried to do a clear on a nonexistent depth or stencil buffer? Direct3D shows you some nice blinking colors if you do this.

Things get looser with the pixel buffer. It makes sense to clear the pixel buffer each frame; however, if you know you will render each pixel of the pixel buffer in the new frame, there is no need to clear the pixel buffer because its contents will be overwritten anyway. Therefore, it can save you time and an expensive fill rate if you can save yourself from clearing that buffer.

Note

In computer graphics, you encounter a lot of terms that mean the same thing. Or to put it the other way around, you find a lot of different terms describing the same object or concept. Roughly speaking, a pixel buffer is the same thing as back buffer–front buffer, frame buffer, or just render target.

The remaining functions are self-explanatory. A separate function does a clear without beginning a scene. You need that for some effects, such as when you have to clear buffers during a scene. If the engine sees that the scene is currently running, it will stop the scene, clear, and then start the scene again but without calling the `IDirect3DDevice9::Present` function from Direct3D.

This is done only when you explicitly end the rendering for the engine. Please note that you must not call `IDirect3DDevice9::Present` more than once for a swap chain per frame. Otherwise, Direct3D will thank you with its nice blinking colors or a messed up screen.

You are now truly, truly done with the implementation issues of this chapter. This I promise you. Both projects are complete and you have only to see the code that is needed to make a test run of your brand new, small but fine engine.

Testing the Implementation

Most of us have already worked with a 3D API, at least enough to know that it is a whole lot of work to crank up such an API, to keep an eye on errors that could occur, select screen modes, and so on. Those are the nasty things we locked away deep inside the DLL that implements the render device interface. As you will see, the more code we can abstract away from an application because of low-standard encapsulation work, the better it is and the shorter the source code of an application using the engine naturally gets.

If you would be so kind now, open a new project in Visual C++ as a Win32 Application and call it Demo or whatever you want it to be called. Please don't use animal names.

You already know how to create a new project and add new empty files to it; therefore, insert `main.cpp` and `main.h` into the project. To eliminate some batch file work and even more Visual C++ settings, copy the files the demo project needs to use from the `ZFXRenderer` directory and the subdirectories that contain the static library implementation and the compiled library itself.

It would be easier to use batch files called in the post-build step to copy these files and to give path names to the IDE (*Integrated Development Environment*) so that it can find the directories for itself. However, you then need to know your way around in Visual C++, so just copy the following files into the directory of your Demo project space:

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

- `ZFXRenderer.lib`
- `ZFXD3D.dll`

With this completed, you can start coding. You will need only four short functions. One of them naturally needs to be the `WinMain()` function, which is accompanied by a message procedure callback. The other two functions necessary for internal organization in the code are the function for initialization of the engine and the function to shut it down.

Here I list for you the program as I implemented it. Object-oriented programmers might call this a mess, but then it is only a demo and it is meant to be simple and to the point without creating a smart class encapsulation with the Windows API and without the cool design stuff. Following is the code:

```
/////////////////////////////////////////////////////////
// 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"  // the interface
#include "ZFX.h"           // return values
#include "main.h"          // prototypes

// link the static library
#pragma comment(lib, "ZFXRenderer.lib")

// Windows stuff
HWND      g_hWnd  = NULL;
HINSTANCE g_hInst = NULL;
TCHAR     g_szAppClass[] = TEXT("FrameWorktest");

// application stuff
BOOL  g_bIsActive = FALSE;
bool  g_bDone     = false;
FILE *pLog        = NULL;
```

```
// zfx objects
LPZFXRENDERER     g_pRenderer = NULL;
LPZFXRENDERDEVICE g_pDevice   = NULL;


/**
 * WinMain entry point
 */
int WINAPI WinMain(HINSTANCE hInst,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
   {
   WNDCLASSEX      wndclass;
   HRESULT         hr;
   HWND            hWnd;
   MSG             msg;

   // initialize the window
   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,
                  "Crancking 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");

// start the 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
ProgramCleanup();
```

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


/**
 * MsgProc to proceed Windows messages.
 */
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg,
                        WPARAM wParam,
                        LPARAM lParam)
   {
   switch(msg)
      {
      // application focus
      case WM_ACTIVATE:
         {
         g_bIsActive = (BOOL)wParam;
         } break;

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

      // destroy window
      case WM_DESTROY:
         {
         g_bDone = true;
         PostQuitMessage(0);
         return 1;
         } break;
      default: break;
      }
   return DefWindowProc(hWnd, msg, wParam, lParam);
   }
/*----------------------------*/


/**
 * Create the render device object.
 */
HRESULT ProgramStartup(char *chAPI)
   {
   HWND hWnd3D[4];
   RECT rcWnd;
   int  x=0,y=0;

   // we don't have OpenGl at all :)
   if (strcmp(chAPI, "OpenGL")==0) return S_OK;

   // create the renderer object
   g_pRenderer = new ZFXRenderer(g_hInst);

   // create the render device
   if (FAILED( g_pRenderer->CreateDevice(chAPI) ))
      return E_FAIL;

   // save pointer to the render device
   g_pDevice = g_pRenderer->GetDevice();
   if(g_pDevice == NULL) return E_FAIL;

   // query client area size
   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);
      }

   // initialize render device (show dialog box))
   return g_pDevice->Init(g_hWnd,   // main window
                           hWnd3D,   // child windows
                           4,        // 4 children
                           16,       // 16 bit Z-Buffer
                           0,        // 0 bit Stencil
                           false);
   } // ProgramStartup
/*----------------------------*/


/**
 * Free allocated resources.
 */
HRESULT ProgramCleanup(void)
   {
   if (g_pRenderer)
      {
      delete g_pRenderer;
      g_pRenderer = NULL;
      }
   if (pLog) {
      fclose(pLog);
      pLog = NULL;
      }

   return S_OK;
   } // ProgramCleanup
/*----------------------------*/
```

What you see here is nothing more than a typical Windows application that opens up one window to start and provides a message pump to process the messages the operation system throws in your direction. There is a little more going on here. This is what sits inside the `ProgramStartup()` function and involves initializing the basic engine.

First, I get an instance from the `ZFXRenderer` class, which is the static library used to load a DLL that implements the render device interface `ZFXRenderDevice`. Then I call the

`ZFXRenderer::CreateDevice` function, which initializes the render device and brings up a dialog in which you can select the settings to run the application in.

In the `ProgramStartup()` function you can also see some weird code that creates four child windows, which are distributed evenly inside the main window's client area. The handles of these windows are given to the initialization function, and if the user selects that he wants to run the engine in windowed mode, you will see the four child windows you can render into just as you saw in Figure 3.2. If the user selects fullscreen mode then it doesn't matter that you hand over the child window handles to the engine. They will get ignored.

To end the program you can either hit the Escape key or use the window termination symbol from the window's caption bar in windowed mode.

One Look Back, Two Steps Forward

This chapter served as a refresher on several topics. You revisited some of the basic programming skills, but you should have learned a lot of new things, namely interfaces and DLLs. Using these two concepts or techniques, you are now able to encapsulate an API and hide its usage deep inside a library. I showed you how you can apply these methods and build a generic engine as an intermediary between a low-level API and the high-level game code.

If you end up writing all API-related stuff right into the game code or use API-dependent calls inside the game code, you will find yourself in a dead end one day. Even a simple switch to the next version of an API you use inside your game code would mess things up, as you would have to write the whole game code from scratch. Don't even try to just rewrite this and that over here and over there. It would just make things worse.

If you cleanly separate your game code from all API-dependent code using a mid-level engine, encapsulating an API, and hiding it totally to outsiders, everything will be okay. You just need to rewrite the engine, leaving the calling methods untouched. Then you do not need to change the game code. It will work with the new API version as soon as the engine is wrapped around this new version.

Another point that might be new to you is the flexible handling of multiple child windows. This enables you to build tools using typical level-editor-style front, side, top and perspective views, as well as a separate view for a material editor and stuff like that. We will revisit this flexibility again when we add the option for multiple viewports inside these child windows in Chapter 6, "The Render Device of the Engine." A viewport is basically a window inside a window; however, the difference is that it is not a real Windows window.

You should now have a fully functionally workspace with two projects that are the kickoff for a simple engine. Nothing will stop us from expanding that basic framework into a comprehensive engine.

The next chapter deals with some other basics you will hate at first, but you will learn to love later on. Trust me. Put on your helmet, soldier, and take point. If you encounter math problems, then fire for effect.