FOCUS ON

2D IN DIRECT3D

Ernest Pazera

Series Editor
André LaMothe
CEO Xtreme Games LLC
Focus on 2D in Direct3D

Ernest Pazera
To Emi and Estelle,
for your understanding and patience,
and for not hating me for making the book late.

Also dedicated to John Hattan,
because I owed him a book dedication.
This book was made possible by a cast of characters too numerous to mention, but I'll point out the ones I remember, and hope that the people I forget (or don't even know about) will forgive me and not T.P. my house.

First, I'd like to thank André LaMothe, for saying “Okay” to a 2D in 3D book, and just in general for getting me into the book writing biz in the first place.

Second, I'd like to thank two lovely, talented, and dedicated women, Emi Smith and Estelle Manticas—my editors.

Third, Lorenzo D. Phillips, Jr., my technical editor and an all around great guy. He has had much faith in me, and that's meant a lot. I hope I didn't disappoint.

The usual cast of characters: Dave Astle, Mason McCuskey, Kevin Hawkins, John Hattan, Ron Penton, and all the other folks that I frequently associate with on the Internet.

And everybody else, thank you too. That's right, everybody. The entire citizenry (alive, dead, or currently in a womb or test tube somewhere) of Planet Earth. That way I don't miss anyone.
About the Author

**Ernest Pazera** is a man who wears many hats, figuratively speaking (he does not like real hats, nor ties, nor shoes). He writes books (the one you are holding is his third), he makes games and other software, he works on GameDev.net writing scripts for the Web, and he runs a store in his hometown of Kenosha, Wisconsin. When not engaged in one of these activities, he enjoys watching and/or reading science fiction, playing most types of games (not limited to those on the computer), and in recent months has become really fond of grapefruit juice.
### Chapter 3  Vertex Formats  ![37](image)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Purpose of Direct3D</td>
<td>38</td>
</tr>
<tr>
<td>Representing Vertex Data</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>40</td>
</tr>
<tr>
<td>RHW</td>
<td>41</td>
</tr>
<tr>
<td>Vertex Normal</td>
<td>42</td>
</tr>
<tr>
<td>Diffuse Color</td>
<td>43</td>
</tr>
<tr>
<td>Specular Color</td>
<td>43</td>
</tr>
<tr>
<td>Texture Coordinates</td>
<td>44</td>
</tr>
<tr>
<td>Telling the Device Which Format You're Using</td>
<td>45</td>
</tr>
<tr>
<td>Transformed or Untransformed</td>
<td></td>
</tr>
<tr>
<td>Vertex Buffers</td>
<td></td>
</tr>
<tr>
<td>Creating a Vertex Buffer</td>
<td>46</td>
</tr>
<tr>
<td>Loading the Buffer with Data</td>
<td>48</td>
</tr>
<tr>
<td>Telling the Device Which Buffer to Use</td>
<td>50</td>
</tr>
<tr>
<td>Destroying the Buffer</td>
<td>50</td>
</tr>
<tr>
<td>Index Buffers</td>
<td></td>
</tr>
<tr>
<td>Creating an Index Buffer</td>
<td>51</td>
</tr>
<tr>
<td>Loading an Index</td>
<td>52</td>
</tr>
<tr>
<td>Telling the Device Which Indices to Use</td>
<td>53</td>
</tr>
<tr>
<td>Destroying Index Buffers</td>
<td>53</td>
</tr>
<tr>
<td>Summary</td>
<td>54</td>
</tr>
</tbody>
</table>

### Chapter 4  Drawing Primitives  ![55](image)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering States</td>
<td>56</td>
</tr>
<tr>
<td>Beginning and Ending a Scene</td>
<td>59</td>
</tr>
<tr>
<td>Primitive Types</td>
<td>59</td>
</tr>
<tr>
<td>Drawing Primitives without a Vertex Buffer</td>
<td>62</td>
</tr>
<tr>
<td>Indexed Primitives with User Pointers</td>
<td>67</td>
</tr>
<tr>
<td>Drawing Primitives with Vertex Buffers</td>
<td>70</td>
</tr>
<tr>
<td>Drawing Indexed Primitives with Index Buffers</td>
<td>71</td>
</tr>
<tr>
<td>Summary</td>
<td>72</td>
</tr>
</tbody>
</table>

### Chapter 5  Surfaces  ![73](image)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Types</td>
<td>74</td>
</tr>
<tr>
<td>Back Buffers</td>
<td>76</td>
</tr>
<tr>
<td>Image Surfaces</td>
<td>77</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------</td>
</tr>
<tr>
<td>6</td>
<td>Textures</td>
</tr>
<tr>
<td>chapter 6</td>
<td>Textures</td>
</tr>
<tr>
<td>Creating a Texture</td>
<td>94</td>
</tr>
<tr>
<td>Loading Texture Data.</td>
<td>96</td>
</tr>
<tr>
<td>Texture Coordinates</td>
<td>96</td>
</tr>
<tr>
<td>Applying Textures to Primitives</td>
<td>97</td>
</tr>
<tr>
<td>Texture Address Modes.</td>
<td>99</td>
</tr>
<tr>
<td>Multi-Texturing</td>
<td>102</td>
</tr>
<tr>
<td>Summary</td>
<td>106</td>
</tr>
<tr>
<td>7</td>
<td>Alpha Testing</td>
</tr>
<tr>
<td>chapter 7</td>
<td>Alpha Testing</td>
</tr>
<tr>
<td>Alpha Testing</td>
<td>108</td>
</tr>
<tr>
<td>Loading an Image as a Texture</td>
<td>110</td>
</tr>
<tr>
<td>One Image, Many Textures.</td>
<td>112</td>
</tr>
<tr>
<td>Summary</td>
<td>116</td>
</tr>
<tr>
<td>2</td>
<td>Extending Direct3D</td>
</tr>
<tr>
<td>Part Two</td>
<td>Extending Direct3D</td>
</tr>
<tr>
<td>chapter 8</td>
<td>Coordinate Spaces</td>
</tr>
<tr>
<td>The Coordinate Spaces</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>Model Space</td>
</tr>
<tr>
<td></td>
<td>World Space</td>
</tr>
<tr>
<td></td>
<td>View Space</td>
</tr>
<tr>
<td></td>
<td>Cube Space</td>
</tr>
<tr>
<td></td>
<td>Geometry Pipeline.</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
<tr>
<td>chapter 9</td>
<td>Vectors</td>
</tr>
<tr>
<td>Vector: Defined</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>D3DVector</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
</tbody>
</table>
CHAPTER 13 Lighting ........................................ 175
Direct3D Lighting ............................................. 176
D3DMATERIAL8 ................................................ 177
D3DLIGHT8 ...................................................... 179
Type ............................................................. 180
Ambient, Diffuse, and Specular ......................... 180
Attenuation0m Attenuation1, and Attenuation2 ........ 182
Theta, Phi, and Falloff. ........................................ 182
Setting up Lights .............................................. 184
Point Lights ..................................................... 185
DirectionalLights .............................................. 187
Spotlights ....................................................... 187
Ambient Light ................................................... 189
Specular Light ................................................... 192
Emissive Light ................................................... 192
Summary .......................................................... 194

CHAPTER 14 Alpha Blending ......................... 195
Alpha Blending Defined ........................................ 196
Setting up for Alpha Blending ............................. 198
Using Alpha Blending ........................................ 201
Summary .......................................................... 202

Part Three
Bringing It All Together ................................. 203

CHAPTER 15 Models ......................................... 205
What Is a Model? ............................................... 206
The X Files ...................................................... 207
Making a Simple X File Loader ......................... 216
The STL Vector Template ................................ 216
Getting the Data from File into Memory ............ 218
Tokenizing Text ............................................... 219
The Mesh ....................................................... 223
Grouping the Types. ......................................... 234
Summary .......................................................... 236
Letter from the Series Editor

In about 1995 I attended a little event at Microsoft HQ code-named “Judgment Day.” The main point of the event was to show developers, or should I say persuade developers, that DOS was dead and DirectX was here to stay. Some of the developers were skeptical—many had already trusted Microsoft once before with WinG, and if you ask Microsoft about that little technology now they’ll disavow any knowledge that it ever existed. In any case, this time they were right: DOS was dead, and DirectX has made a huge impact in the world of game programming, graphics, and multimedia. But it wasn’t an easy changeover. In fact, it was only after DirectX 3.0 that the majority of game developers switched to DirectX and forgot DOS4G and 32-bit flat mode DOS game programming once and for all.

These days there are hundreds of books on the subject of DirectX, tens of thousands of developers writing code for it, and a software base that’s in the hundreds of millions of lines of code.

Recently Microsoft decided to completely erase DirectDraw from version 8.0 of DirectX. When I heard the news, I nearly had a heart attack. Sure, you can always request version 7.0 interfaces and use DirectDraw, but still, why on earth did they do it? Well, the simple answer is that they wanted to integrate a number of DirectX components to create the new, cleaner DirectX Graphics and DirectX Audio system. This is a great idea, but still, they could have kept DirectDraw....

But no use crying over it now. What’s the damage, you might ask? Well, to do any 2D graphics, you must use the 3D interfaces—sound stupid? Well of course it is, but with a little code and a lot of patience you can use Direct3D’s new interfaces to perform 2D graphics, plus you get all kinds of cool
effects like scaling, rotation, transparency, and lighting. Basically, you’re going to render 2D images and sprites as flat polygons, so anything you can do to a polygon, you can do to these new “3D Sprites.”

To help you make this painful transition, Mr. Pazera has written this great book. *Focus on 2D in Direct3D* will show you how to use the new DirectX 8.0 interfaces to create 2D sprites and 3D effects using those sprites. In these pages you’ll learn about IDirect3D8, the new interface that is the key to doing anything with Direct3D. You’ll also learn how to create surfaces and textures, and see how to manipulate and transform these objects using Direct3D hardware acceleration. After you’re comfortable with 2D the book moves into some 3D concepts and helps you leverage some of the 3D abilities in your 2D work. The book ends with a number of game programming examples that illustrate scrolling, top down view games, ISO games, and more.

This book is not something you *should* have in your library, it’s something you *must* have in your library! I assure you, you don’t want to take the time to figure all this stuff out on your own. The best way to do things—the right methods rather than just what works—have been painfully researched and tested by Mr. Pazera, and this book is the result of his work, presented in a format that you can absorb and put to use immediately in your games.

Sincerely,

[Signature]

André LaMothe
Series Editor
This page intentionally left blank
Introduction

Welcome to my book. I sincerely hope you enjoy it. I, as well as many other people (most of whom I have not met nor am likely ever to meet), worked very hard and very long hours to bring it to you in the form you now hold it (presumably in your hands at a bookstore somewhere).

If you have already skipped ahead to Chapter 1, you probably noticed that I get right into the subject at hand immediately, rather than taking a couple of hundred pages to wade slowly into the topic by first going over basic Windows programming and basic DirectX information.

The Focus On... series isn’t like that, for a reason. The biggest complaints I got about my last book, *Isometric Game Programming with DirectX 7.0* were that it was (a) too expensive and that (b) the WIN32/DirectX tutorial sucked up a full third of the book. The reasons for the complaints are related. Because of the 250-page WIN32/DirectX section, *Isometric* swelled to over 700 pages. More pages make the cost of printing the book higher, and that cost gets passed along to you, the consumer (that’s just how business works). So no more basics. Nowadays there are dozens of books out there that have the basic WIN32/DirectX information in them (and chances are that you already own one), so no further beating of the (very) dead horse in this book.

However, because I don’t cover the basics, to get the most out of this book you should already be at a certain level of knowledge in your game programming (for exactly what that level is, see the “Who You Are” section below).

What’s in this Book?

This book is about Direct3D, a component of DirectX used primarily to render three dimensional scenes (or at least a rough approximation of them). However, unlike all the other Direct3D books, I don’t
actually cover 3D rendering in the traditional sense. I cover 2D render-
dering, which is a very different approach to the topic of Direct3D. I
will concentrate on showing you what the API actually does rather than
telling you how to do what it was originally designed for. The differ-
ence is subtle, but important.

For the most part, I have made this book into something of a bridge.
While there is currently a place for 2D games (and indeed there
always will be... not all games need 3D), it is shrinking steadily. Where
does this bridge lead? Well, after reading this book, you’ll have a
much better grasp of what is really going on in Direct3D, and so going
back to those Direct3D books that befuddled you before should be
easy.

Who You Are

You are a 2D programmer. You should be familiar with some sort of
2D API, be it GDI, DirectDraw, SDL, or something else. Above all else,
you should have a good command of the techniques used for 2D
game programming. Naturally, this means that you should be a game
programmer, and somewhere above the rank of total newbie; but you
don’t have to know a whole lot.

The ideal scenario is that you are a 2D game programmer who has
looked at/purchased/tried to read one or more books on Direct3D
without success. For whatever reason, Direct3D lies just beyond your
grasp. All of the books on the topic and/or all of the tutorials on the
Web have frustrated you, as, even if they are on the topic of 2D with
Direct3D, they are still written from the 3D programmer perspective,
not your perspective as a 2D programmer.

The programming language of this book is C++, and I do use some
C++-specific features, but a good knowledge of C should get you by,
for the most part.
PART ONE

From DirectDraw 7.0 to Direct3D 8.0
1 Initializing Direct3D
2 Creating a Device
3 Vertex Formats
4 Drawing Primitives
5 Surfaces
6 Textures
7 Alpha Testing
CHAPTER 1

Initializing Direct3D
Until the appearance of DirectX 7.0, life was pretty easy for the 2D game programmer. The DirectDraw section of DirectX went through relatively minor changes from one version of the API to the next. These changes were more like refinements than actual differences in functionality. As of DirectX 8.0, this is no longer the case.

Although the DirectX 7.0 version of DirectX is still available, it has not been updated, and if you want to continue to make 2D applications and games you must either use the legacy DirectDraw 7.0 or move into Direct3D 8.0. This is not an impossible task by any means. Indeed, I will be taking you into the shallow end before totally immersing you in all that Direct3D has to offer. In fact, the first half of this book is concerned with translating all that you have done in DirectDraw into its Direct3D equivalent.

The mission of this first chapter is to expose you to the IDirect3D8 object, which is the Direct3D 8.0 equivalent of the IDirectDraw7 object from DirectDraw 7.0.

Your goals for this chapter should be the following:

- Understand the initialization steps for Direct3D 8 and how they compare to the initialization steps taken in DirectDraw 7.0
- Understand the purpose of the IDirect3D8 object
- Know how to create an IDirect3D8 object
- Know how to examine the capabilities of an IDirect3D8 object on the adapters available to the system
- Understand how Direct3D represents display formats
- Know how to enumerate adapter display modes
- Know how to check for available media formats of a particular display mode
Initialization Steps

When initializing DirectDraw in an application, you go through the same steps (some of which are optional) every time. The same is true when initializing Direct3D. The steps you take in DirectDraw are as follows:

1. Enumerate display adapters (optional).
2. Create an IDirectDraw7 object.
3. Set the cooperative level of the IDirectDraw7 object.
4. Examine the capabilities of the IDirectDraw7 object (optional).
5. Enumerate display modes (if targeting full screen).
6. Select and set an appropriate display mode (if targeting full screen).
7. Create primary surface and back buffers or double buffers (as appropriate for windowed or full screen).
8. Load game images onto off-screen surfaces for later use.

Naturally, if you are using the subset of DirectDraw that is compatible with all video cards (as I tend to do), you often won’t have to examine the object’s capabilities. Also, if you are making a windowed application, you won’t be enumerating display modes.

When moving to Direct3D you also need to go through a number of steps to prepare applications that will use it. Many of the steps are roughly analogous to the steps taken to initialize DirectDraw.

1. Create an IDirect3D8 object.
2. Examine the capabilities of the IDirect3D8 object and choose an appropriate device type.
3. Decide on a display adapter and mode for that adapter.
4. Create and initialize an IDirect3DDevice8 object.
5. Load the game’s images onto textures.

You might be surprised to see that the initialization of the IDirect3D8 object is actually shorter than the IDirectDraw7 initialization. Don’t be
fooled. Creating and initializing the IDirect3DDevice8 object can be rather involved, depending on what you want to do. Because you are using Direct3D for something other than its intended purpose, this book will explore some unconventional ideas.

The remainder of this chapter discusses steps 1 through 3. Hopefully, this chapter will help demystify the beast that is Direct3D.

Purpose of the IDirect3D8 Object

In DirectDraw 7.0, the IDirectDraw7 object was rather important. It operated as a “factory” to create other DirectDraw objects, including surfaces, clippers, and palettes.

The IDirect3D8 object is only slightly similar to the IDirectDraw7 object. It is not used as the core factory for creating other Direct3D objects; it only creates an IDirect3DDevice8 (examined in Chapter 2).

The primary purpose of an IDirect3D8 object is to examine capabilities so that you can pick a device that will best be able to run your game. Because of the wide variety of video cards, the features that you want to use in your game might be unavailable on some machines. Many of these missing features can be emulated in software, but as you may know, emulation of a feature often slows down a game that would be lightning fast in the presence of hardware acceleration. For the foreseeable future, you just have to live with this. Be careful with your system requirements, so as not to totally alienate your target audience.

Creating an IDirect3D8 Object

To initialize the Direct3D 8 subsystem of DirectX, start with a call to the Direct3DCreate8 function, which is nearly analogous to the DirectDrawCreateEx function from DirectDraw. Here is the prototype:

```c
IDirect3D8* Direct3DCreate8(
    UINT SDKVersion
);
```
This function takes one parameter, which must always be the constant D3D_SDK_VERSION. The return value is a pointer to an IDirect3D8 object. The following is a snippet of code that creates, and then destroys, an IDirect3D8 object:

```c
IDirect3D8* pd3d; //pointer to an IDirect3D8 object
pd3d = Direct3DCreate8 ( D3D_SDK_VERSION ) ; //create IDirect3D8 object
//destroy IDirect3D8 object
if ( pd3d )
{
    pd3d -> Release ( ) ;
    pd3d = NULL ;
}
```

As with all other DirectX objects, IDirect3D8 is a COM object, and must be released in order to “destroy” it. The internal reference counting done by DirectX ensures that it is not destroyed until it is no longer needed.

Creating an IDirect3D8 object is a lot simpler than calling DirectDrawCreateEx or calling CoCreateInstance (yuck).

Example 1.1 on the CD (X:\Source\Example1_1) is a very simple example. It sets up a simple window, creates an IDirect3D8 object, then releases it. Diagnostic output is sent to stdout.txt. Figure 1.1 shows the application (it isn’t much), and Figure 1.2 shows the output to stdout.txt.

![Figure 1.1 Creating your first IDirect3D8 object](image)
Examining Capabilities

Once you’ve created your IDirect3D8 object, your work begins in earnest. You have to check different device types, potentially on more than one display adapter, for the functionality that you must have for your game. This “examination” is like enumeration (which is simply the process of creating a list of something; in our case, what we’re listing is either display adapters, like video cards, and/or display modes for those display adapters) but is not done in the same way that prior versions of Direct3D did its enumeration, and thus not similar to how DirectDraw did its, either. I had a little trouble at first, because I was used to the callback enumeration common with DirectX 7.0 and lower. However, the process is not terribly difficult.

Adapters

First off, I’m going to show you how to determine which display adapters are attached to the system. More than one video card might be attached to the system, and you should be ready for such a contingency.

In DirectDraw, you used DirectDrawEnumerate or DirectDrawEnumerateEx to grab the available display adapters prior to the creation of the IDirectDraw7 object. In Direct3D 8.0, you do this after the object is created.
There are three functions used together to enumerate display adapters in Direct3D 8: IDirect3D8::GetAdapterCount, IDirect3D8::GetAdapterMonitor, and IDirect3D8::GetAdapterIdentifier.

The IDirect3D8::GetAdapterCount function’s prototype is as follows:

```c
UINT IDirect3D8::GetAdapterCount ( ) ;
```

This function takes no parameters and returns a UINT (unsigned int). The returned value is the number of display adapters available to the system. If this function returns 1, your decision is made for you, because you only have one adapter to choose from.

The adapters are numbered from 0 to the result of GetAdapterCount() minus 1. So if the function returns a 1, the only adapter number is 0. If the return value is 2, 0 and 1 are available, and so on. There is a special value for the default display adapter, D3DADAPTER_DEFAULT. Its value is 0. The default adapter is always 0.

After you have the number of display adapters you can take a look at them by using IDirect3D8::GetAdapterIdentifier. This function gives you access to important human-readable information about that adapter. Here is the prototype:

```c
HRESULT GetAdapterIdentifier(  
    UINT Adapter,  
    DWORD Flags,  
    D3DADAPTER_IDENTIFIER8* pIdentifier  
);
```

Like most DirectX functions, this one returns an HRESULT, which reports the success or failure of the function. The values returned are either D3D_OK (success) or D3DERR_INVALIDCALL, which means you probably messed up one of the parameters (hey, it happens).

The Adapter parameter is a UINT, and is the number of the adapter for which you are retrieving information. This number can be between zero and the number of adapters minus one.

The Flags parameter is either 0 or the constant D3D_ENUM_NO_WHQL_LEVEL. I’ll get to the meaning of this flag in a moment. Typically, 0 is sent in this parameter.

Finally, pIdentifier is a pointer to a D3DADAPTER_IDENTIFIER8 structure.
You didn’t think you’d be getting away from the gamut of DirectX structs, did you? Table 1.1 shows `D3DADAPTER_IDENTIFIER8` members and their meanings.

The two strings, `Driver` and `Description`, are meant for human readability. If, for example, you wanted to populate a combo box with the various drivers available to the system, you could use these members. Most of the other members are used to identify particular chipsets.

The `WHQLLevel` member contains a bit-encoded date, a 1 or a 0. WHQL stands for Window Hardware Quality Lab. If this value is 0, the driver/adapter pair is not certified. If this value is 1, it has been certified, but no date information is available. Otherwise, bits 0 through 7 have the day, bits 8 through 15 have the month, and bits 16 through 31 have the year. If you specify `D3D_ENUM_NO_WHQL_LEVEL`, this member will be filled in with 0 no matter what. This can be advantageous, as it takes a couple of seconds to retrieve this value from the display driver.

<table>
<thead>
<tr>
<th>Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>String containing the name of the driver</td>
</tr>
<tr>
<td>Description</td>
<td>String containing a description of the adapter</td>
</tr>
<tr>
<td>DriverVersion</td>
<td>Version of the driver</td>
</tr>
<tr>
<td>VendorID</td>
<td>Used to identify a particular chipset</td>
</tr>
<tr>
<td>DeviceID</td>
<td>Used to identify a particular chipset</td>
</tr>
<tr>
<td>SubSysID</td>
<td>Used to identify a particular chipset</td>
</tr>
<tr>
<td>Revision</td>
<td>Used to identify a particular chipset</td>
</tr>
<tr>
<td>DeviceIdentifier</td>
<td>A GUID for the driver and adapter</td>
</tr>
<tr>
<td>WHQLLevel</td>
<td>A certification date</td>
</tr>
</tbody>
</table>
Example 1.2 on the CD (X:\Source\Example1_2) enumerates the display adapters and reports their Driver and Description members to stdout.txt. The example is still rudimentary at this point; it pretty much only shows you a black window that you’ll want to immediately close anyway. Figure 1.3 shows the contents of stdout.txt generated by this program on my machine.

**Device Types**

Now that you can peek at your system’s display adapters (even if you have only one), you’re ready to take a look at these driver’s capabilities for different types of devices.

Because of widely varying hardware support, there are three types of devices you can make:

- One is a device that works with the *hardware abstraction level* (HAL). This type of device will give you the hardware acceleration you want. Unfortunately, if a feature isn’t supported on the adapter you are using, the HAL device will not emulate the feature.

- Another device type is called the *reference rasterizer*. This device type gives you most of the functionality of Direct3D emulated in software. It does use special CPU instructions whenever it can,
but as with all software emulation, it can be slow compared to using the HAL.

- The third type of device is a *pure software device*. I will not cover these as they are a book unto themselves.

Each of the device types has a value in the D3DDEVTYPE enumeration. Table 1.2 shows these values, their numerical equivalents, and which devices they represent.

Because this book ignores D3DDEVTYPE_SW, you have to worry only about D3DDEVTYPE_HAL and D3DDEVTYPE_REF. Therefore, you can examine the capabilities of each device on each of the display adapters attached to the system.

The function for doing just that is called IDirect3D8::GetDeviceCaps. Here is the prototype:

```csharp
HRESULT GetDeviceCaps(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
    D3DCAPS8* pCaps
);
```

The return value is another HRESULT. If successful, it returns D3D_OK. If it’s not successful, it returns one of D3DERR_INVALIDCALL, D3DERR_INVALIDDEVICE, or D3DERR_OUTOFVIDEOMEMORY.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DDEVTYPE_HAL</td>
<td>1</td>
<td>Hardware acceleration using the HAL</td>
</tr>
<tr>
<td>D3DDEVTYPE_REF</td>
<td>2</td>
<td>Reference rasterizer</td>
</tr>
<tr>
<td>D3DDEVTYPE_SW</td>
<td>3</td>
<td>Software rasterizer</td>
</tr>
</tbody>
</table>
The first parameter is the number of the adapter you are checking. The second parameter is a D3DDEVTYPE value, either D3DDEVTYPE_HAL or D3DDEVTYPE_REF. The last parameter is a pointer to a D3DCAPS8 structure. The capabilities (and there are a number of them) are copied into this structure so that you can examine them later. The following minitable lists the members of D3DCAPS8.

<table>
<thead>
<tr>
<th>DeviceType</th>
<th>AdapterOrdinal</th>
<th>Caps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caps2</td>
<td>Caps3</td>
<td>caps</td>
</tr>
<tr>
<td>CursorCaps</td>
<td>DevCaps</td>
<td>PrimitiveMiscCaps</td>
</tr>
<tr>
<td>RasterCaps</td>
<td>ZCmpCaps</td>
<td>SrcBlendCaps</td>
</tr>
<tr>
<td>DestBlendCaps</td>
<td>AlphaCmpCaps</td>
<td>ShadeCaps</td>
</tr>
<tr>
<td>TextureCaps</td>
<td>TextureFilterCaps</td>
<td>CubeTextureFilterCaps</td>
</tr>
<tr>
<td>VolumeTextureFilterCaps</td>
<td>TextureAddressCaps</td>
<td>VolumeTextureAddressCaps</td>
</tr>
<tr>
<td>LineCaps</td>
<td>MaxTextureWidth</td>
<td>MaxTextureHeight</td>
</tr>
<tr>
<td>MaxVolumeExtent</td>
<td>MaxTextureRepeat</td>
<td>MaxTextureAspectRatio</td>
</tr>
<tr>
<td>MaxAnisotropy</td>
<td>MaxVertexW</td>
<td>GuardBandLeft</td>
</tr>
<tr>
<td>GuardBandTop</td>
<td>GuardBandRight</td>
<td>GuardBandBottom</td>
</tr>
<tr>
<td>ExtentsAdjust</td>
<td>StencilCaps</td>
<td>FVFCaps</td>
</tr>
<tr>
<td>TextureOpCaps</td>
<td>MaxTextureBlendStages</td>
<td>MaxSimultaneousTextures</td>
</tr>
<tr>
<td>VertexProcessingCaps</td>
<td>MaxActiveLights</td>
<td>MaxUserClipPlanes</td>
</tr>
<tr>
<td>MaxVertexBlendMatrices</td>
<td>MaxVertexBlendMatrixIndex</td>
<td>MaxPointSize</td>
</tr>
<tr>
<td>MaxPrimitiveCount</td>
<td>MaxVertexIndex</td>
<td>MaxStreams</td>
</tr>
<tr>
<td>MaxStreamStride</td>
<td>VertexShaderVersion</td>
<td>MaxVertexShaderConst</td>
</tr>
<tr>
<td>PixelShaderVersion</td>
<td>MaxPixelShaderValue</td>
<td></td>
</tr>
</tbody>
</table>

There are so many of these that explaining them all would be more confusing than helpful. The important thing here is to know how to read the capabilities of a device type and adapter. Most of the members of D3DCAPS8 are bit flags, and when checking for the support of something in particular, you check a member against one of those bit flags. For a detailed explanation of these members check out the DirectX documentation.
Just as a quick example, I’ll show you how to check for support of windowed Direct3D. The bit for that is in Caps2. The value of the bit is D3DCAPS2_CANRENDERWINDOWED. To grab the device capabilities and then check for windowed rendering, you might do the following:

```
//g_pd3d is a pointer to an IDirect3D8, initialized prior to this snippet
D3DCAPS8 caps;
g_pd3d->GetDeviceCaps ( 0 , DEVTYPE_HAL , &caps ) ;
if ( caps.Caps2 & D3DCAPS2_CANRENDERWINDOWED )
{
    //you can render to a window
}
else
{
    //you cannot render to a window
}
```

This is essentially how you use the capabilities of Direct3D. You check for whatever capabilities you might need, and use that information to determine what you can use or which device type you must pick.

As situations come along throughout the rest of the book, I will point out which capabilities must be checked in order to use the feature at hand. If I were to explain all of them now, you’d be reading about them for the next 50 or so pages.

### Display Formats

If you’re at all like me, one of the things you really hated about DirectDraw was the use of some of the informational structures, like DDSURFACEDESC2 and DDPIXELFORMAT. Although Direct3D 8 does have structures like these, they are not nearly as complicated as those in DirectDraw, as you have seen with D3DADAPTER_IDENTIFIER8 or D3DCAPS8.

One DirectDraw structure that I particularly found hard to work with was DDPIXELFORMAT, which you used any time you wanted to determine how many bits the red, green, or blue components were using. Because it didn’t come right out and tell you, you had to calculate the bits yourself.

In Direct3D 8, that has changed somewhat. All display formats have an entry in the D3DFORMAT enumerated type. Table 1.3 lists the most commonly used values.
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DFMT_R8G8B8</td>
<td>24 bits per pixel; eight bits for each of red, green, and blue</td>
</tr>
<tr>
<td>D3DFMT_A8R8G8B8</td>
<td>32 bits per pixel; eight bits for each of alpha, red, green, and blue</td>
</tr>
<tr>
<td>D3DFMT_X8R8G8B8</td>
<td>32 bits per pixel; eight bits for each of red, green, and blue; eight bits remain unused</td>
</tr>
<tr>
<td>D3DFMT_R5G6B5</td>
<td>16 bits per pixel; five bits for red and blue, six bits for green</td>
</tr>
<tr>
<td>D3DFMT_X1R5G5B5</td>
<td>16 bits per pixel; five bits for each of red, green, and blue; one bit is unused</td>
</tr>
<tr>
<td>D3DFMT_A1R5G5B5</td>
<td>16 bits per pixel; five bits for each of red, green, and blue; one bit reserved for alpha</td>
</tr>
<tr>
<td>D3DFMT_A4R4G4B4</td>
<td>16 bits per pixel; four bits for each of alpha, red, green, and blue</td>
</tr>
<tr>
<td>D3DFMT_R3G3B2</td>
<td>Eight bits per pixel; three bits for red and green, two bits for blue</td>
</tr>
<tr>
<td>D3DFMT_A8</td>
<td>Eight bits per pixel; all eight bits represent alpha</td>
</tr>
<tr>
<td>D3DFMT_A8R3G3B2</td>
<td>16 bits per pixel; eight bits for alpha, three bits for red and green, two bits for blue</td>
</tr>
<tr>
<td>D3DFMT_X4R4G4B4</td>
<td>16 bits per pixel; four unused bits, four bits for each of red, green, and blue</td>
</tr>
<tr>
<td>D3DFMT_D16_LOCKABLE</td>
<td>16 bits per pixel; all 16 bits used for depth information</td>
</tr>
<tr>
<td>D3DFMT_D32</td>
<td>32 bits per pixel; all 32 bits used for depth information</td>
</tr>
<tr>
<td>D3DFMT_D15S1</td>
<td>16 bits per pixel; 15 bits for depth, and one bit for stencil</td>
</tr>
<tr>
<td>D3DFMT_D24S8</td>
<td>32 bits per pixel; 24 bits for depth, eight bits for stencil</td>
</tr>
<tr>
<td>D3DFMT_D16</td>
<td>16 bits per pixel; all 16 used for depth</td>
</tr>
<tr>
<td>D3DFMT_D24X8</td>
<td>32 bits per pixel; 24 used for depth, eight unused</td>
</tr>
<tr>
<td>D3DFMT_D24X4S4</td>
<td>32 bits per pixel; 24 used for depth, four unused, four used for stencil</td>
</tr>
</tbody>
</table>
As you can see from the table, this way of describing pixel formats is pretty simple and straightforward. Each format tells you which types of information are present with a single letter (A for alpha; R, G, B for red, green, and blue; X for unused; D for depth; S for stencil). For the moment, knowing what alpha, depth, and stencil bits mean is unimportant. They are all covered later, in the appropriate chapters. The important thing here is that you know how to read these pixel formats.

Another important thing to know about these formats is that each one lists bits from the most significant bit to the least significant bit. So, in D3DFMT_R5G6B5, the top five bits are red, the next six bits are green, and the last five bits are blue.

Based on these D3DFMT values, you can easily construct functions that give you the information you need, such as bit masks for each component, the number of bits per pixel for a particular format, and so on.

**Enumerating Display Modes**

Now that you know how Direct3D 8 describes display formats, you can look at display modes for a given adapter (if you intend to make a full screen application, which is likely). If you are making a windowed application, you only need to determine which display mode is used by the adapter you are using (more about that in a moment).

**Looking at Adapter Display Modes**

You look at adapter display modes the same way you enumerated the adapters. That means you don’t have to set up a callback function, and that’s a good thing.

You use two functions in tandem to get the display mode information out of an IDirect3D8 object. The first of these is IDirect3D8::GetAdapterModeCount.

```c
UINT IDirect3D8::GetAdapterModeCount(
    UINT Adapter
);
```
The return value is a UINT; this value contains the total number of display modes available to the adapter in question. The single parameter is the number of the adapter in question. This is much simpler than the DirectDraw way of counting modes.

Now you can examine each mode with a call to IDirect3DB::EnumAdapterModes:

```c
HRESULT EnumAdapterModes(
    UINT Adapter,
    UINT Mode,
    D3DDISPLAYMODE* pMode
);
```

The return value is an HRESULT. On success, this is D3D_OK. On failure, it is D3DERR_INVALIDCALL.

The first parameter is the number of the adapter you’re looking at, and the second parameter is the number of the mode you’re looking at, which ranges from 0 to the return value of GetAdapterModeCount minus one.

The last parameter is a pointer to a D3DDISPLAYMODE structure, which is filled in with the information about the mode in question. Table 1.4 shows the members of D3DDISPLAYMODE.

---

**Table 1.4 D3DDISPLAYMODE Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>The width of the display mode, in pixels</td>
</tr>
<tr>
<td>Height</td>
<td>The height of the display mode, in pixels</td>
</tr>
<tr>
<td>RefreshRate</td>
<td>The refresh rate of the monitor, in hertz; zero means a default refresh rate</td>
</tr>
<tr>
<td>Format</td>
<td>The format of the display mode (one of the D3DFORMAT constants)</td>
</tr>
</tbody>
</table>
The D3DDISPLAYMODE structure is pretty simple and easy to understand. Much of the time, you are likely to ignore the RefreshRate member, and only concern yourself with Width, Height, and Format.

Example 1.3 on the CD (X:\Source\Example1_3) is a workspace that enumerates display adapters and reports information about them to stdout.txt. There are two new files in this workspace, d3dfmutils.h and d3dfmutils.cpp. These two files contain functions that give you information about a display format based on the D3DFORMAT value, including a string name, the bits taken up for each component, and a bitmask for that component within the format. You’ll be using this file quite a bit, especially when you have to convert from one pixel format into another. Figure 1.4 shows the stdout.txt generated by this program on my machine. Your mileage might vary.

![Figure 1.4 Example 1.3’s stdout.txt (it enumerates 20 modes; only the first five are shown)]
Getting the Current Display Mode

Now that you’ve learned how to enumerate display modes, getting the current display mode of an adapter is child’s play. The function for doing this is IDirect3D8::GetAdapterDisplayMode.

HRESULT GetAdapterDisplayMode(
    UINT Adapter,
    D3DDISPLAYMODE* pMode
);

The return value, as usual, is an HRESULT, which is D3D_OK on success or D3DERR_INVALIDCALL on failure. The two parameters are the number of the display adapter and a pointer to a D3DDISPLAYMODE that is filled in with the current display mode for the adapter in question.

Checking for Compatible Formats

In DirectDraw, you never really worried about varying the pixel formats of the different surfaces you were using. Whatever pixel format you used for the primary surface was the format used for all other surfaces, and that was that.

In Direct3D this is no longer the case. You might use a format other than the one on the display for a texture or for a depth/stencil buffer, for instance. Also, if you are making a windowed application, you need to know whether or not a particular display mode with a particular device type will allow you to do windowed rendering. This is something of a headache, but it must be done.

Checking Resource Formats

The IDirect3D8::CheckDeviceFormat function is used to check whether a particular type of resource can be used on a given adapter in a given mode using a given device type.

HRESULT IDirect3D8::CheckDeviceFormat(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
    D3DFORMAT AdapterFormat,
DWORD Usage,
D3DRESOURCETYPE RType,
D3DFORMAT CheckFormat
);

This function returns an HRESULT that is D3D_OK if successful, and
D3DERR_INVALIDCALL or D3DERR_NOTAVAILABLE on failure.

You are already familiar with most of the parameters. You send this
function an adapter number, a device type, the format of that adapter
that you will be using in your application, and the format you want to
use (as the final parameter). The two mystery parameters are Usage
and RType.

The Usage parameter is a combination of flags, which can be zero.
There are two flags you can use: D3DUSAGE_DEPTHSTENCIL and
D3DUSAGE_RENDERTARGET. Depth/stencil buffers are for depth informa-
tion and stencil information (covered in Chapter 12). Render targets
are types of resources on which you use Direct3D to draw graphics.
Naturally, not all of your resources will need one or both of these
flags, and so you can leave the Usage parameter zero.

The RType parameter is one of the constants in the D3DRESOURCETYPE
enumeration. These values are shown in Table 1.5.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DRTYPE_SURFACE</td>
<td>A generic surface</td>
</tr>
<tr>
<td>D3DRTYPE_VOLUME</td>
<td>A volume</td>
</tr>
<tr>
<td>D3DRTYPE_TEXTURE</td>
<td>A texture</td>
</tr>
<tr>
<td>D3DRTYPE_VOLUMETEXTURE</td>
<td>A volume texture</td>
</tr>
<tr>
<td>D3DRTYPE_CUBETEXTURE</td>
<td>A cube texture</td>
</tr>
<tr>
<td>D3DRTYPE_VERTEXBUFFER</td>
<td>A vertex buffer</td>
</tr>
<tr>
<td>D3DRTYPE_INDEXBUFFER</td>
<td>An index buffer</td>
</tr>
</tbody>
</table>
As far as you are concerned, only a few of these, D3DRTYPE_SURFACE, D3DRTYPE_TEXTURE, and D3DRTYPE_VERTEXBUFFER are of use. Because you are doing 2D graphics with a 3D API, the other types of resources are unnecessary.

I presented the IDirect3D8::CheckDeviceFormat function here so that you’ll be aware of it when you finally need to use it (which won’t be for a while yet).

**Checking Support**

This is the final function covered in this chapter; there are still a few more functions, but they won’t make any sense until Chapter 12, which discusses depth buffers. This final function is called IDirect3D8::CheckDeviceType. It enables you to determine whether hardware acceleration is available when you’re using a certain configuration. Here is the prototype:

```c
HRESULT IDirect3D8::CheckDeviceType(
    UINT Adapter,
    D3DDEVTYPE CheckType,
    D3DFORMAT DisplayFormat,
    D3DFORMAT BackBufferFormat,
    BOOL Windowed
);
```

The returned value is an HRESULT. It is D3D_OK if successful, and D3DERR_INVALIDCALL, D3DERR_INVALIDDEVICE, or D3DERR_NOTAVAILABLE if not successful.

The first parameter is the number of the adapter you are checking. If you exceed the range for this number, the function will return D3DERR_INVALIDCALL.

The second parameter is the type of device you are checking. If the device does not exist on your system, the function will return D3DERR_INVALIDDEVICE.

The third and fourth parameters are both display formats, one for the display itself, and another for the back buffer. In your case, these will always be the same.
Finally, the last parameter is a Boolean value. It’s TRUE if you are intending to render windowed, and FALSE if you are intending to render full screen.

If the display formats are not available, or hardware acceleration is not available for that particular display format, this function returns D3DERR_NOTAVAILABLE.

**Summary**

You have just waded through a large amount of information. Much of what this chapter covers isn’t of immediate use, but it will be later. It’s easy to forget this stuff before you need it; be sure to refer back to this chapter when necessary. Direct3D 8 is one of those “chicken and egg” APIs. You can’t understand anything about it until you actually do it, yet you can’t do anything with it until you understand it!
CHAPTER 2

Creating a Device
In DirectDraw, the most important object was IDirectDrawSurface7. In Direct3D, the most important object is the IDirect3DDevice8. This chapter gives you a baseline understanding of the IDirect3DDevice8 object. The interface is pretty big, and this single chapter is not a complete reference by any means. Instead, this chapter concerns itself with how to set up the object.

Your objectives for this chapter are to:

- Learn how to create an IDirect3DDevice8 object
- Learn how to create and use a viewport
- Learn how to clear a viewport
- Learn how to present a scene
- Learn how to reset the display with different presentation parameters

Basically, this chapter’s mission is to teach you to set up an IDirect3DDevice8 object and prepare it for the rendering you’ll be doing with it later.

Creating IDirect3DDevice8 Objects

In Chapter 1, I spent a lot of time discussing display adapters, the modes of those adapters, and the capabilities of those adapter’s devices. Chapter 1 was time well spent. Indeed, the primary goal was to determine the configuration best suited for your application. Once you figure that out, you can create an IDirect3DDevice8 object.

The function for creating the device is IDirect3D8::CreateDevice. Here is the prototype:

HRESULT IDirect3D8::CreateDevice(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
HWND  hFocusWindow,
DWORD BehaviorFlags,
D3DPRESENT_PARAMETERS* pPresentationParameters,
IDirect3DDevice8** ppReturnedDeviceInterface
);

This function returns an HRESULT. It is D3D_OK on success, or D3DERR_ 
INVALIDCALL, D3DERR_NOTAVAILABLE, or D3DERR_OUTOFVIDEOMEMORY on 
failure.

The first three parameters, Adapter, DeviceType, and hFocusWindow, 
will be familiar to you. They are the number of the adapter you are 
using, the type of device you are using (either D3DDEVTYPE_HAL or 
D3DDEVTYPE_REF), and the window to which you will be rendering, 
respectively. One quick note about the window: if you intend to have a 
full-screen application, the window must be a top-level window.

The BehaviorFlags parameter is a set of bit flags. Table 2.1 explains 
these flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DCREATE_FPU_PRESERVE</td>
<td>Uses additional error checking, but slows down Direct3D</td>
</tr>
<tr>
<td>D3DCREATE_HARDWARE_VERTEXPROCESSING</td>
<td>Vertices will be processed using hardware</td>
</tr>
<tr>
<td>D3DCREATE_MIXED_VERTEXPROCESSING</td>
<td>Vertices will be processed using a mixture of hardware and software</td>
</tr>
<tr>
<td>D3DCREATE_MULTITHREADED</td>
<td>The application must be made threadsafe; that is, you intend for the application to run in multiple threads (often a hazardous prospect)</td>
</tr>
<tr>
<td>D3DCREATE_PUREDEVICE</td>
<td>You do not intend to use the device for geometry transformation if hardware support doesn't exist</td>
</tr>
<tr>
<td>D3DCREATE_SOFTWARE_VERTEXPROCESSING</td>
<td>Vertices will be processed using software only</td>
</tr>
</tbody>
</table>
For now, the chapter addresses three of these, D3DCREATE_HARDWARE_VERTEXPROCESSING, D3DCREATE_MIXED_VERTEXPROCESSING, and D3DCREATE_SOFTWARE_VERTEXPROCESSING. These are mutually exclusive flags, so you only have to specify one of them. They state your intent to use hardware, software, or both for processing vertices. The first half of the book doesn’t cover the vertex processing capabilities of Direct3D, so you can concentrate on D3DCREATE_SOFTWARE_VERTEXPROCESSING for the time being.

The pPresentationParameters parameter (I hate saying the word “parameter” twice in a row) is a pointer to a D3DPRESENT_PARAMETERS structure. Before calling CreateDevice, you have to fill in such a structure with information about how you want your device to present scenes. Table 2.2 briefly explains the members of D3DPRESENT_PARAMETERS.

The names of many of these members are self-explanatory, like the ones beginning with BackBuffer. One of the really cool things about Direct3D 8 is that it gives you a back buffer even if you are windowed, although it isn’t always a “true” back buffer as in DirectDraw. It might be simply a double buffer that is copied onto the screen, but Direct3D 8 hides that for you and allows you to program as though it were a flipping chain.

If you are making a windowed application, you’ll want your back buffer to be at least as big as the window to which it is rendering. You can set it to exactly the size of the client area by specifying 0 in each of these members. You’ll also want to set the format to the current display format of the system. If you’re working in full screen, you will want the width, height, and format of the back buffer to match one of the values returned by enumerating the display modes.

The back buffer count can be 0, 1, 2, or 3. If it is 0, it is treated as a 1. The examples in this book typically use the value 1.

NOTE

Not all video cards support using Direct3D 8 in a window. To check whether yours does, get the capabilities of the device type you’re using for the adapter you’re using and check Caps2 for D3DCAPS2_CANRENDERWINDOWED. If this flag is not set, you have to use full screen.
The SwapEffect member can contain one of four values: \texttt{D3DSWAPEFFECT_DISCARD}, \texttt{D3DSWAPEFFECT_FLIP}, \texttt{D3DSWAPEFFECT_COPY}, or \texttt{D3DSWAPEFFECT_COPY_VSYNC}. Specifying \texttt{D3DSWAPEFFECT_DISCARD} copies whatever is currently on the back buffer to the screen, and does not guarantee that anything useful will be on the new back buffer. This value should be used whenever you need to update the entire screen/display area every frame.
The **D3DSWAPEFFECT_FLIP** value works in the same manner as did flipping surfaces in DirectDraw. It cycles the buffers between the display and the back buffer. The other two (**D3DSWAPEFFECT_COPY** and **D3DSWAPEFFECT_COPY_VSYNC**) copy the back buffer onto the screen. **D3DSWAPEFFECT_COPY_VSYNC** does so only during the vertical retrace, to prevent shearing. These two values are best to use when you are doing dirty rectangle updates.

You’ll learn about the **EnableAutoDepthStencil** and **AutoDepthStencilFormat** parameters in Chapter 12, when you take a look at Z buffers. For now, set them to **FALSE** and **D3DFMT_UNKNOWN**, respectively.

For the **FullScreen_RefreshRateInHz**, simply set it to **D3DPRESENT_RATE_DEFAULT**. Set **FullScreen_PresentationInterval** to **D3DPRESENT_INTERVAL_DEFAULT**. Your 2D applications don’t need the same control over the display as some of the 3D applications out there.

Finally, the last parameter (**ppReturnedDeviceInterface**) is a pointer to a pointer to an **IDirect3DDevice8** object, which is filled with a pointer to a valid object if the function is successful.

At last, you can create an **IDirect3DDevice8**! Of course, you can’t do anything with it yet, but at least you have a start.

As always, when you are done with your **IDirect3DDevice8**, be sure to release it, like so:

```cpp
    g_pd3ddev->Release();
```

Examples 2.1 and 2.2 on the CD show how to create a device for both a windowed (2.1) and a full-screen (2.2) application. These examples are found in X:\Source\Example2_1 and X:\Source\Example2_2, respectively. Both of these examples report what they are doing to stdout.txt, because simply watching the application run doesn’t tell you much. Example 2.2 is a little more interesting, as it actually goes to a full screen.

## Setting up Viewports

Once you have created an **IDirect3DDevice8** object, you’re nearly ready to make it do something. You have just a few tasks left. The most important of those tasks is to set up a viewport. A **viewport** is simply the
area of the device to which you will be rendering. Without a viewport, the device does not know where to render.

Setting up a viewport is rather simple. You do it with a call to IDirect3DDevice8::SetViewport.

HRESULT IDirect3DDevice8::SetViewport(
    CONST D3DVIEWPORT8* pViewport
);

This function, as usual, returns an HRESULT, which is D3D_OK on success, and D3DERR_INVALIDCALL on failure.

The sole parameter is a pointer to a D3DVIEWPORT8 structure. This structure contains a description of the viewport prior to calling this function. Table 2.3 lists the members and their meanings.

Normally, X and Y are set to 0, Width and Height are set to the dimensions of the full screen or window (as appropriate), and MinZ and MaxZ are set to 0.0 and 1.0, respectively. This need not be the case, of course. You can set the viewport to whatever you like, as long as the dimensions fit on the screen or window.

You can even have a number of different viewports for rendering to different parts of the screen.

<table>
<thead>
<tr>
<th>Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>The viewport’s leftmost pixel coordinate</td>
</tr>
<tr>
<td>Y</td>
<td>The viewport’s topmost pixel coordinate</td>
</tr>
<tr>
<td>Width</td>
<td>The viewport’s width in pixels</td>
</tr>
<tr>
<td>Height</td>
<td>The viewport’s height in pixels</td>
</tr>
<tr>
<td>MinZ</td>
<td>Minimum Z value, typically 0.0f</td>
</tr>
<tr>
<td>MaxZ</td>
<td>Maximum Z value, typically 1.0f</td>
</tr>
</tbody>
</table>
To determine which viewport is currently in use, you can use IDirect3DDevice8::GetViewport. It looks like and is used similarly to SetViewport, except that the structure receives information.

Direct3D Colors, and Clearing a Viewport

Once you have a viewport set up and in use, you need to clear out the screen. Depending on your game, you might also need to clear it out each frame. Before you learn about the function, though, you need to understand how color works in Direct3D 8.

Most often, a color in Direct3D is described using a D3DCOLOR variable. This is a simple typedef for a DWORD. Within it is four bytes, and each byte describes a certain aspect of the color. Full alpha is 0xFF000000, full red is 0x00FF0000, full green is 0x0000FF00, and full blue is 0x000000FF.

Unlike DirectDraw, Direct3D often does the conversion from one pixel format to another for you, so you can work with color in a consistent way.

There are a few macros built to take components and build colors from them. These are D3DCOLOR_ARGB, D3DCOLOR_RGBA, and D3DCOLOR_XRGB. Here’s how you might use them:

D3DCOLOR Yellow = D3DCOLOR_ARGB ( 255 , 255 , 255 , 0 ) ;
D3DCOLOR Cyan = D3DCOLOR_RGBA ( 0 , 255 , 255 , 255 ) ;
D3DCOLOR Magenta = D3DCOLOR_XRGB ( 255 , 0 , 255 ) ;

Now that you can represent color, you can clear out the back buffer with IDirect3DDevice8::Clear.

HRESULT IDirect3DDevice8::Clear(
    DWORD Count,
    CONST D3DRECT* pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,

NOTE
The D3DCOLOR_XRGB macro does not truly ignore the alpha value. It places 0xFF into it.
DWORD Stencil
);

This function returns an HRESULT, which will be D3D_OK on success or D3DERR_INVALIDCALL on failure.

The first two parameters, Count and pRects, work together. If Count is 0, pRects should be NULL, and that indicates that the entire viewport is to be cleared. If Count is not 0, pRects should point to an array of D3DRECT structures. If Count is 10, there should be 10 D3DRECTs in the array. In this case, only this list of rectangles will be cleared.

A D3DRECT structure is similar to a RECT structure. Rather than left, top, right, and bottom, however, it uses x1, y1, x2, and y2 for its members. The meaning of these members is the same as the corresponding RECT members: x1 corresponds to left; x2 corresponds to right, y1 corresponds to top, and y2 corresponds to bottom.

The Flags parameter is a combination of bit flags. The flags are D3DCLEAR_STENCIL, D3DCLEAR_TARGET, and D3DCLEAR_ZBUFFER. When you get to the discussion of depth buffers in Chapter 12 you’ll learn about some of these other flags. For now, you only need D3DCLEAR_TARGET.

The Color parameter is a D3DCOLOR value, specifying which color you want to use to clear the target.

The Z and Stencil parameters have to do with the depth/stencil buffer; you’ll learn about them in Chapter 12.

Presenting the Scene

Clearing out the viewport is normally the first thing you do each frame. On the other end of the spectrum, presenting the scene is the last thing that you do. Presenting the scene is accomplished with IDirect3DDevice8::Present.

HRESULT IDirect3DDevice8::Present(
    CONST RECT* pSourceRect,
    CONST RECT* pDestRect,
    HWND hDestWindowOverride,
    CONST RGNDATA* pDirtyRegion
);
This function is roughly analogous to the Flip member function of IDirectDrawSurface7, except that it works even when you’re using a window. It returns an HRESULT, which is D3D_OK on success, or D3D_INVALIDCALL or D3D_DEVICELOST on failure.

You can present only a portion of the display by sending non-NULL pointers to RECTs in pSourceRect and pDestRect. You can also present to a window other than the main window by using a non-NULL value in hDestWindowOverride. The pDirtyRegion parameter is not used, and should be NULL.

Example 2.3 on the CD (X:\Source\Example2_3) is a workspace containing a fully functional, if incomplete, game. It illustrates the use of IDirect3DDevice8::Clear, IDirect3DDevice8::SetViewport, and IDirect3DDevice8::Present. It’s not very complicated (it’s a clone of Lights Out), but it is a game, one based on the amount of Direct3D you have read about thus far.

This example stresses the mission of this book: to think outside of the box when using Direct3D. That is, there are ways that Direct3D was intended to be used, but you don’t have to use it in just those conventional ways. Creating a game using nothing but IDirect3DDevice8::Clear is just one way to be unconventional. You could easily set up a number of D3DRECTs so that there was one for each pixel and then use IDirect3DDevice::Clear to set the individual pixels. You could make a complete game in this manner.

Figure 2.1 shows the initial board. It’s blocky, of course, but there is only so much you can do with the Clear function.

You can also take a look at Example 2.4 (X:\Source\Example2_4). It’s a complete program that is a simulation of Conway’s The Game of Life. It also runs in a window and uses IDirect3DDevice8::Clear as its sole method of rendering. Figure 2.2 shows a sample output of Example 2.4.

## Resetting the Display

Users of your application will naturally expect quite a bit out of it. One of the modern “must have” features is the capability to change the resolution of the display, or the capability to switch from full-screen to windowed mode at the press of a key (or the selection of a
Figure 2.1  A gross misuse of Direct3D, but it’s still a game

Figure 2.2  The Game of Life, \textit{sans} Milton Bradley
menu item). This was a pain in the neck when using previous versions of DirectX—you had to release all of the objects that handled rendering, then rebuild all of the objects with the new parameters.

Resetting the display remains something of a pain in the neck in DirectX 8.0, but it’s a little easier because of the IDirect3DDevice8::Reset function.

```c
HRESULT IDirect3DDevice8::Reset(
    D3DPRESENT_PARAMETERS* pPresentationParameters
);
```

This function returns an HRESULT, which is D3D_OK on success, or an error code (D3DERR_INVALIDCALL, D3DERR_OUTOFVIDEOMEMORY, or E_OUTOFMEMORY) on failure. The sole parameter is a pointer to a D3DPRESENT_PARAMETERS structure. You can fill it out just like you do when using IDirect3D8::CreateDevice.

Even if you don’t support multiple display modes, you might need to make use of Reset if the device goes into a lost state. This happens when IDirect3DDevice8::Present returns D3DERR_DEVICELOST. Typically, you’ll get this return code when the user presses Alt+Tab on a full-screen application.

When you first detect the lost device, you should immediately release all of the assets other than the device itself and those assets that exist in system memory. You should then move into a paused state and periodically check the return value of IDirect3DDevice8::TestCooperativeLevel.

```c
HRESULT IDirect3DDevice8::TestCooperativeLevel ( ) ;
```

This function returns one of three values: D3D_OK, D3DERR_DEVICELOST, or D3DERR_DEVICENOTRESET. If you have already encountered a D3DERR_DEVICELOST from a call to IDirect3DDevice8::Present; you won’t be getting a D3D_OK code until after you reset the device.

**NOTE**

Any other assets your application might have (textures, surfaces, and so on) do need to be released and recreated when you reset the display. You just don’t have to release and create the device again.
As long as you are getting D3DERR_DEVICELOST from TestCooperativeLevel, you should keep the application in the paused state. Once you receive a D3DERR_DEVICENOTRESET you can then reset the device and recreate all other assets.

**Summary**

The IDirect3DDevice8 object is the most important object in a Direct3D 8 application. That fact might not be as obvious right now, because you have only learned a limited subset of what it can do. At this point, you have the ability to create some rudimentary games using Direct3D. In the next couple of chapters, you’ll expand on these basics with some real knowledge.
CHAPTER 3

VERTEX FORMATS
Once you have an IDirect3D8 and IDirect3DDevice8 object set up, it's nearly time to begin rendering. Before really getting into rendering, however, there is one last topic you need to understand: how to format information so that Direct3D can use it to render.

This chapter covers vertex formats. A vertex is nothing more than a point, usually in the context of other points that together make up a polygon. This chapter is about how to represent your vertices so that Direct3D can use them.

By the end of this chapter you will:

- Understand the two main purposes of Direct3D
- Understand how to use the flexible vertex format to represent vertices
- Know the difference between transformed and untransformed vertices
- Know what vertex buffers are and how to use them
- Know what index buffers are and how to use them

The Purpose of Direct3D

I’m big into determining exactly what an API does for me, so that I have no question about what I can do with it. When using an API, I work out for myself a simple definition of what it does. For example, DirectDraw manages video memory as blocks of rectangles, and allows you to copy rectangles from one block of video memory to another. My simplistic definition is that DirectDraw’s purpose is to copy rectangular blocks of pixels. Naturally, this definition doesn’t cover everything that DirectDraw does, but it does explain the essence of the API’s purpose.

If you ask a person who has used Direct3D what its purpose is, you’ll get a barrage of terms like z buffer, model, mesh, vertex shader, pixel
shader, texture, and alpha in response. It seems to do a lot of things. With a few exceptions, most of these things are enhancements or refinements of the two primary purposes, geometry transformation and polygon rendering. My simplistic way of looking at it: Direct3D manipulates points and draws triangles. Of course there is more to it than that, but if you keep these two simple purposes in mind, you are less likely to become confused.

The fact of the matter is that although Direct3D was intended to write 3D scenes, it does not actually do so. The screen on my monitor is still two-dimensional (slightly curved, if you want to be a stickler about it) and essentially has no depth. The games in existence claiming to be 3D are really only clever approximations of 3D scenes projected onto a 2D monitor screen.

So, if it's 2D anyway, why not make use of the specialized hardware for creating 2D games?

Representing Vertex Data

For the time being, this chapter will ignore the first purpose of Direct3D (geometric transformation), and the examples will show the manipulation done manually. As 2D programmers, this is what you are used to. Through the remainder of this chapter, you will learn how to represent vertex data so that Direct3D can work with it.

There are a number of ways to represent vertex data, and the developers of Direct3D have come up with a way to accommodate all of them. Some vertices need more information than others, and some need different information than others.

The way to describe and represent vertex data for Direct3D is called the flexible vertex format, normally abbreviated as FVF in documentation. Table 3.1 shows all of the information you might want to store in a vertex.

A number of these items will be alien to you at this point. You won't use some of them, such as blending weights and vertex point size, in this book because they aren't necessary for the tasks at hand. Some of the others are discussed now, but not used until later chapters. A scant few are discussed in depth in the next sections.
Position

The most important part of a vertex is its position. Vertex position is necessary for every type of vertex you will deal with. This is true when you are manipulating your own vertices, or when Direct3D is manipulating them for you. Without a position, a vertex is meaningless.

If Direct3D is manipulating vertices for you, and for some reason you wanted to make vertices with nothing but the position information, this is how you would do it in code:

```
#define CUSTOM_VERTEX_FVF D3DFVF_XYZ
struct CustomVertex
{
    float x ;
    float y ;
    float z ;
} ;
```
The macro `CUSTOM_VERTEX_FVF` is simply a combination of flexible vertex format flags (they all start with `D3DFVF_`) that describe the vertex data you intend to use. You later give this value to Direct3D so that it knows what kind of vertices you are using.

The `CustomVertex` struct is defined by you, based on which flexible vertex format flags are present in the macro. The `D3DFVF_XYZ` flag specifies that you’re using untransformed vertices (that is, you want Direct3D to manipulate them for you), and Direct3D expects that there are three floats at the very beginning of the struct containing vertex information.

**RHW**

RHW stands for “reciprocal of the homogenous W.” If you’re manipulating your own vertex data without Direct3D’s help (using transformed vertices), you need RHW as part of your vertex information.

So, what the heck is W? It’s just another part of a coordinate, just like X, Y, and Z. Naturally, adding a W to the end of the list will give us four values, and thus a four-dimensional coordinate. Mainly, W is used to make matrices easier to work with.

RHW is not used in a vacuum. It requires a position to be useful. To create a vertex formation that uses RHW, this is the code you use:

```c
#define CUSTOM_VERTEX_FVF D3DFVF_XYZRHW
struct CustomVertex
{
    float x;
    float y;
    float z;
    float rhw;
} ;
```

As you can see, the float for `rhw` follows the position data. This is an important feature of the flexible vertex format. You can specify whatever information you like in your vertex structure, but it must be in proper order. Position is first, followed by `rhw` (if present).

If you specify `D3DFVF_XYZRHW`, you cannot specify `D3DFVF_XYZ`, because they are mutually exclusive types of vertex.
Vertex Normal

A vertex normal is a three-dimensional coordinate that describes the direction of a line perpendicular to the vertex you are rendering in relation to its polygon. This is used for lighting calculations. Although I’m not going to discuss it in depth right now, I’ll show how to make a vertex type that has a vertex normal in it.

```c
#define CUSTOM_VERTEX_FVF D3DFVF_XYZ | D3DFVF_NORMAL
struct CustomVertex
{
    float x;
    float y;
    float z;
    float nx;
    float ny;
    float nz;
};
```

Notice how the `D3DFVF` flags are combined using the `|` (or) operator. Just about any combination can be combined in this way. The vertex normal’s information is stored in `nx`, `ny`, and `nz`. You cannot use `D3DFVF_NORMAL` for a vertex that is using `D3DFVF_XYZRHW`.

**NOTE**

If you were using blending weights, you would place them in between the position of RHW and the position of the vertex normal. Although they are not used in this book, I wanted you to be aware that they do exist and understand where they go. Similarly, you place the vertex point size (a float) after the vertex normal and before the diffuse color.
Diffuse Color

After the vertex normal comes the diffuse color. The diffuse color is the color that something would reflect if light were hitting it just right. Keep in mind that 3D lighting has almost nothing to do with real lighting. It’s all just a bunch of mathematical abstractions.

Diffuse color is stored in a D3DCOLOR (such as a DWORD). Therefore, to add a diffuse color to a transformed vertex (one using D3DFVF_XYZRHW), you would do the following.

#define CUSTOM_VERTEX_FVF D3DFVF_XYZRHW | D3DFVF_DIFFUSE
struct CustomVertex
{
    float x ;
    float y ;
    float z ;
    float rhw ;
    D3DCOLOR diffuse ;
} ;

Specular Color

Specular color is stored after diffuse color. It is also a D3DCOLOR value. A specular color gives an object “shininess.” You won’t often use specular highlights, as they are computationally more expensive. Here’s what a transformed vertex type with both diffuse and specular colors looks like.

#define CUSTOM_VERTEX_FVF D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_SPECULAR
struct CustomVertex
{
    float x ;
    float y ;
    float z ;
    float rhw ;
    D3DCOLOR diffuse ;
    D3DCOLOR specular ;
} ;
Texture Coordinates

At the very end of the vertex type come texture coordinates. Textures are covered in Chapter 6, but for now, suffice it to say that textures make detailed triangles by cleverly using linear algebra to map a bit-mapped image onto a triangle.

You can have up to eight textures applied to a single vertex. Of course, not all video cards support this, so you should primarily stick to one and occasionally use two.

To further confuse this issue, textures can have one, two, three, or four dimensions to them. In this book, you will typically go with only one, but I’ll bring in a second one from time to time.

So, each texture gives you an extra set of coordinates (with between one and four dimensions). These are, of course, floats. To quickly give yourself a number of 2D texture coordinates, you can use the D3DFVF_TEXn flag, where n is the number of coordinate pairs for textures that you need. If you wanted just one set of texture coordinates, you use D3DFVF_TEX1.

If you want something other than two dimensions for a texture coordinate set, you need to use D3DFVF_TEXCOORDSIZEx(y), where x is the number of dimensions and y is the texture stage. For example, D3DFVF_TEXCOORDSIZE1(0) will make the first texture coordinate set one-dimensional. It’s important to note here that you can use this macro instead of the D3DFVF_TEXn flags. To set up the equivalent of D3DFVF_TEX1, you can specify D3DFVF_TEXCOORDSIZE2(0).

Here’s some code that demonstrates how you would put a single set of texture coordinates into your vertex struct.

```c
#define CUSTOM_VERTEX_FVF  D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
                        D3DFVF_SPECULAR | D3DFVF_TEX1

struct CustomVertex
{
    float x ;
    float y ;
    float z ;
    float rhw ;
    D3DCOLOR diffuse ;
};
```
D3DCOLOR specular;
float u;
float v;
}

In every book on Direct3D I have read, $u$ and $v$ are used for texture coordinates. I’m going to keep with this standard. Direct3D doesn’t actually care what you call them in your struct, because it reads them based on their position in the struct rather than by name.

The previous vertex type is actually the same vertex format you often use in the first half of the book.

**Telling the Device Which Format You’re Using**

After you decide which vertex format you’re using, you have to tell the IDirect3DDevice8. This is a simple matter because you’ll always define a macro CUSTOM_VERTEX_FVF or something like it that describes the type of vertex you’re using. The function to use is IDirect3DDevice8::SetVertexShader.

```c
HRESULT IDirect3DDevice8::SetVertexShader( 
    DWORD Handle 
);
```

This function returns an HRESULT, which is D3D_OK on success or an error code on failure. To make use of vertices in the standard way, simply provide the FVF description to this function, like so:

```c
g_pd3ddev->SetVertexShader ( CUSTOM_VERTEX_FVF ) ;
```

The SetVertexShader function has other uses, but for now this is all you need to know.

**Transformed or Untransformed**

From the names alone, the difference between these two types of vertices should be fairly obvious. Naturally, one is transformed already and the other is not.
What that really means is that transformed vertices are not having their positions manipulated by Direct3D, whereas untransformed vertices are. From a 2D standpoint, transformed vertices are more like what you’re used to working with, and so you’ll be using them for the rest of the first half of the book.

**Vertex Buffers**

There are a few ways to store vertex information. You can use something as simple as an array of vertices. Or, you can use vertex buffers, which is nearly the same as using an array. A vertex buffer is nothing more than a block of memory set aside for storing vertex data. Vertex buffers are different from arrays because their memory is managed by Direct3D, meaning it’s stored in a more efficient memory location. Vertex buffers are slightly more difficult to work with than arrays (and indeed, they are normally used in conjunction with an array of vertices).

There are five tasks you need to complete in order to make use of vertex buffers.

1. Create the buffer.
2. Fill the buffer with data.
3. Tell the device which buffer you want to use.
4. Render the geometry of the scene (covered in Chapter 4).
5. Destroy the buffer.

**Creating a Vertex Buffer**

Of course, you must create a vertex buffer before you can use one. The function for doing this is `IDirect3DDevice8::CreateVertexBuffer`.

```c
HRESULT IDirect3DDevice8::CreateVertexBuffer(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPOOL Pool,
    IDirect3DVertexBuffer8** ppVertexBuffer
);
```

HRESULT IDirect3DDevice8::CreateVertexBuffer(  
    UINT Length,  
    DWORD Usage,  
    DWORD FVF,  
    D3DPOOL Pool,  
    IDirect3DVertexBuffer8** ppVertexBuffer
);
This function returns an HRESULT. If successful, the returned value is D3D_OK. If not successful, it is D3DERR_INVALIDCALL, D3DERR_OUTOFVIDEOMEMORY, or E_OUTOFMEMORY.

The first parameter is the desired length of the buffer, in bytes. It has to be at least big enough to store a single vertex, but as long as that restriction is met, you don’t have to divide evenly by the size of the vertex structure. Typically, you pass sizeof(CustomVertex) times the number of vertices you want to store in the buffer.

The second parameter, _Usage_, is a set of bit flags that tells Direct3D how you intend to use this buffer. Table 3.2 shows the flags and briefly describes when to use them.

Most of the time, you’re just going to leave _Usage_ as 0. Occasionally, you’ll use D3DUSAGE_WRITEONLY.

<table>
<thead>
<tr>
<th>Flag</th>
<th>When to Use It</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DUSAGE_DONOTCLIP</td>
<td>Use this only when you do not need clipping</td>
</tr>
<tr>
<td>D3DUSAGE_DYNAMIC</td>
<td>Use when you need dynamic memory access</td>
</tr>
<tr>
<td>D3DUSAGE_RTPATCHES</td>
<td>The buffer is used for high order primitives (not covered)</td>
</tr>
<tr>
<td>D3DUSAGE_NPATCHES</td>
<td>The buffer is used for drawing n-patches (not covered)</td>
</tr>
<tr>
<td>D3DUSAGE_POINTS</td>
<td>The buffer is used for drawing point sprites (not covered)</td>
</tr>
<tr>
<td>D3DUSAGE_SOFTWAREPROCESSING</td>
<td>The buffer is used for software vertex processing</td>
</tr>
<tr>
<td>D3DUSAGE_WRITEONLY</td>
<td>Use when you need only to write to a buffer, and not read from it</td>
</tr>
</tbody>
</table>
The third parameter is the Flexible Vertex Format description for this vertex buffer. Simply supply whichever macro or constant you have defined for this value. For the vertex types defined in this chapter, this parameter is CUSTOM_VERTEX_FVF.

The fourth parameter is a D3DPOOL constant, which you’re seeing for the first time now. The POOL simply tells Direct3D where in memory to store the vertex buffer. Most of the time, you probably don’t care. The three values are D3DPOOL_DEFAULT, D3DPOOL_MANAGED, and D3DPOOL_SYSTEMMEM. The DEFAULT (wherever the video driver finds it most convenient to store objects, usually on the video card) and SYSTEMMEM (system memory) pools are pretty obvious from their names. You don’t use the MANAGED pool in this book. Most of the time, you will use D3DPOOL_DEFAULT.

The fifth and final parameter is a pointer to a pointer to an IDirect3DVertexBuffer8, which gets filled in with a pointer to a new IDirect3DVertexBuffer8 object.

Here’s what a typical call to IDirect3DDevice8::CreateVertexBuffer looks like:

```c
//create vertex buffer with 1000 vertices
IDirect3DVertexBuffer8 pvb ;
g_pd3ddev->CreateVertexBuffer ( sizeof ( CustomVertex ) * 1000 , 0 , CUSTOM_VERTEX_FVF , D3DPOOL_DEFAULT , &pvb ) ;
```

## Loading the Buffer with Data

Typically, you store vertex information in two places. One place is in an array in system memory (or some other type of array-like container). The other is in the vertex buffer itself. From time to time, you will have to move the vertex information from your array into the vertex buffer.

First lock the buffer using IDirect3DVertexBuffer8::Lock. This gives you a byte pointer to which you can write new information. You then call
IDirect3DVertexBuffer8::Unlock to unlock the buffer. Here is what the lock function looks like:

```cpp
HRESULT IDirect3DVertexBuffer8::Lock(
    UINT OffsetToLock,
    UINT SizeToLock,
    BYTE** ppbData,
    DWORD Flags
);
```

This function returns an HRESULT, which is D3D_OK on success or D3DERR_INVALIDCALL on failure.

The OffsetToLock parameter specifies where in the buffer you want to begin the lock. You can lock anywhere you like. Normally, you start the lock at position 0, which is the beginning of the buffer.

The SizeToLock parameter specifies how many bytes you want to lock. Any size may be locked. If you want to lock the entire buffer, you can use 0 for this parameter. Normally, you will lock the entire buffer.

The ppbData parameter is a BYTE**, which is set to a byte pointer that you can use to fill in vertex information.

Finally, the Flags parameter specifies any special conditions of this lock. The possible values are D3DLOCK_DISCARD, D3DLOCK_NOOVERWRITE, D3DLOCK_NOSYSLOCK, and D3DLOCK_READONLY. The meanings of these flags are fairly obvious. You’ll normally put 0 into this parameter, unless there is an overwhelming need for something else.

After the buffer is locked, the pointer returned in the ppbData parameter points to where you write the vertex data. Typically, this is through a call to memcpy.

After you are through writing the vertex data to the buffer, call IDirect3DVertexBuffer8::Unlock.

```cpp
HRESULT IDirect3DVertexBuffer8::Unlock();
```

This function returns D3D_OK or D3DERR_INVALIDCALL. It takes no parameters.
**Telling the Device Which Buffer to Use**

In order to make use of the information in a vertex buffer, you must first tell Direct3D where that information is by calling `IDirect3DDevice8::SetStreamSource`.

```c
HRESULT IDirect3DDevice8::SetStreamSource(
    UINT StreamNumber,
    IDirect3DVertexBuffer8* pStreamData,
    UINT Stride
);
```

This function returns an HRESULT, which is `D3D_OK` on success or `D3DERR_INVALIDCALL` on failure.

The first parameter is the stream number. You use zero here. However, it is possible to use up to 16 streams for vertex buffers. To determine the number of streams supported, check out the value of `D3DCAPS`'s `MaxStreams` member.

After you have set the stream source, you can officially render using Direct3D; this is covered in Chapter 4, “Rendering States.”

---

**NOTE**

When you no longer need a vertex buffer associated with a stream, set the pointer to NULL with `SetStreamSource`.

---

**Destroying the Buffer**

Finally, destroying a vertex buffer is like destroying any other DirectX object. You simply call `Release`.

```c
if ( pvb )
{
    pvb->Release ( ) ;
    pvb = NULL ;
}
```
Index Buffers

Quite often, you will have triangles that share vertices with other triangles. To save room, you want to avoid duplicating vertices in the vertex buffer. You can use index buffers for this. An index buffer is just like an array of numbers that references vertices in the vertex buffer. It’s kind of like using the vertex buffer as a palette of vertices.

For example, if you were making a square or rectangle, you would need two triangles, but only four points. Because each triangle only needs three points, two of the points are shared. Rather than fill the vertex buffer with six vertices with duplicates, you can put only four vertices into the vertex buffer, and place six entries in the index buffer.

Setting up an index buffer is a lot like setting up a vertex buffer. You follow the same steps.

1. Create the buffer.
2. Fill the buffer with data.
3. Tell the device which index buffer you want to use.
4. Render the geometry of the scene (covered in Chapter 4).
5. Destroy the buffer.

Creating an Index Buffer

An index buffer begins its life with a call to IDirect3DDevice8::CreateIndexBuffer.

HRESULT IDirect3DDevice8::CreateIndexBuffer(
    UINT Length,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DIndexBuffer8** ppIndexBuffer
);
This function returns an HRESULT. The returned value is D3D_OK if successful. If not successful, it is D3DERR_INVALIDCALL, D3DERR_OUTOFVIDEOMEMORY, D3DERR_INVALIDDATA, or E_OUTOFMEMORY.

The first parameter, Length, is the desired size of the index buffer, in bytes. The Usage parameter is a combination of bit flags, and are the same as in IDirect3DDevice8::CreateVertexBuffer.

The Format parameter is a D3DFORMAT value, but because this is an index buffer and not color information, you need to be familiar with two new D3DFMT constants: D3DFMT_INDEX16 and D3DFMT_INDEX32. Respectively, these are the formats for 16-bit and 32-bit indexes. Which one you choose is totally up to you. In this book, you will not be dealing with a large number of vertices and indexes, so you can stick to D3DFMT_INDEX16.

The Pool parameter is one of the D3DPOOL constants you learned about in the last section. You’ll use D3DPOOL_DEFAULT.

Finally, ppIndexBuffer is a pointer to a pointer to an IDirect3DIndexBuffer8 object, which contains a pointer to a new index buffer object.

### Loading an Index

You load information into an index buffer in exactly the same way you do a vertex buffer. You lock it with IDirect3DIndexBuffer8::Lock, you unlock it with IDirect3DIndexBuffer8::Unlock, and while it is locked, you copy over information with memcpy.

```c
HRESULT IDirect3DIndexBuffer8::Lock(
    UINT OffsetToLock,  
    UINT SizeToLock,    
    BYTE** ppbData,     
    DWORD Flags
);

HRESULT IDirect3DIndexBuffer8::Unlock();
```

Do these functions look familiar? They should, because the functions are exactly the same as those that work with vertex buffers. Consistency is good.
Like vertex buffers, index buffers are typically loaded from an array or array-like container. If you are using D3DFMT_INDEX16, this container is an array of WORD values. If you are using D3DFMT_INDEX32, this container is an array of DWORD values.

**Telling the Device Which Indices to Use**

To let the device know which indices you want to use, you have to give a call to IDirect3DDevice8::SetIndices.

```plaintext
HRESULT IDirect3DDevice8::SetIndices(
    IDirect3DIndexBuffer8* pIndexData
    UINT BaseVertexIndex
);
```

This function returns D3D_OK or D3DERR_INVALIDCALL, depending on success or failure.

The pIndexData parameter is a pointer to your IDirect3DIndexBuffer8 object. BaseVertexIndex is an index into the vertex buffer, telling Direct3D which vertex to treat as index zero.

If you had 32 vertices in your vertex buffer, and your index buffer used all 32, you would simply supply a 0 in BaseVertexIndex. However, if your index buffer only used 16 of them, and could use either the first 16 (starting with 0) or the second 16 (starting with 16), you could put either 0 or 16 as the BaseVertexIndex. You won’t use this too much, but it does come in handy at times.

**Destroying Index Buffers**

As always, a tiny call to Release destroys index buffers once you are done with them.

```plaintext
if ( pib )
{
    pib->Release ( ) ;
    pid = NULL ;
}
```
Summary

This chapter includes a lot of information, and it doesn’t include many code examples. This is one of the reasons people find learning Direct3D so difficult. There is a certain level of knowledge you have to have about how Direct3D represents vertices before you can do any real rendering. Luckily, you are now through it, and you can move on to something a little more exciting.
CHAPTER 4

Drawing Primitives
You’ve finally achieved a level of knowledge wherein you can actually use Direct3D to start doing the fun stuff. This chapter concentrates on drawing primitives (points, lines, and triangles). By the end of it, you should have a solid grasp of how to draw the basic primitives used in Direct3D.

Your goals for this chapter are to:

- Know how to set and get a rendering state
- Know how to begin and end a scene
- Understand the types of primitives used by Direct3D
- Know how to render primitives without a vertex buffer
- Know how to render indexed primitives without a vertex buffer or index buffer
- Know how to render primitives with a vertex buffer
- Know how to render indexed primitives with both a vertex buffer and an index buffer

Rendering States

Direct3D is often referred to as a state-based rendering machine. This means that there are a number of variables in Direct3D that you can modify depending on what you want Direct3D to do. The states might contain a Boolean value, a color value, or some enumerated type, depending on what the state affects. You change these states by using the IDirect3DDevice8::SetRenderState function.

```c
HRESULT IDirect3DDevice8::SetRenderState(
    D3DRENDERSTATETYPE State,
    DWORD Value
);
```
This function returns D3D_OK if successful or D3DERR_INVALIDCALL if not successful. The first parameter is one of the D3DRENDERSTATETYPE constants listed below. The second parameter is a value to set that render state.

D3DRS_ZENABLE
D3DRS_SHADEMODE
D3DRS_ZWRITEENABLE
D3DRS_LASTPIXEL
D3DRS_DESTBLEND
D3DRS_ZFUNC
D3DRS_ALPHAENABLE
D3DRS_ALPHABLENDENABLE
D3DRS_FOGCOLOR
D3DRS_FOGSTART
D3DRS_FOGDENSITY
D3DRS_ZBIAS
D3DRS_STENCILENABLE
D3DRS_STENCILZFAIL
D3DRS_STENCILFUNC
D3DRS_STENCILMASK
D3DRS_TEXTUREFACTOR
D3DRS_WRAP0
D3DRS_WRAP1
D3DRS_WRAP2
D3DRS_WRAP3
D3DRS_WRAP4
D3DRS_WRAP5
D3DRS_WRAP6
D3DRS_WRAP7
D3DRS_LIGHTING
D3DRS_FOGVERTEXMODE
D3DRS_LOCALVIEWER
D3DRS_DIFFUSEMATERIALSOURCE
D3DRS_FILLMODE
D3DRS_LINEPATTERN
D3DRS_ALPHATESTENABLE
D3DRS_SRCBLEND
D3DRS_CULLMODE
D3DRS_ALPHAREF
D3DRS_DITHERENABLE
D3DRS_FOGENABLE
D3DRS_ZVISIBLE
D3DRS_EDGEANTIALIAS
D3DRS_RANGEFOGENABLE
D3DRS_STENCILFAIL
D3DRS_STENCILPASS
D3DRS_STENCILREF
D3DRS_STENCILWRITEMASK
D3DRS_COLORVERTEX
D3DRS_AMBIENT
D3DRS_COLORVERTEX
D3DRS_NORMALIZENORMALS
D3DRS_SPECULARMATERIALSOURCE
As you can see, there are many render states—74, to be exact. At this point, I’m not going to explain all of them; I’ll just explain certain states as you need them. The important thing to note here is the SetRenderState function, because you’re going to be using it quite a bit.

On the flip side is IDirect3DDevice8::GetRenderState, which does the opposite of SetRenderState.

HRESULT IDirect3DDevice8::GetRenderState(
    D3DRENDERSTATETYPE State,
    DWORD* pValue
);

This function returns D3D_OK if successful and D3DERR_INVALIDCALL on failure. The first parameter is a D3DRENDERSTATETYPE constant, and the second parameter is a pointer to a DWORD that is filled in with the current value of that render state.

During this part of the book you’re not using lighting, so you should turn lighting off for Direct3D. The render state constant for lighting is D3DRS_LIGHTING, and it is a Boolean value (so it’s either TRUE or FALSE). This is how you do it:

    g_pd3ddev->SetRenderState( D3DRS_LIGHTING , TRUE ) ;

And it’s just that simple.
Beginning and Ending a Scene

Before you draw any primitives in Direct3D, you have to let the device know what you are doing. After you are done drawing primitives, you have to tell the device that you are done and are ready to present the information on the screen.

This is done with two functions, IDirect3DDevice8::BeginScene and IDirect3DDevice8::EndScene.

HRESULT IDirect3DDevice8::BeginScene ( ) ;
HRESULT IDirect3DDevice8::EndScene ( ) ;

Both of these functions return an HRESULT. If successful, the returned value is D3D_OK. If not successful, the returned value is D3DERR_INVALIDCALL. Neither function takes a parameter.

You will typically go through something like the following code at each frame.

//clear the back buffer to black
g_pd3ddev->Clear ( 0 , NULL , D3DCLEAR_TARGET , 0 , 0.0f , 0 ) ;
//begin the scene
g_pd3ddev->BeginScene ( ) ;
/*DRAWING PRIMITIVES GOES HERE*/
//end the scene
g_pd3ddev->EndScene ( ) ;
//present the scene on screen
g_pd3ddev->Present ( NULL , NULL , NULL , NULL ) ;

Naturally, if you are doing dirty rectangle updates, your code won’t look exactly like this, but it’s a fair approximation of the series of function calls needed for each frame.

Primitive Types

In Direct3D, everything is built from three basic types of primitive. These are points, lines, and triangles. A point is simple enough—it is
a single pixel on the screen. You cannot have a simpler primitive.
A line is a line segment; a series of pixels connecting two end points.
A triangle is a three-sided polygon. In this book, triangles are used
most often. Figure 4.1 shows the three basic primitive types.

In addition, you can use line strips and triangle strips. In a line strip, a
series of lines are drawn between any number of points in a “connect-
the-dots” fashion. The end point of one line segment becomes the
beginning point of the next line segment. A triangle strip has sort of a
“ribbon” effect. The first three points make up the first triangle. After
that, the last two points of the previous triangle and one new point
make up a new triangle. Figure 4.2 shows a line strip, and Figure 4.3
shows a triangle strip.

Lastly, you can use a triangle fan. A triangle fan uses one point for all
the triangles, and two consecutive points thereafter for a new piece of
the fan. In this way, it is something of a hybrid of a triangle and a line
strip. Figure 4.4 shows a triangle fan.

Figure 4.1  The three basic primitive types: the point, the line, and the triangle
Figure 4.2  An example of a line strip

Figure 4.3  An example of a triangle strip
Ready to render? You’re going to start out easy, not using vertex buffers or index buffers, and the chapter will bring them into play a little later. The first function for rendering primitives that you are going to look at is IDirect3DDevice8::DrawPrimitiveUP. The UP stands for user pointer.

HRESULT IDirect3DDevice8::DrawPrimitiveUP(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT PrimitiveCount,
    CONST void* pVertexStreamZeroData,
    UINT VertexStreamZeroStride
);

The returned value is a D3D_OK or D3DERR_INVALID call, depending on whether or not it is successful.

The first parameter is the type of primitive you want to render. There is a special enumeration for this called D3DPRIMITIVETYPE, and the val-
ues correspond to each of the types of primitives that can be drawn with this function. They are listed in Table 4.1.

The second parameter is the number of primitives you’re going to draw. This depends somewhat on the type of primitive you’re drawing. For a point list, each vertex is a single primitive, so however many vertices you have is the number of primitives you need. For a line list, two vertices make up a line, so just divide the number of vertices you have by two. For a triangle list, three vertices make up a triangle, so you divide by three. For a line strip, you need at least two vertices, and the number of primitives is the vertex count minus one. For a triangle strip or triangle fan, you need at least three vertices, and the number of primitives is the vertex count minus two. I have put this information into a more easily read form in Table 4.2.

The third parameter is a pointer to the vertex data. For this, you use an array of your vertex type and send a pointer to that array.

The last parameter is the stride of the vertex type. That’s just a fancy way of saying the size of the vertex. Here you simply pass the size of your vertex type.

If, for example, you were drawing a list of 10 points using an array of CustomVertex variables, this is what it would look like:

```cpp
g_pd3ddev->DrawPrimitiveUP ( D3DPT_POINTLIST, 10, vertarray, sizeof (CustomVertex) );
```

### TABLE 4.1 D3DPRIMITIVETYPE Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type of Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DPT_POINTLIST</td>
<td>A list of points</td>
</tr>
<tr>
<td>D3DPT_LINELIST</td>
<td>A list of lines</td>
</tr>
<tr>
<td>D3DPT_TRIANGLELIST</td>
<td>A list of triangles</td>
</tr>
<tr>
<td>D3DPT_LINESTRIP</td>
<td>A strip of connected lines</td>
</tr>
<tr>
<td>D3DPT_TRIANGLESTRIP</td>
<td>A strip of connected triangles</td>
</tr>
<tr>
<td>D3DPT_TRIANGLEFAN</td>
<td>A fan of triangles</td>
</tr>
</tbody>
</table>
Example programs? I’ve got them. I wrote six examples for this section, one for each primitive type. Table 4.3 lists the examples and which primitive type they use.

Figures 4.5 through 4.9 show the output of the various example programs.

### Table 4.2

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Number of Vertices</th>
<th>Number of Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point list</td>
<td>Any number</td>
<td>Same as the number of vertices</td>
</tr>
<tr>
<td>Line list</td>
<td>Multiple of two</td>
<td>Number of vertices divided by 2</td>
</tr>
<tr>
<td>Triangle list</td>
<td>Multiple of three</td>
<td>Number of vertices divided by 3</td>
</tr>
<tr>
<td>Line strip</td>
<td>At least two</td>
<td>Number of vertices minus 1</td>
</tr>
<tr>
<td>Triangle strip</td>
<td>At least three</td>
<td>Number of vertices minus 2</td>
</tr>
<tr>
<td>Triangle fan</td>
<td>At least three</td>
<td>Number of vertices minus 2</td>
</tr>
</tbody>
</table>

### Table 4.3

<table>
<thead>
<tr>
<th>Example</th>
<th>Primitive Type</th>
<th>Location on CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 4.1</td>
<td>Point list</td>
<td>X:\Source\Example4_1</td>
</tr>
<tr>
<td>Example 4.2</td>
<td>Line list</td>
<td>X:\Source\Example4_2</td>
</tr>
<tr>
<td>Example 4.3</td>
<td>Line strip</td>
<td>X:\Source\Example4_3</td>
</tr>
<tr>
<td>Example 4.4</td>
<td>Triangle list</td>
<td>X:\Source\Example4_4</td>
</tr>
<tr>
<td>Example 4.5</td>
<td>Triangle strip</td>
<td>X:\Source\Example4_5</td>
</tr>
<tr>
<td>Example 4.6</td>
<td>Triangle fan</td>
<td>X:\Source\Example4_6</td>
</tr>
</tbody>
</table>
Figure 4.5  Output of Example 4.1 (point list)

Figure 4.6  Output of Examples 4.2 (line list) and 4.3 (line strip)
Figure 4.7  Output of Example 4.4 (triangle list)

Figure 4.8  Output of Example 4.5 (triangle strip)
Indexed Primitives with User Pointers

You can get away with not using indexed primitives most of the time, but there are times when you’ll do better to use them. If you have a lot of vertices (which at this point you don’t), many of which are shared between primitives, it is better to use indexed primitives.

Example 4.2 draws an outlined triangle with a line list. Because there are three lines in this program, and it uses line lists, you need six vertices. However, there are only three positions, so each position is duplicated so that there are enough lines.

The custom vertex you’re using is rather light. It has as members x, y, z, rhw, and diffuse. It is 20 bytes in size. Six vertices take up 120 bytes. For the moment, vertex 0 is the same as vertex 5, vertex 1 is the same as vertex 2, and vertex 3 is the same as vertex 4. You only really need three vertices, which would take up only 60 bytes, and you could use six indexes (each 2 bytes in size for 24 bytes), so using indexed
primitives in this case does save you something. It allows you to use 84 bytes instead of 120. This might not seem very significant right now because you are only drawing a few lines. As more primitives are added, however, more space is saved.

So, how do you accomplish this? First, you make an array of vertices that contain the positions that you need. Next, you make an array of WORD values that reference that array. For this example, vertices 0, 1, and 2 are used for the triangle. You need a line between each pair, so the six indexes are 0, 1, 2, 2, 1, and 0. Each pair makes a line. If you were instead doing a line list, you would only need four indexes: 0, 1, 2, and 0, and each two consecutive indices would make a line.

Now the function that does it:

```c
HRESULT IDirect3DDevice8::DrawIndexedPrimitiveUP(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT MinIndex,
    UINT NumVertices,
    UINT PrimitiveCount,
    CONST void* pIndexData,
    D3DFORMAT IndexDataFormat,
    CONST void* pVertexStreamZeroData,
    UINT VertexStreamZeroStride
);
```

This function returns D3D_OK if successful and D3DERR_INVALIDCALL if not.

The first parameter is the primitive type, just like in a call to IDirect3DDevice8::DrawPrimitiveUP. The second parameter is the minimum index. This acts like the first parameter of the SetIndices function. The number in MinIndex is the number of the vertex that is treated as vertex 0. The third parameter is the number of vertices. The fourth parameter is a pointer to an array of indexes. The fifth parameter specifies which type of index you’re using. If this value is D3DFMT_INDEX16, pIndexData should be a pointer to an array of WORD values. If this value is D3DFMT_INDEX32, pIndexData should be a pointer to an array of DWORD values. The last two parameters are the same as the last two parameters of IDirect3DDevice8::DrawPrimitiveUP: a pointer to the vertex data and the size of an individual vertex.
Example 4.7 (X:\Source\Example4_7) on the CD is a re-write of Example 4.2, using indexed primitives. And just for fun, check out Example 4.8 (X:\Source\Example4_8). It’s a little spaceship (Asteroids style) that flies around the screen with keyboard controls. Figure 4.10 shows the output of Example 4.8.

**NOTE**

There is no point (no pun intended) in using IDirect3DDevice8::DrawIndexedPrimitiveUP to draw a point list, because each vertex is its own primitive, and indexing would actually be a waste of space.
Drawing Primitives with Vertex Buffers

Once you can draw primitives with `DrawPrimitiveUP`, learning to use vertex buffers is really easy. You just have to add a couple of steps.

For one thing, you need a vertex buffer. (Review the section “Vertex Buffers” in Chapter 3 if you need to refresh your memory on how to make a vertex buffer.) Next, load the vertex information into the buffer, explained in the “Vertex Buffers” section in Chapter 3. After that has been accomplished, set the stream source to that vertex buffer. Because you are not blending vertices, you use stream zero. Then it’s just a simple call to `IDirect3DDevice8::DrawPrimitive`.

```c
HRESULT IDirect3DDevice8::DrawPrimitive(
    D3DPRIMITIVETYPE PrimitiveType,
    UINT StartVertex,
    UINT PrimitiveCount
);
```

This function returns `D3D_OK` if successful and `D3DERR_INVALIDCALL` if not. If you’ve been paying attention to the return value discussions, you’ve probably noticed a pattern.

The first parameter is the type of primitive you’re rendering. The second parameter is what vertex to start with (you might want to use a vertex buffer to render more than one primitive type, so this parameter is handy). The last parameter is the number of primitives to draw, starting with `StartVertex`.

So, how easy is that? To see it in action, check out Example 4.9 (X:\Source\Example4_9) on the CD. This example is a remake of the “rotating star” example from Example 4.6 using vertex buffers.
Drawing Indexed Primitives with Index Buffers

The last way to render primitives with Direct3D is with both a vertex buffer and an index buffer. All the information you need to know about index buffers can be found in Chapter 3, in the section “Vertex Buffers.” Because you have already dealt with indexes in relation to IDirect3DDevice8::DrawIndexedPrimitiveUP, using an actual index buffer instead is a rather simple matter; you just have the additional step of loading the indexes into the buffer.

Once this has been done, drawing primitives is a simple call to IDirect3DDevice8::DrawIndexedPrimitive.

HRESULT IDirect3DDevice8::DrawIndexedPrimitive(
    D3DPRIMITIVETYPE Type,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);

This function returns D3D_OK on success and D3DERR_INVALIDCALL on failure, just like the other primitive drawing functions.

The first parameter is the type of primitive. The second parameter is the minimum index (vertex number) to use for this call. You might remember a similar parameter in IDirect3DDevice8::SetIndices. This number is in relation to the number given in the call to SetIndices. For example, if you put a 16 in the call to SetIndices as the minimum index, and an 8 in the call to DrawIndexedPrimitive, the vertex 24 in the vertex buffer would be treated as vertex 0. The third parameter is the number of vertices, starting from the vertex being treated as vertex 0. You cannot reference any vertex outside of this range. The fourth parameter is the starting index within the index buffer. If you had the number 10 in this parameter, rendering would begin with index 10. The last parameter is the number of primitives to be drawn. The function determines how many indexes are used to do the drawing.

Example 4.10 (X:\Source\Example4_10) on the CD is a remake of Examples 4.6 and 4.9, this time using both a vertex buffer and an index buffer.
Summary

Believe it or not, the hard part is over. You can now render points, lines, and triangles with Direct3D, which is the mainstay of this part of the DirectX SDK. From here on, you will be extending and refining this knowledge. If you can draw a triangle, you can make a game. Example 4.9 illustrates that. With a little work, you could create a full game of *Asteroids* from that example; all it needs is the ability to shoot and something to shoot at. Direct3D wasn’t as hard as you thought it would be, is it?
CHAPTER 5

Surfaces
This chapter concerns surfaces. In chapters to come you will be making use of textures, and a good understanding of Direct3D surfaces is required before you can do so. There is a part of DirectX, called D3DX, that consists of “helper” functions for Direct3D, but I don’t believe using these functions promotes understanding, so this book will not use them. Instead, you’ll learn to work with surfaces manually through code. Doing so will make you a more able Direct3D programmer.

By the end of this chapter you will:

■ Be familiar with the types of surfaces and resources that Direct3D uses
■ Know how to access a device’s back buffer
■ Know how to create an image surface
■ Know how to copy rectangular areas of one surface to another
■ Know how to load an image onto a surface
■ Know how to use a Direct3D cursor
■ Know how to make and use an additional rendering target

Resource Types

There are a number of resources that you can use in Direct3D for images and bitmaps. Fundamentally, they are all similar—they are all based on one or more surfaces that comprise them. A surface, in Direct3D, refers to an area of memory (video or system) that is treated as a rectangular area comprised of pixel data. A surface is like a bitmap, although not all surfaces contain color information. Indeed, depth/stencil surfaces contain depth and stencil information. You’ll learn about depth/stencil in Chapter 12; for now, you’ll learn only about those surfaces and resources that deal with color information.
However, in the name of completeness, all of the types of resources that you might use with Direct3D are listed in Table 5.1.

You are already familiar with vertex and index buffers and how to use them, so I won’t mention them again. Cube textures are not going to be used in this book, so they aren’t covered. Swap chains you are already somewhat familiar with, even if you don’t realize it, because your device is using a swap chain for the front and back buffers. Textures are covered in Chapter 6 (there’s a full chapter’s worth of information on them). That leaves only IDirect3DSurface8, which is the primary focus of this chapter. Swap chains are discussed a bit at the end of the chapter.

<table>
<thead>
<tr>
<th>Class</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDirect3DSurface8</td>
<td>Contains color data for pixels or depth stencil information for pixels. The basic building block of most other resources.</td>
</tr>
<tr>
<td>IDirect3DTexture8</td>
<td>Contains one or more IDirect3DSurface8 objects; used for texturing polygons</td>
</tr>
<tr>
<td>IDirect3DSwapChain8</td>
<td>Contains at least two surfaces, one front buffer and one back buffer</td>
</tr>
<tr>
<td>IDirect3DCubeTexture8</td>
<td>Contains six textures (one for each face of a cube), each of which contains one or more surfaces</td>
</tr>
<tr>
<td>IDirect3DVertexBuffer8</td>
<td>Contains vertex data</td>
</tr>
<tr>
<td>IDirect3DIndexBuffer8</td>
<td>Contains index data</td>
</tr>
</tbody>
</table>
I said earlier that the IDirect3DDevice has a *swap chain* (a primary surface and a back buffer). However, you weren’t privy to how these surfaces were created when the device was created. This is actually a good thing, because you don’t really need to know how these surfaces are created, as you just want to specify a screen size and have the surfaces created automatically. When you call IDirect3DDevice8::CreateDevice, the swap chain is created.

There are times when you’ll need access to these buffers. For example, you might have some sort of control panel on another surface that you need to copy to the back buffer before presenting it on-screen. Normally, you do not want to access the front buffer, and so you cannot do so directly. You can, however, copy the contents of the front buffer into another surface, which is explained a bit later.

To grab the back buffer, simply call IDirect3DDevice8::GetBackBuffer.

```c
HRESULT IDirect3DDevice8::GetBackBuffer(
    UINT BackBuffer,
    D3DBACKBUFFER_TYPE Type,
    IDirect3DSurface8** ppBackBuffer
);
```

This function returns either D3D_OK or D3DERR_INVALIDCALL. The first parameter is the number of the back buffer you want to view. You have only been using one back buffer so far, and so the only valid number for this parameter is 0. If you specify more than one back buffer when you create your device then you can specify any number up to the number of back buffers minus 1.

The Type parameter is rarely used. It is for future compatibility with stereoscopic devices such as VR goggles. The only valid value for this parameter is D3DBACKBUFFER_TYPE_MONO. I can only presume that there will eventually be a stereo value that can be placed here.

The final parameter is a pointer to a pointer to an IDirect3DSurface8, which is filled with a pointer to the back buffer surface. You can now use this surface to affect the contents of the back buffer.
Here’s a quick code snippet demonstrating how you would access the back buffer’s surface.

```c
//surface pointer
IDirect3DSurface8 pSurface ;
//grab back buffer
g_pd3ddev->GetBackBuffer ( 0 , D3DBACKBUFFER_TYPE_MONO , &pSurface ) ;
/*do something with surface here*/
//release the surface
pSurface->Release ( ) ;
```

As always, the theory here is “put your toys away when you are done with them.” This really applies in Direct3D.

**Image Surfaces**

If you’ve worked with DirectDraw at all, you’re undoubtedly familiar with off-screen surfaces. If your experience lies more in the use of GDI, you’ve undoubtedly created a number of memory devices into which you selected bitmaps. That’s what image surfaces are: surfaces that sit somewhere in memory and are used as storage areas for image information. They are not normally used for much other than storing and loading information.

To create an image surface, call `IDirect3DDevice8::CreateImageSurface`. Clever name, no?
HRESULT IDirect3DDevice8::CreateImageSurface(
    UINT Width,
    UINT Height,
    D3DFORMAT Format,
    IDirect3DSurface8** ppSurface
);

If successful, this function will return D3D_OK. If not, it will return D3DERR_INVALIDCALL, D3DERR_OUTOFVIDEOMEMORY, or E_OUTOFMEMORY.

The purpose of the first three parameters should be obvious just by looking at them. They are the dimensions (Width and Height) and the color format (Format) of the desired image surface.

The fourth parameter is a pointer to a pointer to an IDirect3DSurface8 object, which is filled with a pointer to the newly created image surface. Image surfaces live in system memory, without exception.

So, if you wanted to create a 640x480 surface with D3DFMT_R5G6B5 as the color format, this is how you would do so:

    //surface pointer
    IDirect3DSurface8* pSurface ;
    //create image surface
    g_pd3ddev->CreateImageSurface ( 640, 480, D3DFMT_R5G6B5, &pSurface ) ;
    /*do something with surface*/
    //release the surface
    pSurface->Release ( ) ;

Image surfaces, like every other type of object in Direct3D, have to be released when you are done with them. If you’re consistently using an image surface throughout an application, be sure to release it when you exit the program so as to avoid memory leaks.

**Copying from Surface to Surface**

If you watch me closely, you can always see that I’m building up to something. In this case, I’ve been driving you towards this section. You now know how to grab the back buffer surface, and you know how to create an image surface. The next logical step is to transfer information from one to the other. The function you use for this is
IDirect3DDevice8::CopyRects, which is like a limited feature version of the Blt function from DirectDraw, or the BitBlt function from GDI.

HRESULT IDirect3DDevice8::CopyRects(
    IDirect3DSurface8* pSourceSurface,
    CONST RECT* pSourceRectsArray,
    UINT cRects,
    IDirect3DSurface8* pDestinationSurface,
    CONST POINT* pDestPointsArray
);

This function returns D3D_OK if successful or D3DERR_INVALIDCALL if it fails.

The first parameter is a pointer to an IDirect3DSurface8 object, which can be any surface you might have in your program. The second parameter is a pointer to an array of RECTs describing the rectangular areas you want to copy from. The third parameter is the number of rectangles you are copying. The fourth parameter is another pointer to an IDirect3DSurface8 object to which you will be copying data. The last parameter is a list of POINTs that describe the upper-left corner to which each RECT in the pSourceRectsArray will be copied on the destination surface. You must have the same number of POINTs as RECTs.

As a quick example, here’s how you would copy a 640x480 surface onto the back buffer:

IDirect3DSurface8* pSurface ; //image surface
IDirect3DSurface8* pBackBuffer ; //back buffer
//create an image surface
  g_pd3ddev->CreateImageSurface ( 640 , 480 , D3DFMT_R5G6B5 , &pSurface ) ;
/*this is where you would load the image surface with data*/
//grab the back buffer
  g_pd3ddev->GetBackBuffer ( 0 , D3DBACKBUFFER_TYPE_MONO , &pBackBuffer ) ;
//set up RECT
  RECT rcSrc ;
  SetRect ( &rcSrc , 0 , 0 , 640 , 480 ) ;
//set up POINT
  POINT ptDst ;
  ptDst.x = 0 ;
  ptDst.y = 0 ;
//copy rectangle
Example 5.1 (X:\Source\Example5_1) on the CD is a small, perhaps not very interesting example of using CopyRects. It creates an image surface one quarter the size of the screen, and copies each frame onto the back buffer. Because an image surface is initially all black, the example makes the screen white before copying from the image surface. This might not look impressive right now, but it should be encouraging to know that any data on the image surface will be copied over to the back buffer in this manner. Figure 5.1 shows the output of Example 5.1.
Loading Images

In order to be useful, an image surface has to have pixel data on it. There are functions for doing so that come with the DirectX8 SDK, but you won’t be using them. Loading images with your own code is really not that hard, and doing so gives you greater control over how the image is converted from one format to another.

In this section, you’re going to deal strictly with standard Windows bitmaps, because every Windows user has an editor for this type of image. Additionally, the WIN32 API has functions for loading them, so you don’t have to make your own.

When you load in a bitmap with the `LoadBitmap` or `LoadImage` function and then select it into a device context, you can use the `GetPixel` function on the device context to get the color of any pixel in that bitmap. The pixel color is returned in a `COLORREF` variable, and has the following format:

```
00000000RRRRRRRRGGGGGGGGBBBBBBBB
```

Each character represents a single bit in the `COLORREF`. The 0’s are unused bits and are located at the eight most significant bits of the `COLORREF`. The next eight bits are for red, the next for green, and the last for blue. There are three macros for working with `COLORREFs`: `GetRValue`, `GetGValue`, and `GetBValue`. These macros extract each of the color components from a `COLORREF` into an eight-bit value from 0 to 255.

Now let’s take a look at how a `D3DFORMAT` constant represents a pixel, for instance `D3DFMT_R5G6B5`. Its format looks like this:

```
RRRRGGGGGBBBBBBB
```

Here you can see that red has five bits, green has six, and blue has five. That makes 32 (0-31) possible values for red and blue, and 64 (0-63) possible values for green. In order to build a color that `D3DFMT_R5G6B5` can recognize from a `COLORREF`, you just have to do a little bit shifting.

For the red and blue components, the destination format has five bits each, and you have eight source bits. You simply shift left by three to
convert. For the green component, you have six destination bits and eight source bits. You simply shift left by two in this case.

Once you’ve extracted and left-shifted the individual color values, you need to rebuild the pixel. The D3DFORMATs always have red, followed by green, followed by blue, from most significant bit to least significant bit. So, you can leave the blue bits as they are, and shift right the green and red bits by the appropriate number and rebuild the pixel into the destination format. You don’t have to shift the blue bits at all. You have to shift the green bits right by the number of blue bits in the color format. You have to shift the red bits right by the number of blue and green bits in the color format. You can then OR the bits together, and you have your color.

Here’s a function that will change a COLORREF into a pixel that is meaningful to a D3DFMT_R5G6B5 function:

```c
DWORD ColorRefToR5G6B5 ( COLORREF crSource )
{

    //extract color components
    DWORD dwRed  = GetRValue ( crSource ) ;
    DWORD dwGreen = GetGValue ( crSource ) ;
    DWORD dwBlue = GetBValue ( crSource ) ;
    //shift values to the left
    dwRed >>= 3 ;  //8-5=3
    dwGreen >>= 2 ; //8-6=2
    dwBlue >>= 3 ; //8-5=3
    //shift values to the right
    dwRed <<= 11 ; //5+6=11
    dwGreen <<= 5 ;
    //recombine components
    DWORD dwColor = dwRed | dwGreen | dwBlue ;
    //return the color
    return ( dwColor ) ;
}
```

And from this you could make a generic solution that converts from a COLORREF to a format that you specify by the number of red, green, and blue bits in the format.

```c
DWORD ConvertFromColorRef ( COLORREF crSource , DWORD dwRBits , DWORD dwGBits , DWORD dwBBits )
```
Now that you can convert from COLORREF to any of the formats used by Direct3D, your job is a lot simpler. You simply need to know where to put these values in order for them to be on your image surface.

You need to lock the image surface; doing so is similar to the way you transferred vertex data to a vertex buffer. You can lock a portion of an image surface (or the entire image surface) with a call to IDirect3DSurface8::LockRect.

HRESULT IDirect3DSurface8::LockRect(
    D3DLOCKED_RECT* pLockedRect,
    CONST RECT* pRect,
    DWORD Flags
);

This function returns D3D_OK if successful and D3DERR_INVALIDCALL on failure.

The first parameter is a pointer to a D3DLOCKED_RECT that is filled with the information about the locked area. This is a pretty simple structure.

typedef struct _D3DLOCKED_RECT {
    INT                  Pitch;
    void*                pBits;
} D3DLOCKED_RECT;
There are only two members. The Pitch member is the number of bytes that make up a row of horizontal pixels in the image. The pBits parameter is a pointer to a block of memory containing information about the pixel data (this is where you read and write information).

The second parameter is a RECT. It describes the area you want to lock. If you specify NULL, the entire surface is locked.

The last parameter is a combination of bit flags, or 0. The bit flags are D3DLOCK_NO_DIRTY_UPDATE, D3DLOCK_NOSYSLOCK, and D3DLOCK_READONLY. The D3DLOCK_NO_DIRTY_UPDATE flag specifies that the “dirty” region of the surface should not be updated. Don’t worry about this flag too much. D3DLOCK_NOSYSLOCK requires that the system not freeze when you are locking this surface. Because image surfaces are in system memory anyway, this isn’t a problem. Finally, you use D3DLOCK_READONLY only when you want to look at, but not write to, the information on the surface. You will typically use 0 for the Flags parameter, unless you have a good reason not to.

After you are through with writing to or reading from the image surface, you call IDirect3DSurface8::Unlock.

HRESULT IDirect3DSurface8::Unlock ( ) ;

Unlock doesn’t have any parameters. It returns a D3D_OK on success and a D3DERR_INVALIDCALL on failure. Every time you lock, you must unlock.

Now you are most of the way there. You can convert a color from a COLORREF to any format you like, and you have a place to write it to.

The pBits member of the D3DLOCKED_RECT structure is a void*. This usually isn’t very useful, so you should put its value into a BYTE* variable right away.

Next, you need to calculate the position of the pixel in relation to the upper-left corner, which has a position of zero. The position within the buffer increases by Pitch bytes for each value of y, and by the bytes per pixel for each value in x. Therefore, the location within the buffer could be calculated like so:

\[
\text{Position} = y \times \text{Pitch} + x \times \text{BytesPerPixel} ;
\]

You get the Pitch from the D3DLOCKED_RECT. The only missing component is the bytes per pixel, which you can easily calculate from the
red, green, blue, alpha, and unused bits in the pixel format. For example, D3DFMT_R5G6B5 has 5+6+5=16 bits, and D3DFMT_X8R8G8B8 has 8+8+8+8=32 bits. You divide the number of bits by eight, and get the number of bytes in the pixel format. Note that you must include the alpha bits and the unused bits in the calculation of bytes per pixel, or you will be off, and your image will look screwy.

Now that you’ve got all of the pertinent information, you can write the pixel. You should write the number of bytes according to the bytes per pixel you calculated. I typically use memcpy for this.

And once you can plot a single pixel, you can plot many pixels, meaning you now know how to load an image onto a surface.

Example 5.2 (X:\Source\Example5_2) on the CD shows how to load a bitmap onto an image surface. It then uses CopyRects to put the bitmap onto the back buffer. The image is a picture of my work area (messy, isn’t it?). Figure 5.2 shows the output of Example 5.1.

---

**Figure 5.2**  Output of Example 5.2.
Example 5.3 (X:\Source\Example5_3) is a remake of one of the games you made in Chapter 2—the *Lights Out* clone—this time using an image and *CopyRects* instead of *Clear*. It also features a stand-alone function for loading an image surface from a .bmp file. Figure 5.3 shows the output of Example 5.3.

The problem with *CopyRects* is that it doesn’t support transparency. If it did, you could stop here and make 2D games with Direct3D until the cows came home.

### Direct3D Cursors

One of the applications of image surfaces is to make cursors. You can, if you want, use standard Windows type cursors just like you can in any other Windows application. However, there are some special functions for *IDirect3DDevice8* that allow you to specify your own cursors using image surfaces.
Hardware support for Direct3D cursors might not exist on a user’s machine, so you should check the CursorCaps member of the D3DCAPS8 structure. This member is a combination of bit flags. If D3DCURSORCAPS_COLOR is specified, then high-resolution hardware support for cursors is available. High resolution is defined as a display resolution with 400 or more scan lines. If D3DCURSORCAPS_LOWRES is specified, hardware support is also available for display resolutions with fewer than 400 scan lines. There is software emulation for cursors, but if the hardware support isn’t there, you might as well just use the normal Windows cursors.

To make use of Direct3D’s cursors, you need three pieces of information. First, you need an image surface. The dimensions of this surface must be a power of two, although they can each be a different power of two. A typical value is 32. A lot of hardware can’t go higher than 32, so it’s just a good number to use. Another limitation is that the cursor cannot be larger than the screen (like you’d want that!). Additionally, the image surface must have the D3DFMT_A8R8G8B8 format. This has an alpha component to it. If the alpha component is 0xFF, the pixel will be visible. If it is 0x00, it will be transparent. No other values are valid for the alpha component of a pixel on a surface containing a cursor. You create and load the image surface in the same way you did earlier.

The second and third pieces of information you need make up the hot spot. The hot spot is the center of the cursor. If you had a 32x32 cursor and wanted to center it on the position of the mouse, you would specify 16 for the hotspot x and 16 for the hotspot y. Some cursors, like the standard Windows arrow cursor, have their hotspots closer to 0,0.

Once you have this information, you can set the device’s cursor properties with a call to IDirect3DDevice8::SetCursorProperties.

HRESULT IDirect3DDevice8::SetCursorProperties(
    UINT XHotSpot,
    UINT YHotSpot,
    IDirect3DSurface8* pCursorBitmap
);

This function returns D3D_OK on success and D3DERR_INVALIDCALL on failure. The XHotSpot and YHotSpot parameters define the hotspot for the cursor. The pCursorBitmap is a pointer to an image surface that contains the image data for the cursor.
To make the Direct3D cursor visible and to eliminate the Windows cursor, you need to add an event handling case to your window procedure for WM_SETCURSOR.

```c
    case WM_SETCURSOR :
        { //this code keeps windows from changing the cursor
            SetCursor ( NULL ) ;
            g_pd3ddev->ShowCursor ( TRUE ) ;
            return ( TRUE ) ;
        } break ;
```

This snippet of code eliminates the Windows cursor by setting it to NULL, and then shows the cursor with IDirect3DDevice8::ShowCursor. Finally, it returns from the window procedure to keep the default message handler from setting the window’s class’s cursor. You can also use IDirect3DDevice8::ShowCursor to hide the cursor by putting FALSE as the parameter. The function returns whatever the previous state of the cursor was, TRUE or FALSE.

Unlike the Windows cursors, the Direct3D cursor does not move around the screen according to mouse movements. If you want that behavior, you have to code it yourself, using IDirect3DDevice8::SetCursorPosition.

```c
void IDirect3DDevice8::SetCursorPosition( 
    UINT XScreenSpace, 
    UINT YScreenSpace, 
    DWORD Flags 
);
```

This function returns no value and takes three parameters. The XScreenSpace and YScreenSpace parameters define where on the screen you want the cursor to be. These coordinates are in relation to the entire screen, whether you are making a full screen application or a windowed one, so be sure to make use of ClientToScreen and ScreenToClient to determine exactly where you want the cursor to be. The Flags parameter should be 0 or D3DCURSOR_IMMEDIATE_UPDATE. If D3DCURSOR_IMMEDIATE_UPDATE is specified, the cursor is updated more frequently. Some hardware skips a few frames before updating the cursor. Unfortunately, there is no way of telling from the programming side, so you may want to use this flag no matter what.
Example 5.4 (X:\Source\Example5_4) on the CD demonstrates using cursors. For ease of use, I created a few functions that make working with cursors easier in d3dcursors.h and d3dcursors.cpp. I felt that the hot spot and the image should be stored together, so that’s what I did. Also, I came up with a way to store the hotspot information into a simple .bmp file. This example is a partially completed puzzle game. It’s a slide puzzle, based on the image I used in Example 5.2. If you thought my work area was messy before, you should take a look now! Figure 5.4 shows the output of this example.

Creating Additional Render Targets

A render target is nothing more than a surface to which the IDirect3DDevice8 object draws primitives. When you initially set up your device, it creates a render target for you, and that might be all you ever need. You can create and use other surfaces for render tar-
gets as well, if you so desire. You can swap out the render target of the
device from one to another. In any case, being able to swap rendering
targets is a handy thing to know how to do.

To create a new render target, use IDirect3DDevice8::CreateRenderTarget. I like functions that are named for what they do, don’t you?

HRESULT IDirect3DDevice8::CreateRenderTarget(
    UINT Width,
    UINT Height,
    D3DFORMAT Format,
    D3DMULTISAMPLE_TYPE MultiSample,
    BOOL Lockable,
    IDirect3DSurface8** ppSurface
);

If successful, the function returns a D3D_OK. If not, it returns a D3DERR_INVALIDCALL. The parameter list looks a lot like the CreateImageSurface
function, with a few extras. The Width, Height, and Format parameters
specify the dimensions of the render target. The ppSurface parameter
is filled in with a pointer to the newly created surface.

The extra parameters, MultiSample and Lockable, should normally be
set to D3DMULTISAMPLE_NONE and FALSE, respectively. In this book you
aren’t ever going to use multisampling, and a lockable render target
will often incur a performance hit on substandard hardware, so don’t
fall into that trap. Use image surfaces and CopyRects to transfer data to
the rendering target rather than requiring the ability to lock it.

Once you have the render target created, you can set the device to use
it with IDirect3DDevice8::SetRenderTarget.

HRESULT IDirect3DDevice8::SetRenderTarget(
    IDirect3DSurface8* pRenderTarget,
    IDirect3DSurface8* pNewZStencil
);

Looks simple enough, doesn’t it? This function returns D3D_OK if success-
ful and D3DERR_INVALIDCALL if not. The pRenderTarget parameter is a
pointer to a surface that can be used as a render target, or NULL. If it is
NULL, the render target does not change for the device. The pNewZStencil
is a pointer to a surface to be used as a depth/stencil surface, or NULL.
If NULL, there will be no depth/stencil surface. You’ll learn more about this in Chapter 12, “Z Buffering.” For now, you will just use NULL.

What about the old render target? Well, if you plan on restoring it later, you should retrieve it before calling SetRenderTarget. The function for doing so is IDirect3DDevice8::GetRenderTarget.

HRESULT IDirect3DDevice8::GetRenderTarget(
    IDirect3DSurface8** ppRenderTarget
);

This function returns D3D_OK or D3DERR_INVALIDCALL. The ppRenderTarget parameter contains a pointer to the current render target. This increases the reference count. When you later restore this surface as the render target, you should afterwards Release it. Here’s a small snippet of code for creating, swapping out, and restoring render targets.

```c
//create new render target
IDirect3DSurface8* pTargetNew;
g_pd3ddev->CreateRenderTarget( 640, 480, D3DFMT_A8R8G8B8, D3DMULTISAMPLE_NONE, FALSE, &pTargetNew );

//retrieve old render target
IDirect3DSurface8* pTargetOld;
g_pd3ddev->GetRenderTarget( &pTargetOld );

//set new render target
g_pd3ddev->SetRenderTarget( pTargetNew, NULL );

//do something with the new render target here*/

//restore old render target
g_pd3ddev->SetRenderTarget( pTargetOld, NULL );

//release old render target
pTargetOld->Release( );

//release new render target
pTargetNew->Release( );
```

Uses for additional rendering targets vary. You might want to make one so you can draw a heads up display or a control panel for a game. These sorts of things only need to be drawn one time, and so you might draw it at the start of the game and use CopyRects to bring it to the back buffer. Another really good use of an additional rendering target is a mini-map (a shrunken version of a world map).
Summary

The IDirect3DSurface8 object is very important in Direct3D. It comprises just about everything you will be using in Direct3D, including textures, which you are going to learn about in Chapter 6, “Textures.” Surfaces give you bitmaps, and bitmaps give you realism, and realism is what you want. And they aren’t really that hard to work with, so that’s a good thing.
In Direct3D, a texture is a resource used to apply bitmapped images onto primitives. The programs we did with image surfaces in Chapter 5 are nice, but you don’t want to be limited to rendering solid bitmap blocks. If you did, you would still be using GDI or DirectDraw rather than Direct3D. Let’s get started.

Your goals for this chapter are as follows:

- Know how to create a texture
- Know how to load a texture with a bitmap
- Understand how the texture coordinate system works
- Know how to apply a texture to a primitive
- Understand how texture address modes work
- Know how to do multi-texturing

Creating a Texture

Like with all objects in Direct3D, the first step in using textures is knowing how to create one. The function for creating a texture is IDirect3DDevice8::CreateTexture.

```c
HRESULT IDirect3DDevice8::CreateTexture(
    UINT Width,
    UINT Height,
    UINT Levels,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DTexture8** ppTexture
);
```

If successful, this function returns D3D_OK. If it fails, one of the following values is returned: D3DERR_INVALIDCALL, D3DERR_OUTOFVIDEOMEMORY, or E_OUTOFMEMORY.
Some of the parameters, such as Width, Height, Format, and Pool, should look familiar. The Width and Height must each be a power of 2, although they can be different powers of 2. The Format option can be any format you like, and is usually the same format as the screen. If you want to use a format other than the screen’s format, you should check it with IDirect3D::CheckDeviceType. The Pool, if you intend to apply this texture to primitives, should be D3DPOOL_DEFAULT. You can create textures in system memory, of course, but you can’t use them on primitives.

The Levels parameter is for MIP mapping, which is beyond the scope of this book. Specify 1 for this parameter.

The Usage parameter is a combination of bit flags. Most of the time, you will want to use 0 here. The flags for this parameter are D3DUSAGE_DEPTHSTENCIL and D3DUSAGE_RENDERTARGET. If you need to write to a texture using Direct3D, specify the D3DUSAGE_RENDERTARGET flag.

Finally, the ppTexture parameter is a pointer to a pointer to an IDirect3DTexture8 object, which is filled with a pointer to a new texture object.

**TIP**

Generally speaking, use square textures, such as 64x64, 128x128, and 256x256. Non-square textures are permitted, but not all hardware supports them. If you do want to use non-square textures, check the capabilities of the device. In the D3DCAPS8 structure, the TextureCaps member will have a flag called D3DPTURECAPS_SQUAREONLY set if the hardware only supports square textures.

**NOTE**

It’s best to avoid really big textures if you can help it. You can pretty much count on sizes up to 256x256 to be supported on hardware, but anything bigger is a huge question mark. For maximum compatibility, stick to 256x256 or smaller. If you want to use larger textures, check the MaxTextureWidth and MaxTextureHeight members of D3DCAPS8 for the largest size.
Loading Texture Data

When you create a texture, the image stored inside of it is blank and black, just like an image surface is when initially created. Your next task is to load it with bitmap data. This is a pretty simple task. First you must grab a pointer to the surface of the texture, and then you use CopyRects to move data from an image surface to it. To get the surface of a texture, you simply call IDirect3DTexture8::GetSurfaceLevel.

```c
HRESULT IDirect3DTexture8::GetSurfaceLevel(
    UINT Level,
    IDirect3DSurface8** ppSurfaceLevel
);
```

This function returns D3D_OK to report success and D3DERR_INVALIDCALL to report failure. The Level parameter indicates which level of the texture you want to check. Again, this is mostly for MIP mapping. Because you are using only one level on the example surface here, you can use 0 for this parameter. The ppSurfaceLevel parameter is a pointer to a pointer to an IDirect3DSurface8, which is filled with a pointer to an IDirect3DSurface8. This surface has its reference count incremented as a result of this function call, so you should release it after you are done with it.

Once you have the surface level, copying image data to it is a simple call to CopyRects. As always, the source surface should have the same format as the destination surface for CopyRects to work correctly.

Texture Coordinates

You’ll learn how to apply textures to primitives soon. Before you can do that, however, you have to understand how texture coordinates work. They aren’t quite what you might expect (they weren’t what I expected when I started to work with Direct3D).

With each primitive, you specify a pair of texture coordinates, which relates the vertex to a position on the texture. As the primitive is being drawn, Direct3D determines which pixel on the texture is applied to that point of the primitive. This is referred to as texture mapping, and it is really just an application of linear algebra.

Although you might think that texture coordinates are the positional values in the texture, this is not the case. No matter what size the tex-
ture is, the texture coordinates range from 0.0 to 1.0. Horizontally, 0.0 represents the left edge and 1.0 represents the right edge. Vertically, 0.0 represents the top and 1.0 represents the bottom. Figure 6.1 shows what this means.

**Applying Textures to Primitives**

Once you have created the texture and loaded it with image data, you can use it on primitives. You simply need to have texture coordinates in your vertex format. These are normally named U and V. The U component represents the x coordinate within the texture, and the V component represents the y coordinate within the texture. You specify these (U, V) pairs in the vertex, and Direct3D determines the rest.

However, you need to tell Direct3D which texture you are using. The function used for doing this is IDirect3DDevice8::SetTexture.

```c
HRESULT IDirect3DDevice8::SetTexture(
    DWORD Stage,
    IDirect3DBaseTexture8* pTexture
);
```

![Figure 6.1 Texture coordinates](image)
This function returns D3D_OK if successful and D3DERR_INVALIDCALL if not. The first parameter is the texture stage. There are eight of these, 0-7, but you use only one of them, 0, at this time. The second parameter is a pointer to a texture object.

After a call to IDirect3DDevice8::SetTexture, any of the vertices you draw that have a texture coordinate will be rendered with that texture. In order to use a different texture, you have to send a different texture to IDirect3DDevice8::SetTexture. In order to use no texture, you call SetTexture with NULL as the pTexture parameter.

Example 6.1 (X:\Source\Example6_1) on the CD is a quick demonstration of texture mapping. This example is just a remake of Example 4.5 (the rotating square) with a texture applied to it. One thing I want to point out is that the texture is shaded differently in each corner (in the texture image; the image is not shaded in this way). This has to do with the combination of the diffuse color component of the vertex and the pixels on the texture. It’s just nice to know that you can give a texture a color tint if you desire. If you want to see the texture as-is, always use pure white for the diffuse color. Figure 6.2 shows the output of Example 6.1.
Texture Address Modes

If you design it to be so, you can easily make it so that you never use texture coordinates beyond the 0.0 to 1.0 range. However, there are times when you might want to exceed these bounds; for example, you may want to copy the texture more than once onto a primitive. This is where texture wrapping comes into play.

Assume for example that you have a program that uses two triangles to render a square on the screen, as in Example 6.1. Instead of using 0.0 and 1.0 for all the texture coordinates, try 2.0 instead. Example 6.2 (X:\Source\Example6_2) on the CD shows what this looks like. This program is identical to Example 6.1, except that some of the texture coordinates have been changed. Figure 6.3 shows the screen output.

Basically, what is going on here is that the texture coordinate stretches from 0.0 to 2.0 along the sides of the square. When the texture mapper hits the value 1.0, it starts over with the texture from the beginning (that is 0.0), and goes again to 1.0. If you used 3.0 in the program, you would see nine copies of the texture; if you used 4.0, you would see 16, and so on.

Figure 6.3  Output of Example 6.2
Naturally, that’s not all there is to the story. You can change this behavior to mirror the texture when crossing the boundary, or you can have Direct3D not exceed the boundary and use the edge pixels or a solid color. I’ll show you all of these in just a minute. First, you have to learn about the function that makes it possible.

IDirect3DDevice8::SetTextureStageState.

HRESULT IDirect3DDevice8::SetTextureStageState(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD Value
);

This function returns a D3D_OK on success and a D3DERR_INVALIDCALL on failure.

The first parameter indicates which texture stage you are setting a state for. Up until now you have only been using texture stage 0, and for the rest of the section you will continue to do so.

The second parameter is one of the constants in the D3DTEXTURESTAGESTATETYPE enumeration. There are a number of these, but for now you’ll only use two of them: D3DTSS_ADDRESSU and D3DTSS_ADDRESSV, both of which affect the texture address mode. More of these constants are discussed in the section covering multi-texturing.

The last parameter depends on the value submitted in the Type parameter. In the case of D3DTSS_ADDRESSU and D3DTSS_ADDRESSV, the Value parameter is one of five values in the D3DTEXTUREADDRESS enumeration. These values are shown in Table 6.1.

These address modes are a lot harder to talk about than they are to show, so let’s take a look at them. The D3DTADDRESS_WRAP was already shown in Example 6.2 (it is the default), so you can start with D3DTADDRESS_MIRROR. Example 6.3 (X:\Source\Example6_3) has a program that demonstrates this address mode. The output of Example 6.3 is shown in Figure 6.4.

As you can see, when it’s mirrored, the texture repeats but is flipped vertically or horizontally. This is often a good technique when your textures are going to be side by side, but the edges have seams. In Example 6.2, you can easily see the line where the texture started to repeat. In Example 6.3, it is less obvious.
<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DTADDRESS_WRAP</td>
<td>The texture repeats across integer boundaries.</td>
</tr>
<tr>
<td>D3DTADDRESS_MIRROR</td>
<td>The texture is mirrored across integer boundaries.</td>
</tr>
<tr>
<td>D3DTADDRESS_CLAMP</td>
<td>The texture is not repeated or mirrored. Values outside of 0.0 and 1.0 are clipped to 0.0 and 1.0.</td>
</tr>
<tr>
<td>D3DTADDRESS_BORDER</td>
<td>Any value outside of 0.0 and 1.0 becomes the border color.</td>
</tr>
<tr>
<td>D3DTADDRESS_MIRRORONCE</td>
<td>Values from -1.0 to 1.0 are mirrored; other values are clamped.</td>
</tr>
</tbody>
</table>

**Figure 6.4**  Output of Example 6.3
The third and last texture you’re going to learn about (because this chapter doesn’t cover D3DTADDRESS_BORDER or D3DTADDRESS_MIRRORONCE) is D3DTADDRESS_CLAMP. When a texture is clamped, any texture coordinates that exceed the 0.0 to 1.0 range are changed to 0.0 or 1.0. This creates a “smeared” effect on the texture if the coordinate is outside of 0.0 or 1.0. Example 6.4 (X:\Source\Example6_4) shows a clamped texture. The output is shown in Figure 6.5.

Most of the time you won’t even care about the texture address mode, because you won’t often use texture coordinates outside of the 0.0 to 1.0 range. Still, there are times when the texture address mode is useful, such as when creating a terrain that needs to be seamless. Therefore, it is a good thing to understand.

**Multi-Texturing**

Get ready for some cool effects. Although using a single texture is pretty cool in itself, using more than one texture on a primitive at the same time is even cooler. You can give the illusion of light and
shadow, add colored highlights, and make textures brighter or
dimmer.

Before you really get into it, however, it’s good to understand how tex-
turing really works in Direct3D. You have seen that the pixel data on
the texture is somehow combined with the diffuse color of the vertices
to produce color-shaded pixels. How does this happen?

By default, the first texture stage is combined with the diffuse color
using modulation. Modulation is just a fancy word for multiplication.
When you take two colors and modulate them, you extract each of
their components (red, green, blue) and convert these into numbers
between 0.0 and 1.0. The value 0.0 naturally represents 0, and 1.0
represents 255, or the maximum color value for that component.

To modulate, you take two colors and multiply each component by the
component in the second color. So, if Color1 has R1, G1, and B1 as its
components (in the range 0.0 to 1.0), and Color2 has R2, G2, B2 as its
components, the resulting color will have its red component as R1*R2,
its green G1*G2, and its blue as B1*B2. This is what happens between
the diffuse color of the vertices and the pixels within the texture.

You can change this default behavior to a number of different opera-
tions. Table 6.2 shows the ones covered in this chapter. These are all
constants in the D3DTEXTUREOP enumerated type.

As you can see, these are all ore or less basic mathematical operations.
There are more texture operations, of course, but these are the sim-
plest to use and understand.

To change the behavior of a particular texture stage, you need to call
IDirect3DDevice8::SetTextureStageState. The Type parameter must be
D3DTSS_COLOROP. So, if you wanted to change the first texture stage
(stage 0) to simply use the color in the texture, you would do this:

g_pd3ddev->SetTextureStageState ( 0 , D3DTSS_COLOROP ,
D3DTOP_SELECTARG1 ) ;

When blending textures, the two arguments are the current texture
stage (arg1), and whatever exists from a previous stage (arg2). For
stage 0, arg2 is the diffuse color, although you can change it (you’ll
learn about that in a moment). For now, you’ll learn about color
operations.
Example 6.5 (X:\Source\Example6_5) demonstrates each of the color operations you have learned about so far; you can change them by pressing the numbers 1 through 0. The application uses two textures: the texture in stage 0 is the one used in the chapter so far, and the texture in stage 1 is a lightmap, as shown in Figure 6.6.

Example 6.5 starts out with stage 1 being disabled. You can restore this state later by pressing 1. Pressing 2 or 3 sets the stage 1 color operation to select arg1 or arg2, so you can look at exactly what each

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DTOP_DISABLE</td>
<td>Texture blending is disabled.</td>
<td>N/A</td>
</tr>
<tr>
<td>D3DTOP_SELECTARG1</td>
<td>First parameter is the result.</td>
<td>R=R1</td>
</tr>
<tr>
<td>D3DTOP_SELECTARG2</td>
<td>Second parameter is the result.</td>
<td>R=R2</td>
</tr>
<tr>
<td>D3DTOP_MODULATE</td>
<td>The two parameters are multiplied together.</td>
<td>R=R1*R2</td>
</tr>
<tr>
<td>D3DTOP_MODULATE2X</td>
<td>The two parameters are multiplied together, and the product is multiplied by 2.</td>
<td>R=(R1*R2)*2</td>
</tr>
<tr>
<td>D3DTOP_MODULATE4X</td>
<td>The two parameters are multiplied together, and the product is multiplied by 4.</td>
<td>R=(R1*R2)*4</td>
</tr>
<tr>
<td>D3DTOP_ADD</td>
<td>The two parameters are added together.</td>
<td>R=R1+R2</td>
</tr>
<tr>
<td>D3DTOP_ADDSIGNED</td>
<td>The two parameters are added together, and 0.5 is subtracted from this sum.</td>
<td>R=R1+R2-0.5</td>
</tr>
<tr>
<td>D3DTOP_ADDSIGNED2X</td>
<td>The two parameters are added together, and 0.5 is subtracted from this sum. Then, this difference is multiplied by 2.</td>
<td>R=(R1+R2-0.5)*2</td>
</tr>
<tr>
<td>D3DTOP_SUBTRACT</td>
<td>The second parameter is subtracted from the first.</td>
<td>R=R1-R2</td>
</tr>
</tbody>
</table>
texture looks like by itself. Pressing 4, 5, or 6 selects the modulation, modulation 2x, or modulation 4x color operations. Pressing 7, 8, or 9 sets the addition, signed addition, and signed addition 2x operations. Pressing 0 selects the subtraction operation. All in all, this is a pretty neat example, and you can see how you can use the color operation to simulate a lighting effect (especially with modulation or addition).

Another aspect of multi-texturing that you might want to modify is the meaning of each of the color arguments. You use the values D3DTSS_COLORARG1 and D3DTSS_COLORARG2 as the Type parameter for IDirect3DDevice8::SetTextureStageState. The values that you can set for these stage states are shown in Table 6.3.
From these values you can construct any sort of multi-texturing you like. You can, for example, modulate the diffuse color with the specular color, add one texture, and subtract another texture. The result might look weird, but you can do it. This is one of those techniques with which you will want to experiment. You can even do a crude kind of bitmasking using color operations and color arguments, although making use of the bitmasking is more trouble than it is worth. You’ll learn a good way to bitmask in the next chapter.

**Summary**

You now have a solid grasp of textures and how to use them. They are relatively simple, really. Now you can add realism with bitmaps to your Direct3D applications. That easily beats using colored triangles! Next up, you’re going to look at alpha testing and learn how to use Direct3D to do something that you might have done in 2D: partially transparent rendering.
CHAPTER 7

Alpha Testing
This chapter covers animation, in the 2D sense of the word, as well as 2D transparency. In GDI, you create 2D transparency using bit-masks. In DirectDraw, you use a color key. In Direct3D, neither of these things exist, so you have to get creative.

Your goals for this chapter are to:

- Understand how alpha testing works
- Know how to load a texture for use with alpha testing
- Know how to load a number of textures from a single image

**Alpha Testing**

You learned a bit about the alpha component of a pixel in Chapter 5, when you learned how to make a cursor. Alpha testing is similar to what you did with the cursors.

*Alpha testing* refers to how Direct3D tests the alpha component of each pixel in a texture before rendering it. For example, you might set a zero alpha so that it represents transparent sections of the image, and set non-zero alpha components so that they are drawn. Once you set this up, you can create textures with alpha components and make the alpha component zero wherever the image is transparent. Simple, right?

In order to enable alpha testing (it is disabled by default), you need to set the render state `D3DRS_ALPHATESTENABLE` to `TRUE`, like so:

```c
g_pd3ddev->SetRenderState ( D3DRS_ALPHATESTENABLE , TRUE ) ;
```

Next you have to set up two other render states, `D3DRS_ALPHAFUNC` and `D3DRS_ALPHAREF`. The `D3DRS_ALPHAFUNC` render state tells Direct3D what sort of comparison to perform on the alpha component of a texture’s pixel, and the value sent to `D3DRS_ALPHAREF` is a value to which to compare it.
The D3DRS_ALPHAREF render state takes a value from 0 to 255 for the “reference” alpha. Because you only care that it is not zero, you can supply this render state with a 1, like so:

```
g_pd3ddev->SetRenderState ( D3DRS_ALPHAREF , 0x01 ) ;
```

Finally, the D3DRS_ALPHAFUNC render state takes one of several constants in the D3DCMPFUNC enumerated type. The constants for this enumeration are described in Table 7.1.

This isn’t the last time you’ll see the D3DCMPFUNC enumerated type. When you learn about Z buffering in Chapter 12, you’ll use it again. For now, when the table says the value, it means the alpha component of the texture in question, whereas the reference value is the value set in the D3DRS_ALPHAREF render state. If you want to render wherever the texture has an alpha value greater than zero, you simply have to specify D3DCMPGREATEREQUAL, because your alpha reference value has already been set to 0x01, like so:

```
g_pd3ddev->SetRenderState ( D3DRS_ALPHAFUNC , D3DCMPGREATEREQUAL )
```

Once these three render states are set to their appropriate values you’re nearly set to render using alpha testing. The only thing left is to find a format for your texture that is allowed for rendering to the back buffer (or whatever you are using as a render target), and then to perform the actual rendering.
But what about those textures without alpha components, such as D3DFMT_R5G6B5? Can they be used during alpha testing? The answer is yes. If there is a lack of an alpha component on a surface, all pixels on that surface are assumed to have an alpha of 255, so textures without alpha components are drawn as usual.

### Table 7.1 D3DCMPFUNC Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DCMP_NEVER</td>
<td>The test never passes.</td>
</tr>
<tr>
<td>D3DCMP_LESS</td>
<td>The test passes if the value is less than the reference value.</td>
</tr>
<tr>
<td>D3DCMP_EQUAL</td>
<td>The test passes if the value is the same as the reference value.</td>
</tr>
<tr>
<td>D3DCMP_LESSEQUAL</td>
<td>The test passes if the value is either equal to or less than the reference value.</td>
</tr>
<tr>
<td>D3DCMP_GREATER</td>
<td>The test passes if the value is greater than the reference value.</td>
</tr>
<tr>
<td>D3DCMP_NOTEQUAL</td>
<td>The test passes if the value is not equal to the reference value.</td>
</tr>
<tr>
<td>D3DCMP_GREATEREQUAL</td>
<td>The test passes if the value is greater than or equal to the reference value.</td>
</tr>
<tr>
<td>D3DCMP_ALWAYS</td>
<td>The test always passes.</td>
</tr>
</tbody>
</table>

### Loading an Image as a Texture

The code for loading an image as a texture is similar to both the image surface loading code and the cursor loading code. You cannot use the image surface loading code because it doesn’t take into consideration the alpha component, and you cannot use the cursor loading code.
directly because it only loads a surface that has a D3DFMT_A8R8G8B8 format. So, you have to write a new function for this.

The first step in this process is to load an image surface of the appropriate format. Because you now have an alpha component, you have to account for it in the loading process. As you did with image surface loading, you will load in a bitmap from a file using GDI functions and scan convert from there. Wherever you have the transparent pixel color, you will place a 0 alpha. Everywhere else, you place a 255. This alpha value is bit-shifted into the appropriate format before it is recombined with the red, green, and blue. After the image surface is loaded, a simple call to CopyRects will get the bits of the image onto your texture.

Example 7.1 (X:\Source\Example7_1) on the CD illustrates this concept. The function for loading a surface for use with alpha testing is in the LoadIconTexture function within d3dicon.h and d3dicon.cpp. This example uses the same type of bitmap as was used in the cursor-loading functions, with an extra row of pixels at the top and an extra column of pixels at the bottom. I am not currently using the centering information, but the upper-left corner pixel does contain the transparent color that the function scans for and places a 0 alpha at that point. Figure 7.1 shows the output of Example 7.1.

Figure 7.1  Output of Example 7.1
It’s amazing how easy this stuff is when you know how to load an image surface, eh? Now you have transparency, which means you are equipped to make any of the games that you made when using 2D. You have the full set of operations that you used for blitting, and you even have a more powerful set of tools, because you can arbitrarily rotate the textures.

### One Image, Many Textures

Making a bitmap for each texture works, but it’s hardly ideal. It would be nice if you had a set of functions or a class that would take a file containing a number of images and place them into textures for later use. Even better, this class could provide pixel and texture coordinates for you to use when rendering them.

Due to the varying format of vertices, it is difficult to provide a general solution for rendering these types of images. However, because this technique only works with transformed vertices, and they only deal with one texture at a time, you can use a format that has x, y, z, RHW, a diffuse color and a single pair of texture coordinates, and wrap this into the class.

If you have done any amount of 2D programming using tiles and sprites, you are almost certain to have used graphical templates. A graphical template, in the normal sense of the word, is nothing more than a black image divided into white boxes (the actual colors vary). This works, but in order to make use of them you must hardcode the size of the boxes that contain the pertinent images. I’ve never been a big fan of hardcoding anything, so I came up with my own scheme for encoding information in a bitmap.

First, I decide which colors to use to indicate transparent color, and then I choose four other “formatting” colors. Because images are likely not going to be the same size as the rectangular cell in which they are contained, there has to be a way to describe what part of the cell they are in. Also, I like to be able to reference an image based on a single coordinate. That is, tell the program to render an image at location (100,100) and have it roughly centered on that position automatically if I want. The five colors represent the transparent color, a
border color, a center color, an inside color, and a inside center color. These five colors are placed on the right edge of the bitmap. Figure 7.2 shows the upper-right corner of such an image. Table 7.2 lists the control colors and my usual values for them.

For the transparent color, I tend to favor magenta over black, because I like to use black in the image itself, and magenta is almost never used.

<table>
<thead>
<tr>
<th>Position</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transparent Color</td>
<td>Magenta or Black</td>
</tr>
<tr>
<td>Border Color</td>
<td>White</td>
</tr>
<tr>
<td>Center Color</td>
<td>Blue</td>
</tr>
<tr>
<td>Inside Color</td>
<td>Green</td>
</tr>
<tr>
<td>Inside Center Color</td>
<td>Cyan</td>
</tr>
</tbody>
</table>

Figure 7.2 Controls colors (in black and white!)
So, you have these control colors, but what do they all mean? This is one of those times when it is easier to show how this works rather than explain it. Consider for example Figure 7.3.

The cell itself is much larger than the image it contains, and the rest is transparent. Only the bracketed areas are actually within the image. So, you place the inside color along the top and left edges of the cell to indicate where the image is. Nifty, no?

To determine how large the cell is, place a pixel of the transparent color at the upper-left corner of the cell. This is only actually necessary at the top and left sides of the entire image, as shown in Figure 7.4.

Finally, the center or inside center colors indicate the image’s center point, even though the point might not be the physical center. The idea here is similar to the hotspot of a cursor. If you render the image at a given (x, y) coordinate, that coordinate will correspond to the center point of the image. The difference between the center and inside center depends on whether the center point lies within the bounds of the image (it might not, though it normally does).

This leaves you with a final task: to create the class. I call the class CSpriteSet.

- Upon construction, the class must load an image surface specified by a file name, and scan said image into the component textures as well as determine the relative vertices use for each of the images. This will make use of a triangle strip for each image.

![Figure 7.3](image.jpg)  
*Figure 7.3  A single cell image*
The user of the class must be able to set a few variables for rendering these images. These include diffuse color (one for each of the four vertices in the rectangle), a rotation value, and scaling values for x and y.

The user of the class must be able to render an image of the sprite set onto a device by giving only a single x and y value. The class should then render the image, after applying the rotation and scaling values.

Upon destruction, the class should clean up all of the textures it used.

If this class were making use of a 2D API, your job would be a little simpler. In Direct3D, you have to put each image onto its own texture, and each texture needs a width and height that is a power of two, so you have to calculate this from the width and height of the images you’re using.

I have implemented the CSpriteSet class in Example 7.2 (X:\Source\Example7_2). This example demonstrates all of the features of the
CSpriteSet class. There are four simple images (a circle, an ellipse, a triangle, and a rectangle), which you can cycle through using the Q and E keys. You can move the sprite around the screen using the arrow keys and change the y scaling with W and S. You can change the x scaling with A and D, and finally, you can rotate the images with the Z and X keys.

By using the CSpriteSet class (or something similar), you can easily make use of 2D images with Direct3D, which was one of the main goals of the chapter. Of course, at this point, you’re only making use of transformed vertices. In Chapter 11, you’ll learn how to use a fuller set of Direct3D’s capabilities.

**Summary**

This chapter comprises a good set of tools that will start you on the road to being a 2D programmer in Direct3D. No longer will you be intimidated by the complexity of Direct3D. As you might have noticed by now, it isn’t that difficult. Of course, you’ve only really scratched the surface, and there is much to follow.
8 Coordinate Spaces

9 Vectors

10 Matrices

11 Geometry Pipeline

12 Z Buffering
CHAPTER 8

Coordinate Spaces
Until now, you have been learning only about transformed coordinates. I’ve so far left out one of the primary purposes of Direct3D: geometry transformation. This chapter gives you a leg up on this topic by covering coordinate spaces. In previous chapters you have dealt with coordinates as pixel coordinates on the screen, but starting now you’ll begin looking at the concept of coordinates in a different way, one that will give you more power and flexibility when it comes to rendering.

By the end of this chapter you will:

- Understand the four primary coordinate spaces
- Understand what is meant by model space
- Understand what is meant by world space
- Understand what is meant by view space
- Understand what is meant by cube space
- Understand, in an abstract way, how Direct3D changes from one coordinate space to another

The Coordinate Spaces

Generally speaking, you’ll use four coordinate spaces. The names of the spaces and their general purposes are explained in Table 8.1.

Each of these spaces is explained more thoroughly in their respective sections.

Model Space

The first coordinate space is model space. This is the coordinate system in relation to the center of an object. *Center* doesn’t necessarily refer to the actual center of the object, although this is often the case. The center of mass (versus the center of volume) is another oft-used center. It’s an abstract thing. Generally speaking, you want the origin
of model space to be the point around which the model or object is rotated.

For example, if you were making a box in 3D, you might want it to be 1.0 units in each dimension; if so, you would use the coordinates (-0.5, 0.5, 0.5) and (0.5,0.5,0.5) for opposing corners, with the rest of the corners having values of 0.5 or 0.5 for x, y, and z. The center would then be at (0,0,0), as shown in Figure 8.1.

Note that the coordinates (the 0.5s and 0.5s) don’t actually mean anything in model space. They are just arbitrary numbers. They have no relationship to pixels at this point, nor will they for several spaces. So, the model space represents individual objects with (0,0,0) as their center.

**World Space**

Whereas the model space represented individual objects, the world space represents, well, the world. *World*, in this case, means anywhere that objects can be placed in space (that’s kind of a vague definition, I admit). For an example, consider a chessboard. The squares on a chessboard might be in their own right objects with their own model spaces. You can conceptually think of them as thin rectangular solids. Similarly, each piece has its own model space. When taken as a whole,
all of the board squares and the pieces that dwell on squares collectively make up world space.

When an object moves into world space, its center (i.e. the origin in model space) is moved to a particular point in world space. The rest of the object is placed in relation to the center point. Similarly, the object can be rotated, scaled, skewed, and otherwise mutilated to get it from model space to world space.

In the example of a chessboard, you might have each tile centered on an integral (x,y) location, with the board always having a z coordinate of 0.0. There are only two square objects on the board, but these objects are each drawn several times (32 times each).

Like model space, the units in world space are arbitrary, and have no relationship to pixels. They are related only to other points in world space.
View Space

The third coordinate system is view space. This space is based upon the viewer. In the move from world space to view space, you move the entire world around “the camera.” (Note that the camera doesn’t really exist. It’s a conceptual thing.) The whole point of view space is to rotate the world space so that the viewer is at (0,0,0), and everything is rotated correctly based on the direction in which the viewer is facing.

Einstein’s theory of relativity is in play here. The theory states that if you perceive motion, you can never be sure whether you are moving in relation to the world, or the world is moving in relation to you. Both points of view are equally valid. In Direct3D, the camera moves around the world, but in reality, it is more like the world moves around the camera, giving you the same net result.

Like model space and world space, the coordinates of view space are arbitrary, and mean nothing on their own.

Cube Space

The last space covered here is cube space, which implies that the space is shaped like a cube. It is not. Cube space is shaped like a half cube.

In cube space, x and y range from -1.0 to 1.0, and z ranges from 0.0 to 1.0. See? Half a cube. In order to be shown on-screen, the coordinates must be in this range, and Direct3D doesn’t care too much how you get them there.

Once in cube space, the coordinates are mapped into the viewport. A value of -1.0 for x is mapped onto the left side of the viewport. A value of 1.0 for x is mapped to the right hand side of the viewport. A value of -1.0 for y is mapped onto the bottom of the viewport, and 1.0 for y is mapped onto the top of the viewport. So, a cube space coordinate has a relationship to pixel coordinates, but that relationship depends very much on the viewport you’re using.

One really cool benefit of this is that you can transform your scene’s geometry into cube space and use it for differently-sized viewports without it looking that much different (maybe a little grainier or finer.
due to the screen resolution, but that’s about it). This is as long as the width and height of the viewport are proportional. For instance, you might set up a game that uses a 640x480 screen, and uses the entire screen as the viewport. You could switch to 800x600 and you’d still see the same thing, with just a little more detail, or you could switch to a 320x240 and you would see a grainier version of the same thing.

Geometry Pipeline

Although most of this chapter has been about the geometry pipeline, I wanted to discuss the entire process in an abstract way here.

You start in model space, move to world space, then to view space, and finally to cube space, which then gets mapped onto the viewport. In the transition from one space to another, something happens to the coordinates. This is a transformation. Transformation types include translation (movement), rotation, scaling (making bigger or smaller), and a number of other, not as often used types.

The goal is to reach cube space, with x and y between -1.0 and 1.0 and z between 0.0 and 1.0. If ever there was a time to be mission-oriented, it is now. Cube space is the goal, and model space is what you start with (in your arrays of vertices). The rest is up to you.

So, how does it happen? In each stage of geometry transformation, Direct3D puts the coordinate through a set of four equations. These equations look something like this:

\[
\begin{align*}
  x_1 &= A \times x_0 + B \times y_0 + C \times z_0 + D \times w_0 \\
  y_1 &= E \times x_0 + F \times y_0 + G \times z_0 + H \times w_0 \\
  z_1 &= I \times x_0 + J \times y_0 + K \times z_0 + L \times w_0 \\
  w_1 &= M \times x_0 + N \times y_0 + O \times z_0 + P \times w_0
\end{align*}
\]

The A-P represent constants supplied by you. \(x_0, y_0, z_0,\) and \(w_0\) represent the coordinate in the original coordinate space (for example model space), and \(x_1, y_1, z_1,\) and \(w_1\) represent the same coordinate in the new space (like world space).

You might be thrown for a loop with the \(w_0\) and \(w_1\) parts of the coordinates. You haven’t been exposed much to the w of a coordinate thus far; it was only briefly mentioned when discussing the significance of
RHW in transformed vertices. This chapter still doesn’t talk too much about it, because you’re doing 2D, and w is more important for 3D perspective stuff.

For the purposes here, w is equal to 1, and always stays 1 throughout the process of geometry transformation. This means that there are only three equations to move from one coordinate space to another, because you can get rid of the w’s.

\[
x_1 = A \times x_0 + B \times y_0 + C \times z_0 + D \\
y_1 = E \times x_0 + F \times y_0 + G \times z_0 + H \\
z_1 = I \times x_0 + J \times y_0 + K \times z_0 + L
\]

Direct3D goes through three sets of these equations: one for the transition from model to world, one for the transition from world to view, and one for the transition from view to cube. Technically, it goes through all of them at the same time, but conceptually they are done one at a time.

**Summary**

You’ll learn about the process of the geometry pipeline in greater detail in Chapter 11. You’ll also learn in more depth how these three equations are represented in Direct3D. For now, it is important to have a good abstract understanding of how these coordinate systems work and what you’re dealing with as you move from transformed vertices to untransformed vertices. The most important thing to remember is that you start in model space, and the goal is to get to cube space. It doesn’t matter much how you get there.
CHAPTER 9

Vectors
This chapter is about vectors. Depending on your background, you might think you don’t have much need for this chapter. But because it explains how Direct3D uses vectors you should probably read it anyway, even if you do understand vectors and know how to use them. Treat it as a refresher.

Your goals for this chapter are to:

- Understand what a vector is
- Know how Direct3D represents vectors
- Know how to add vectors
- Know how to subtract vectors
- Know how to scale vectors
- Know how to determine the absolute value of a vector
- Know how to determine the direction of a vector
- Know how to normalize vectors
- Know how to rotate vectors
- Know how to calculate a dot product with vectors
- Know how to calculate a cross product with vectors
- Know how to use D3DX functions with vectors

**Vector: Defined**

From a strictly mathematical point of view, a *vector* is a quantity that has both magnitude and direction. It answers two questions: “which way?” and “how far?” This, to my mind, is a rather dry definition which doesn’t reveal what vectors can actually do for you, but it is a start.

You use vectors every day, whether or not you’re aware of it. Any mathematical operation can be seen in terms of vectors. A vector can
represent any number of dimensions (well, at least one dimension is required), and so even the simplest math problem is a vector problem. For example:

$$2 + 3 = 5$$

In this equation, two vectors are being added to create a third vector. The vectors are one-dimensional, which means that there are only two directions, positive and negative. A $+2$ vector added to a $+3$ vector will give you a $+5$ vector. As you can see, even balancing your checkbook is vector-based.

Conceptually, you can think of vectors as arrows that start at the origin and point to whatever coordinates they describe. In Figure 9.1, four 2D vectors are shown, each pointing in a different direction.

The same idea applies to vectors with more than two dimensions, but a piece of paper is only 2D, so that’s the best type of vector for me to use for demonstration purposes.

**Figure 9.1** 2D vectors
D3DVECTOR

In Direct3D, everything is set up in three dimensions. This includes the types you use for representing vector quantities. D3DVECTOR represents three dimensional vector quantities.

typedef struct _D3DVECTOR
{
    float x ;
    float y ;
    float z ;
} D3DVECTOR ;

This structure has three members, x, y, and z, each of which hold a component of the vector. This is a handy structure. You can even make use of this structure in your custom vertex structures, like so:

#define CUSTOM_VERTEX_FVF D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1
struct CustomVertex
{
    D3DVECTOR position ;
    DWORD           diffuse ;
    float           u , v ;
} ;

In this CustomVertex struct, the customary x, y, and z members have been replaced with a single member, position. Accessing x, y, or z takes a slightly longer piece of code now.

CustomVertex vert ;
vert.position.x = 0.0f ;
vert.position.y = 0.0f ;
vert.position.z = 0.0f ;

It does not affect the calls to DrawPrimitive one bit. If you work with D3DVECTORs, the assignment operation for a vertex is much shorter.

D3DVECTOR vector ;
vector.x = 0.0f ;
vector.y = 0.0f ;
vector.z = 0.0f ;
vert.position = vector ;
And so it is easy to see that D3DVECTOR does help a bit with the organization of the vertex structure. It makes the structure a bit cleaner.

**Vector Addition**

Although there are helper functions and overloaded operators for this, you will first add the vectors manually. Addition of vectors is just as simple as normal addition, except that you add the three components separately in order to make the components for the resulting vector. So, if you have three vectors—v1, v2, and v3—and you need to add v1 and v2 and store the result in v3, this is what you do:

```
//v3=v1+v2
v3.x = v1.x + v2.x ;
v3.y = v1.y + v2.y ;
v3.z = v1.z + v2.z ;
```

Conceptually and graphically, when adding vectors, you take the second vector and move it so that the arrow’s tail meets the head of the first vector’s location, and then you draw a new vector from the origin to the head of the second vector, as demonstrated in 2D in Figure 9.2.

![Figure 9.2 Vector addition](image-url)
Vector Subtraction

Vector subtraction is just like vector addition, just with a minus sign. To do vector subtraction manually, you simply take each of the components and subtract them.

\[
\text{//} \text{v3}=\text{v1}-\text{v2} \\
v3.x = v1.x - v2.x ; \\
v3.y = v1.y - v2.y ; \\
v3.z = v1.z - v2.z ;
\]

Graphically, if you have the equation \( v3=v1-v2 \), you graph both \( v1 \) and \( v2 \) with their tails on the origin, draw a line between the heads of both vectors, and put the head of the vector at the head of \( v1 \) (the vector before the minus sign). Figure 9.3 shows what I'm talking about.

Scaling Vectors

Although addition and subtraction are the most commonly used operations, you might have the opportunity to use three others—scalar multiplication, dot product, and cross product—also explained in this
chapter. The first and simplest multiplication operation is a scalar multiplication. A *scalar* is simply a number, such as 5, -16, or 2.546. It sits all by itself as a quantity. To multiply a vector by a scalar, you simply multiply each of the components of the vector by that scalar value.

```cpp
// v2 = v1 * s
v2.x = v1.x * s;
v2.y = v1.y * s;
v2.z = v1.z * s;
```

Simple, right? Conceptually, it is like adding v1 to itself s times.

**Absolute Value**

Early on in this chapter, I talked about how a vector is a quantity with both a magnitude and a direction. However, so far you have been dealing with vectors as coordinate triplets of x, y, and z, which doesn’t really tell you the magnitude or direction. There are definitely times when you’ll want to know these things.

The *magnitude* of a vector is nothing more than the length of the vector from its tail to its head. This is also called the *absolute value* of the vector. It is a scalar quantity that is always positive. To retrieve the magnitude, you square each of the components of the vector, add the squares together, and then take the square root.

```cpp
// s = abs( v )
s = sqrt( v1.x * v1.x + v1.y * v1.y + v1.z * v1.z );
```

Note that this is an application of the Pythagorean theorem, just in three dimensions. As a game programmer, one thing you might not like is the square root, because it is computationally expensive. I’m not a big fan of it either, but it’s unavoidable in many cases. Just try not to use it very often.

**Direction**

There are a number of ways to show the direction of a vector. For the purposes of this book, however, you only need to know the direction of any two of the components, xy, xz, or yz, at a time, because your primary use of direction will be to rotate around one of the axes. This
make life a little easier because you now only have to deal with a two-
 dimensional direction rather than a three-dimensional direction.

Although the example here demonstrates direction only in the xy
plane, and thus uses the x and y components of the vector, any two
components can be used.

The theory behind doing this deals with inverse trigonometric func-
tions—in particular, the inverse tangent function. Normally, you use
atan for this, but there are values for which atan returns infinity, so
you use atan2. This makes determining the angle as easy as this:

\[
\text{angle} = \text{atan2} ( v1.y, v1.x ) ; \quad \text{//determine the angle from the origin to}
\]
\[
v1 \text{ in the XY plane}
\]

Look! No complicated trigonometric equations; no huge algorithms!
Personally, I find atan2 to be one of the greatest functions you’ll find
in the C and C++ language standards.

The angle that is spit out by this function is, as always, represented in
radians rather than degrees. For the most part, you should leave it that
way, because all the other trigonometric functions also use radians.

If you take the x and y components and determine their magnitude
(ignoring the z component), like so:

\[
\text{magnitude} = \sqrt{ v1.x \times v1.x + v1.y \times v1.y } ;
\]

you now have the magnitude and direction of the xy portion of the
vector. These two values show how this portion of the vector is repre-
sented in polar coordinates. To convert back to the normal way you
are representing them, do this:

\[
x = \text{magnitude} \times \cos ( \text{angle} ) ;
\]
\[
y = \text{magnitude} \times \sin ( \text{angle} ) ;
\]

Converting back and forth between Cartesian (the normal way you
represent vectors) and polar is important in many of the operations
that you do on vectors.

**Normalizing**

Normalizing your vector is important for lighting. What this means is
that the vector has a magnitude of 1, so vectors with a larger or smaller
magnitude have to be scaled to have a magnitude of 1.
The method of doing this is pretty simple. First, determine the magnitude of the vector. Next, take the reciprocal of the magnitude. Finally, multiply by the vector components, and you've got yourself a normalized vector.

```cpp
//normalize v
mag = sqrt ( v1.x * v1.x + v1.y * v1.y + v1.z * v1.z ) ;
//find magnitude
mag = 1 / mag ;      //reciprocal
v1.x *= mag ;      //multiply
v1.y *= mag ;
v1.z *= mag ;
```

Similarly, you can make a vector of any particular magnitude by modifying this snippet of code if you needed to.

### Rotation

Another very common operation is rotation around an axis. The equations for doing so are very short and sweet, but how you arrive at them is a little less so. Because you are rotating around an axis, you only need to rotate two of the vector's components. This example uses x and y. Just keep in mind that any two vectors can be used.

So, to start with, you have an x and y value that you want to rotate by a certain angle (called \( \text{angle} \) here). It isn't very easy to rotate Cartesian coordinates. Luckily, it is easy to rotate polar coordinates. So, you first convert your x and y to their polar equivalents.

\[
X = \text{magnitude} \times \cos ( \text{direction} ) \quad //v.x \\
y = \text{magnitude} \times \sin ( \text{direction} ) \quad //v.y
\]

Once we are in polar coordinates, rotation is very easy. We simply add angle to direction.

\[
X = \text{magnitude} \times \cos ( \text{direction} + \text{angle} ) \quad //v.x \\
y = \text{magnitude} \times \sin ( \text{direction} + \text{angle} ) \quad //v.y
\]

Now you just need a couple of trigonometric identities to bail you out. It just so happens I know a few.

\[
\cos ( a + b ) = \cos ( a ) \times \cos ( b ) - \sin ( a ) \times \sin ( b ) \\
\sin ( a + b ) = \sin ( a ) \times \cos ( b ) + \cos ( a ) \times \sin ( b )
\]
And I’d like to take this moment to thank Mr. Cibrario, my high school math teacher, who taught me this formula.

So, you now substitute in:

\[ \text{magnitude} \times (\cos (\text{direction}) \times \cos (\text{angle}) - \sin (\text{direction}) \times \sin (\text{angle})) / v.x \]

\[ \text{magnitude} \times (\sin (\text{direction}) \times \cos (\text{angle}) + \cos (\text{direction}) \times \sin (\text{angle})) / v.y \]

And now get rid of some of the parentheses.

\[ \text{magnitude} \times \cos (\text{direction}) \times \cos (\text{angle}) - \text{magnitude} \times \sin (\text{direction}) \times \sin (\text{angle}) / v.x \]

\[ \text{magnitude} \times \sin (\text{direction}) \times \cos (\text{angle}) + \text{magnitude} \times \cos (\text{direction}) \times \sin (\text{angle}) / v.y \]

And now you already know that \(v.x\) is the same as \(\text{magnitude} \times \cos (\text{direction})\) and \(v.y\) is the same as \(\text{magnitude} \times \sin (\text{direction})\), so you replace the bold sections like so:

\[ v.x \times \cos (\text{angle}) - v.y \times \sin (\text{angle}) / v.x \]

\[ v.x \times \sin (\text{angle}) + v.y \times \cos (\text{angle}) / v.y \]

And that’s it! I told you these equations were short and sweet, even if getting to them isn’t. You’re going to use these equations often, so try to remember them.

**Dot Product**

The second form of vector multiplication is called the *dot product*. The dot product takes two vectors and produces a scalar value. This value is useful for determining the angle between two vectors.

The mathematical proof of a dot product is beyond the scope of this book. I’m simply going to show you a few equations for determining the dot product, and show you that, yes, they produce the value of the dot product, and we’ll leave it at that.

First equation:

\[ v1 \cdot v2 = \text{abs} (v1) \times \text{abs} (v2) \times \cos (\text{angle}) \]
The \texttt{abs(v1)} and \texttt{abs(v2)} are no big deal; they are simply the magnitudes of vectors \texttt{v1} and \texttt{v2}. \texttt{cos(angle)} refers to the cosine of the angle between \texttt{v1} and \texttt{v2}. That is the part that would take a long time to prove.

\texttt{v1 \cdot v2 = v1.x * v2.x + v1.y * v2.y + v1.z * v2.z ;}

This equation is much nicer. For a dot product, you simply pair up the components of the different vectors and multiply them. You then add up the products to get the result.

How does this help you determine the angle between two vectors? Well, that has to do with the \texttt{cos(angle)} portion of equation 1. If you set these two equations as equal, you can determine the angle between the two vectors.

\begin{align*}
\texttt{v1.x} \cdot \texttt{v2.x} + \texttt{v1.y} \cdot \texttt{v2.y} + \texttt{v1.z} \cdot \texttt{v2.z} &= \texttt{abs(v1)} \cdot \texttt{abs(v2)} \cdot \texttt{cos(angle)} \\
(\texttt{v1.x} \cdot \texttt{v2.x} + \texttt{v1.y} \cdot \texttt{v2.y} + \texttt{v1.z} \cdot \texttt{v2.z}) / (\texttt{abs(v1)} \cdot \texttt{abs(v2)}) &= \texttt{cos(angle)} \\
\texttt{arccos((v1.x} \cdot \texttt{v2.x} + \texttt{v1.y} \cdot \texttt{v2.y} + \texttt{v1.z} \cdot \texttt{v2.z}) / (\texttt{abs(v1)} \cdot \texttt{abs(v2)}) &= \texttt{angle}
\end{align*}

As you know, \texttt{abs(v1)} and \texttt{abs(v2)} are simply the magnitudes of these vectors.

Cross Product

One more to go, and then you’re just about done with vectors.

The final form of multiplication for vectors is the cross product. To actually get to the equations requires determinants, and I hate determinants, so let’s just look at the actual equations, shall we?

\begin{verbatim}
//v3 = v1 x v2
v3.x = v1.y * v2.z - v1.z * v2.y
v3.y = v1.z * v2.x - v1.x * v2.z
v3.z = v1.x * v2.y - v1.y * v2.x
\end{verbatim}

Notice that none of the components that determine the resulting component are the same as either of the two components that make it up. So, determining \texttt{v3.x} does not require \texttt{v1.x} or \texttt{v2.x}. The first part of each equation has the “next” component of the first vector, multiplied by the component after that of the second vector. Therefore, for
v3.x, the first part is v1.y * v2.z, and for v3.y it is v1.z*v2.x. For the second part, you reverse the components of v1 and v2 and subtract the product. It’s a complicated thing to remember.

Remembering the cross product equations, however, isn’t nearly as important as understanding what they do for you. What they do is give you a new vector that is perpendicular to the two for which you are finding the cross product. This perpendicular value is also called the normal, which has nothing to do with normalized vectors.

For example, if you used (1,0,0) for the first vector and (0,1,0) for the second vector, the cross-product would be like so:

\[
\begin{align*}
v3.x &= 0 * 0 - 0 * 0 = 0 \\
v3.y &= 0 * 0 - 1 * 0 = 0 \\
v3.z &= 1 * 1 - 0 * 1 = 1
\end{align*}
\]

The result is (0,0,1), which is perpendicular to the plane defined by (1,0,0) and (0,1,0). This type of thing is used most often when making use of Direct3D’s lighting, and you’ll learn more about it when you read Chapter 13.

**D3DX Vector Functions**

Up until now, I haven’t said much of anything about D3DX. D3DX comes with DirectX, and consists of a number of helper functions and classes. These functions and classes are not essential to the use of Direct3D, but in many cases they do make the job easier. In order to make use of D3DX functions and classes, you have to include d3dx8math.h and d3dx8.lib in your program.

One of the areas in which D3DX is a great help is in the vector department. In this chapter, I have shown you how to perform all of the vector operations manually, because I believe it’s important to know how to do so. However, because just about everything done in Direct3D has to do with vectors, you definitely don’t want to do all of these vector operations manually. On one hand, you could make your own functions and classes to ease this problem. On the other, more practical hand, you can use the functions provided.

There are three classes in D3DX that you can use for vectors. These are D3DXVECTOR2, D3DXVECTOR3, and D3DXVECTOR4. Respectively, they have two, three, and four dimensions, and operate in pretty much the same
way. I’m not going to talk about D3DXVECTOR2 or D3DXVECTOR4; just keep in mind that the operations for these are the same as the operations for D3DXVECTOR3.

The D3DXVECTOR3 class is special in that it is derived from D3DVECTOR. This means that, because of polymorphism, the two types can be copied to and from one another, and type conversion helps out in this regard. D3DXVECTOR3 also converts into either a float* or const float*.

D3DVECTOR v1;
D3DXVECTOR3 v2;
float* pf;
const float* pcf;
v1 = v2;    //works
v2 = v1;    //works
pf = v2;    //works
pcf = v2;   //works

Another handy thing is that the operators +, -, *, and / are defined for D3DXVECTOR3, so you don’t have to do any manual additions, subtractions, multiplications (by a scalar), or divisions (by a scalar). You can simply use the operators, like so:

D3DXVECTOR3 v1, v2, v3;
float scalar;
v3 = v1 + v2;
v3 = v1 - v2;
v3 = v1 * scalar;
v3 = scalar * v1;
v3 = v1 / scalar;

Also, the +=, -=, *=, and /= operators are defined as well, and are used as normal.

D3DXVECTOR3 v1, v2;
float scalar;
v1 += v2;
v1 -= v2;
v1 *= scalar;
v1 /= scalar;

Finally, the unary + and - operators work, as well as the == and != comparison operators. This gives you a really nice, full set of operators for working with vectors.
D3DXVECTOR3 v1, v2;
v1 = + v2;
v1 = - v2;
if ( v1 == v2 ) {}  
if ( v1 != v2 ) {}

There are specific functions for calculating dot products and cross products. Use D3DXVec3Dot for dot products and D3DXVec3Cross for cross products.

FLOAT D3DXVec3Dot(
    CONST D3DXVECTOR3* pV1,
    CONST D3DXVECTOR3* pV2
);
D3DXVECTOR3* D3DXVec3Cross(
    D3DXVECTOR3* pOut,
    CONST D3DXVECTOR3* pV1,
    CONST D3DXVECTOR3* pV2
);

The purpose of these functions should be fairly obvious. There are a large number of vector-manipulation functions in D3DX, most of which you don’t use at this point. A few final functions of note include D3DXLength, which takes a D3DXVECTOR3 and spits out the magnitude of that vector, and D3DXNormalize, which normalizes a vector.

FLOAT D3DXVec3Length(
    CONST D3DXVECTOR3* pV
);
D3DXVECTOR3* D3DXVec3Normalize(
    D3DXVECTOR3* pOut,
    CONST D3DXVECTOR3* pV
);

Summary

Vectors are just so darn useful that you can only go so far in talking about Direct3D without bringing them into the picture. They are especially important as you move away from transformed vertices and into untransformed vertices. Being able to work with vectors is an absolute must as you progress.
CHAPTER 10

MATRICES
There. You made me do it. I said the “M” word. This chapter is about matrices and matrix operations, which are essential to using untransformed coordinates with Direct3D. Matrices make geometry transformation a lot easier than it was in earlier chapters, because you no longer have to apply the equations manually; you can instead abstract them into a matrix.

Your objectives for this chapter are to:

- Understand what a matrix really represents
- Know how to add matrices
- Know how to multiply matrices
- Know how to set up a matrix so it will perform translation
- Know how to set up a matrix so it will perform rotation
- Know how to set up a matrix so it will perform scaling
- Know how matrices are stored in Direct3D
- Know how to use D3DX matrix functions

What Is a Matrix?

Yeah, yeah. I saw the movie, too. In any case, the purpose of this book is not to give you a firm foundation in the mathematics behind matrices. If that’s what you want, I suggest a good college-level mathematics book that spends several chapters on the topic. It’s a big topic, and there is no possible way I can do it justice in this short book.

As far as Direct3D is concerned, a matrix is a set of 16 numbers ordered into four rows of four columns. These 16 numbers represent a set of equations. The layout of a matrix looks like Figure 10.1.

This is a 4x4 matrix, because it has four rows and four columns. In Direct3D, you use only 4x4 matrices. In mathematics, matrices can be of any size.
So, what do these numbers really mean? Well, as I stated earlier, a matrix is nothing more than a mathematical abstraction for a set of equations. A 4x4 matrix abstracts four equations, one for each column, and these equations each have four variables in them: x, y, z, and w. You learned a little about this in Chapter 9.

Here are the equations:

\[
\begin{align*}
    x_1 &= M_{11} \cdot x_0 + M_{21} \cdot y_0 + M_{31} \cdot z_0 + M_{41} \cdot w_0 \\
    y_1 &= M_{12} \cdot x_0 + M_{22} \cdot y_0 + M_{32} \cdot z_0 + M_{42} \cdot w_0 \\
    z_1 &= M_{13} \cdot x_0 + M_{23} \cdot y_0 + M_{33} \cdot z_0 + M_{43} \cdot w_0 \\
    w_1 &= M_{14} \cdot x_0 + M_{24} \cdot y_0 + M_{34} \cdot z_0 + M_{44} \cdot w_0
\end{align*}
\]

In these equations, \( x_0, y_0, z_0, \) and \( w_0 \) are the initial values sent into the matrix, and \( x_1, y_1, z_1, \) and \( w_1 \) are the values you get out of it. The initial values come from a vertex’s position, and the \( w \) value starts out at 1.0. If you’re doing 2D programming, you will pretty much leave \( w \) set to 1.0.

Now, this way of explaining matrices is nothing like most of the conventional methods of explaining them. To determine a matrix for a particular operation, you start with the equations for an operation, and then move it into a matrix.

For example, say you wanted to construct a matrix that will spit out the same numbers it was given, like so:

\[
\begin{align*}
    x_1 &= x_0 \\
    y_1 &= y_0 \\
    z_1 &= z_0 \\
    w_1 &= w_0
\end{align*}
\]
The first thing you need to do is put the coefficients for each of $x_0$, $y_0$, $z_0$, and $w_0$ into each equation. Most of these will be 0.

\[
\begin{align*}
x_1 &= 1 \times x_0 + 0 \times y_0 + 0 \times z_0 + 0 \times w_0 \\
y_1 &= 0 \times x_0 + 1 \times y_0 + 0 \times z_0 + 0 \times w_0 \\
z_1 &= 0 \times x_0 + 0 \times y_0 + 1 \times z_0 + 0 \times w_0 \\
w_1 &= 0 \times x_0 + 0 \times y_0 + 0 \times z_0 + 1 \times w_0
\end{align*}
\]

Next, you strip out all of the variable names to just leave the coefficients and put these in the matrix. Just keep in mind that each equation represents a column in the matrix, not a row, so you have to flip the number across the diagonal (it doesn’t matter much in this case because the results would be the same either way). The results appear in Figure 10.2.

This matrix has a special name. It is called the identity matrix because sending a coordinate into it will give you back the same coordinate with no changes. It is therefore the matrix equivalent of multiplying by 1.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Figure 10.2  Identity matrix

Adding Matrices

Sometimes you will need to apply two matrices to a coordinate and add the results. Rather than actually using two matrices for this, you should add the matrices themselves together. For example, consider two matrices, $M$ and $N$, the equations for which are shown here.

//matrix M
\[
\begin{align*}
x_1 &= M_{11} \times x_0 + M_{21} \times y_0 + M_{31} \times z_0 + M_{41} \times w_0 \\
y_1 &= M_{12} \times x_0 + M_{22} \times y_0 + M_{32} \times z_0 + M_{42} \times w_0 \\
z_1 &= M_{13} \times x_0 + M_{23} \times y_0 + M_{33} \times z_0 + M_{43} \times w_0 \\
w_1 &= M_{14} \times x_0 + M_{24} \times y_0 + M_{34} \times z_0 + M_{44} \times w_0
\end{align*}
\]
Because you want to add the results of these two transactions, in reality it would look something like this:

\[
\begin{align*}
x_1 &= N_{11} \cdot x_0 + N_{21} \cdot y_0 + N_{31} \cdot z_0 + N_{41} \cdot w_0 + M_{11} \cdot x_0 + M_{21} \cdot y_0 + M_{31} \cdot z_0 + M_{41} \cdot w_0 \\
y_1 &= N_{12} \cdot x_0 + N_{22} \cdot y_0 + N_{32} \cdot z_0 + N_{42} \cdot w_0 + M_{12} \cdot x_0 + M_{22} \cdot y_0 + M_{32} \cdot z_0 + M_{42} \cdot w_0 \\
z_1 &= N_{13} \cdot x_0 + N_{23} \cdot y_0 + N_{33} \cdot z_0 + N_{43} \cdot w_0 + M_{13} \cdot x_0 + M_{23} \cdot y_0 + M_{33} \cdot z_0 + M_{43} \cdot w_0 \\
w_1 &= N_{14} \cdot x_0 + N_{24} \cdot y_0 + N_{34} \cdot z_0 + N_{44} \cdot w_0 + M_{14} \cdot x_0 + M_{24} \cdot y_0 + M_{34} \cdot z_0 + M_{44} \cdot w_0
\end{align*}
\]

From here, you combine terms like so:

\[
\begin{align*}
x_1 &= ( M_{11} + N_{11} ) \cdot x_0 + ( M_{21} + N_{21} ) \cdot y_0 + ( M_{31} + N_{31} ) \cdot z_0 + ( M_{41} + N_{41} ) \cdot w_0 \\
y_1 &= ( M_{12} + N_{12} ) \cdot x_0 + ( M_{22} + N_{22} ) \cdot y_0 + ( M_{32} + N_{32} ) \cdot z_0 + ( M_{42} + N_{42} ) \cdot w_0 \\
z_1 &= ( M_{13} + N_{13} ) \cdot x_0 + ( M_{23} + N_{23} ) \cdot y_0 + ( M_{33} + N_{33} ) \cdot z_0 + ( M_{43} + N_{43} ) \cdot w_0 \\
w_1 &= ( M_{14} + N_{14} ) \cdot x_0 + ( M_{24} + N_{24} ) \cdot y_0 + ( M_{34} + N_{34} ) \cdot z_0 + ( M_{44} + N_{44} ) \cdot w_0
\end{align*}
\]

And from here, you put the coefficients into another matrix, and voilá, you can see that matrix addition works. Figure 10.3 shows the matrix.

<table>
<thead>
<tr>
<th>(M11 + N11)</th>
<th>(M12 + N12)</th>
<th>(M13 + N13)</th>
<th>(M14 + N14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M21 + N21)</td>
<td>(M22 + N22)</td>
<td>(M23 + N23)</td>
<td>(M24 + N24)</td>
</tr>
<tr>
<td>(M31 + N31)</td>
<td>(M32 + N32)</td>
<td>(M33 + N33)</td>
<td>(M34 + N34)</td>
</tr>
<tr>
<td>(M41 + N41)</td>
<td>(M42 + N42)</td>
<td>(M43 + N43)</td>
<td>(M44 + N44)</td>
</tr>
</tbody>
</table>

**Figure 10.3** Matrix addition
From the image, you can see that applying two matrices and then adding them is more easily accomplished by simply adding the matrices, and then using the matrix to transform a point.

With a bit more logic you can take this even further. If you can add two matrices, you can add the same matrix to itself, which is essentially multiplying the matrix by two. To that resulting matrix you can add the same matrix again, making it the equivalent of multiplying by three. So you can say that multiplication of a matrix by a number also works, simply because you know that addition works. Neat, huh?

### Multiplying Matrices

Now you’re in for something interesting. This section illustrates matrix multiplication, but in terms of the equations that those matrices represent. Again, the examples use M and N as the matrices, like so:

```plaintext
//matrix M
x1 = M11 * x0 + M21 * y0 + M31 * z0 + M41 * w0
y1 = M12 * x0 + M22 * y0 + M32 * z0 + M42 * w0
z1 = M13 * x0 + M23 * y0 + M33 * z0 + M43 * w0
w1 = M14 * x0 + M24 * y0 + M34 * z0 + M44 * w0

//matrix N
x2 = N11 * x1 + N21 * y1 + N31 * z1 + N41 * w1
y2 = N12 * x1 + N22 * y1 + N32 * z1 + N42 * w1
z2 = N13 * x1 + N23 * y1 + N33 * z1 + N43 * w1
w2 = N14 * x1 + N24 * y1 + N34 * z1 + N44 * w1
```

In this case, I have revised the equations so that they start with x0, y0, z0, and w0, go through matrix M to get x1, y1, z1, and w1, and then send these values through matrix N to get x2, y2, z2, and w2. This will allow you to determine the values of x2, y2, z2, and w2 in relation to x0, y0, z0, and w0, so you can then combine these matrices.

First, you substitute the M equations into the matrix N equations. Beware: these equations are going to get pretty long.

```plaintext
//matrix N
x2 = N11 * ( M11 * x0 + M21 * y0 + M31 * z0 + m41 * w0 ) + N21 * ( M12 * x0 + M22 * y0 + M32 * z0 + m42 * w0 ) + N31 * ( M13 * x0 + M23 * y0 + M33 * z0 + m43 * w0 ) + N41 * ( M14 * x0 + M24 * y0 + M34 * z0 + m44 * w0 )
```
y₂ = N₁₂ * ( M₁₁ * x₀ + M₂₁ * y₀ + M₃₁ * z₀ + m₄₁ * w₀ ) + N₂₂ * ( M₁₂ * x₀ + M₂₂ * y₀ + M₃₂ * z₀ + m₄₂ * w₀ ) + N₃₂ * ( M₁₃ * x₀ + M₂₃ * y₀ + M₃₃ * z₀ + m₄₃ * w₀ ) + N₄₂ * ( M₁₄ * x₀ + M₂₄ * y₀ + M₃₄ * z₀ + m₄₄ * w₀ )

z₂ = N₁₃ * ( M₁₁ * x₀ + M₂₁ * y₀ + M₃₁ * z₀ + m₄₁ * w₀ ) + N₂₃ * ( M₁₂ * x₀ + M₂₂ * y₀ + M₃₂ * z₀ + m₄₂ * w₀ ) + N₃₃ * ( M₁₃ * x₀ + M₂₃ * y₀ + M₃₃ * z₀ + m₄₃ * w₀ ) + N₄₃ * ( M₁₄ * x₀ + M₂₄ * y₀ + M₃₄ * z₀ + m₄₄ * w₀ )

w₂ = N₁₄ * ( M₁₁ * x₀ + M₂₁ * y₀ + M₃₁ * z₀ + m₄₁ * w₀ ) + N₂₄ * ( M₁₂ * x₀ + M₂₂ * y₀ + M₃₂ * z₀ + m₄₂ * w₀ ) + N₃₄ * ( M₁₃ * x₀ + M₂₃ * y₀ + M₃₃ * z₀ + m₄₃ * w₀ ) + N₄₄ * ( M₁₄ * x₀ + M₂₄ * y₀ + M₃₄ * z₀ + m₄₄ * w₀ )

I told you they would get long. The next task is to expand these equations, like so:

//matrix N
x₂ = N₁₁ * M₁₁ * x₀ + N₁₁ * M₂₁ * y₀ + N₁₁ * M₃₁ * z₀ + N₁₁ * M₄₁ * w₀ + N₁₁ * M₁₂ * x₀ + N₁₁ * M₂₂ * y₀ + N₁₁ * M₃₂ * z₀ + N₁₁ * M₄₂ * w₀ + N₁₁ * M₁₃ * x₀ + N₁₁ * M₂₃ * y₀ + N₁₁ * M₃₃ * z₀ + N₁₁ * M₄₃ * w₀ + N₁₁ * M₁₄ * x₀ + N₁₁ * M₂₄ * y₀ + N₁₁ * M₃₄ * z₀ + N₁₁ * M₄₄ * w₀

y₂ = N₁₂ * M₁₁ * x₀ + N₁₂ * M₂₁ * y₀ + N₁₂ * M₃₁ * z₀ + N₁₂ * M₄₁ * w₀ + N₁₂ * M₁₂ * x₀ + N₁₂ * M₂₂ * y₀ + N₁₂ * M₃₂ * z₀ + N₁₂ * M₄₂ * w₀ + N₁₂ * M₁₃ * x₀ + N₁₂ * M₂₃ * y₀ + N₁₂ * M₃₃ * z₀ + N₁₂ * M₄₃ * w₀ + N₁₂ * M₁₄ * x₀ + N₁₂ * M₂₄ * y₀ + N₁₂ * M₃₄ * z₀ + N₁₂ * M₄₄ * w₀

z₂ = N₁₃ * M₁₁ * x₀ + N₁₃ * M₂₁ * y₀ + N₁₃ * M₃₁ * z₀ + N₁₃ * m₄₁ * w₀ + N₁₃ * M₁₂ * x₀ + N₁₃ * M₂₂ * y₀ + N₁₃ * M₃₂ * z₀ + N₁₃ * m₄₂ * w₀ + N₁₃ * M₁₃ * x₀ + N₁₃ * M₂₃ * y₀ + N₁₃ * M₃₃ * z₀ + N₁₃ * m₄₃ * w₀ + N₁₃ * M₁₄ * x₀ + N₁₃ * M₂₄ * y₀ + N₁₃ * M₃₄ * z₀ + N₁₃ * m₄₄ * w₀

w₂ = N₁₄ * M₁₁ * x₀ + N₁₄ * M₂₁ * y₀ + N₁₄ * M₃₁ * z₀ + N₁₄ * m₄₁ * w₀ + N₁₄ * M₁₂ * x₀ + N₁₄ * M₂₂ * y₀ + N₁₄ * M₃₂ * z₀ + N₁₄ * m₄₂ * w₀ + N₁₄ * M₁₃ * x₀ + N₁₄ * M₂₃ * y₀ + N₁₄ * M₃₃ * z₀ + N₁₄ * m₄₃ * w₀ + N₁₄ * M₁₄ * x₀ + N₁₄ * M₂₄ * y₀ + N₁₄ * M₃₄ * z₀ + N₁₄ * m₄₄ * w₀

Ack! Now they’re even longer. The next step is to bring together common terms, like so:

//matrix N
x₂ = N₁₁ * M₁₁ * x₀ + N₁₁ * M₂₁ * y₀ + N₁₁ * M₃₁ * z₀ + N₁₁ * M₄₁ * w₀ + N₁₁ * M₁₂ * x₀ + N₁₁ * M₂₂ * y₀ + N₁₁ * M₃₂ * z₀ + N₁₁ * M₄₂ * w₀ + N₁₁ * M₁₃ * x₀ + N₁₁ * M₂₃ * y₀ + N₁₁ * M₃₃ * z₀ + N₁₁ * M₄₃ * w₀ + N₁₁ * M₁₄ * x₀ + N₁₁ * M₂₄ * y₀ + N₁₁ * M₃₄ * z₀ + N₁₁ * M₄₄ * w₀

y₂ = N₁₂ * M₁₁ * x₀ + N₁₂ * M₂₁ * y₀ + N₁₂ * M₃₁ * z₀ + N₁₂ * M₄₁ * w₀ + N₁₂ * M₁₂ * x₀ + N₁₂ * M₂₂ * y₀ + N₁₂ * M₃₂ * z₀ + N₁₂ * M₄₂ * w₀ + N₁₂ * M₁₃ * x₀ + N₁₂ * M₂₃ * y₀ + N₁₂ * M₃₃ * z₀ + N₁₂ * M₄₃ * w₀ + N₁₂ * M₁₄ * x₀ + N₁₂ * M₂₄ * y₀ + N₁₂ * M₃₄ * z₀ + N₁₂ * M₄₄ * w₀

z₂ = N₁₃ * M₁₁ * x₀ + N₁₃ * M₂₁ * y₀ + N₁₃ * M₃₁ * z₀ + N₁₃ * m₄₁ * w₀ + N₁₃ * M₁₂ * x₀ + N₁₃ * M₂₂ * y₀ + N₁₃ * M₃₂ * z₀ + N₁₃ * m₄₂ * w₀ + N₁₃ * M₁₃ * x₀ + N₁₃ * M₂₃ * y₀ + N₁₃ * M₃₃ * z₀ + N₁₃ * m₄₃ * w₀ + N₁₃ * M₁₄ * x₀ + N₁₃ * M₂₄ * y₀ + N₁₃ * M₃₄ * z₀ + N₁₃ * m₄₄ * w₀

w₂ = N₁₄ * M₁₁ * x₀ + N₁₄ * M₂₁ * y₀ + N₁₄ * M₃₁ * z₀ + N₁₄ * m₄₁ * w₀ + N₁₄ * M₁₂ * x₀ + N₁₄ * M₂₂ * y₀ + N₁₄ * M₃₂ * z₀ + N₁₄ * m₄₂ * w₀ + N₁₄ * M₁₃ * x₀ + N₁₄ * M₂₃ * y₀ + N₁₄ * M₃₃ * z₀ + N₁₄ * m₄₃ * w₀ + N₁₄ * M₁₄ * x₀ + N₁₄ * M₂₄ * y₀ + N₁₄ * M₃₄ * z₀ + N₁₄ * m₄₄ * w₀
\[y2 = N12 \times M11 \times x0 + N22 \times M12 \times x0 + N32 \times M13 \times x0 + N42 \times M14 \times x0 + N12 \times M21 \times y0 + N22 \times M22 \times y0 + N32 \times M23 \times y0 + N42 \times M24 \times y0 + N12 \times M31 \times z0 + N22 \times M32 \times z0 + N32 \times M33 \times z0 + N42 \times M34 \times z0 + N12 \times M41 \times w0 + N22 \times M42 \times w0 + N32 \times M43 \times w0 + N42 \times M44 \times w0\]

\[z2 = N13 \times M11 \times x0 + N23 \times M12 \times x0 + N33 \times M13 \times x0 + N43 \times M14 \times x0 + N13 \times M21 \times y0 + N23 \times M22 \times y0 + N33 \times M23 \times y0 + N43 \times M24 \times y0 + N13 \times M31 \times z0 + N23 \times M32 \times z0 + N33 \times M33 \times z0 + N43 \times M34 \times z0 + N13 \times M41 \times w0 + N23 \times M42 \times w0 + N33 \times M43 \times w0 + N43 \times M44 \times w0\]

\[w2 = N14 \times M11 \times x0 + N24 \times M12 \times x0 + N34 \times M13 \times x0 + N44 \times M14 \times x0 + N14 \times M21 \times y0 + N24 \times M22 \times y0 + N34 \times M23 \times y0 + N44 \times M24 \times y0 + N14 \times M31 \times z0 + N24 \times M32 \times z0 + N34 \times M33 \times z0 + N44 \times M34 \times z0 + N14 \times M41 \times w0 + N24 \times M42 \times w0 + N34 \times M43 \times w0 + N44 \times M44 \times w0\]

Better, but now you need to factor out coefficients, like so:

```c
//matrix N
x2 = ( N11 \times M11 + N21 \times M12 + N31 \times M13 + N41 \times M14 ) \times x0 + ( N11 \times M21 + N21 \times M22 + N31 \times M23 + N41 \times M24 ) \times y0 + ( N11 \times M31 + N21 \times M32 + N31 \times M33 + N41 \times M34 ) \times z0 + ( N11 \times M41 + N21 \times M42 + N31 \times M43 + N41 \times M44 ) \times w0

y2 = ( N12 \times M11 + N22 \times M12 + N32 \times M13 + N42 \times M14 ) \times x0 + ( N12 \times M21 + N22 \times M22 + N32 \times M23 + N42 \times M24 ) \times y0 + ( N12 \times M31 + N22 \times M32 + N32 \times M33 + N42 \times M34 ) \times z0 + ( N12 \times M41 + N22 \times M42 + N32 \times M43 + N42 \times M44 ) \times w0

z2 = ( N13 \times M11 + N23 \times M12 + N33 \times M13 + N43 \times M14 ) \times x0 + ( N13 \times M21 + N23 \times M22 + N33 \times M23 + N43 \times M24 ) \times y0 + ( N13 \times M31 + N23 \times M32 + N33 \times M33 + N43 \times M34 ) \times z0 + ( N13 \times M41 + N23 \times M42 + N33 \times M43 + N43 \times M44 ) \times w0

w2 = ( N14 \times M11 + N24 \times M12 + N34 \times M13 + N44 \times M14 ) \times x0 + ( N14 \times M21 + N24 \times M22 + N34 \times M23 + N44 \times M24 ) \times y0 + ( N14 \times M31 + N24 \times M32 + N34 \times M33 + N44 \times M34 ) \times z0 + ( N14 \times M41 + N24 \times M42 + N34 \times M43 + N44 \times M44 ) \times w0
```

And from this step, you can construct a new matrix with the coefficients. This new matrix is big and ugly; it is shown in Figure 10.4.

Although the matrix may be big and ugly, it does tell you exactly how you multiply two matrices, and that’s certainly something. Don’t bother memorizing it, though. Like with vectors, you’ll look at D3DX functions for multiplying matrices later.
In order to translate, you simply offset x, y, and z by a constant value. The equations for this are as follows:

\[
\begin{align*}
x_1 &= x_0 + dx \\
y_1 &= y_0 + dy \\
z_1 &= z_0 + dz \\
w_1 &= w_0
\end{align*}
\]

The constant term does not take a part in a 4x4 matrix at this point, and so you have to make \(dx\), \(dy\), and \(dz\) coefficients of \(w\), and ensure that \(w\) is 1.0 whenever you are using a translation matrix. The revised equations look like this:

\[
\begin{align*}
x_1 &= x_0 + dx \cdot w_0 \\
y_1 &= y_0 + dy \cdot w_0 \\
z_1 &= z_0 + dz \cdot w_0 \\
w_1 &= w_0
\end{align*}
\]

Luckily, this isn’t a problem, because \(w\) starts out as 1.0 and stays that way unless you change it. If you need to change \(w\), do so at the last possible moment. Now you can place the coordinate values into the equations, like so:

\[
\begin{align*}
x_1 &= 1 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 + dx \cdot w_0 \\
y_1 &= 0 \cdot x_0 + 1 \cdot y_0 + 0 \cdot z_0 + dy \cdot w_0 \\
z_1 &= 0 \cdot x_0 + 0 \cdot y_0 + 1 \cdot z_0 + dz \cdot w_0 \\
w_1 &= 0 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 + 1 \cdot w_0
\end{align*}
\]

These equations are shown in their matrix form in Figure 10.5.
Rotation Matrix

Another handy use for a matrix is to store rotational data. In Chapter 9, you learned about the equations required for rotating two dimensions. Here, you’re going to take those equations and apply them to the xy, xz, and the yz planes.

XY Plane

The first of the equation sets are used for rotating on the xy plane (also described as rotation around the z axis or roll). Here’s what they look like:

\[
x_1 = \cos(a) * x_0 - \sin(a) * y_0 \\
y_1 = \sin(a) * x_0 + \cos(a) * y_0 \\
z_1 = z_0 \\
w_1 = w_0
\]

The matrix form of these equations is shown in Figure 10.6.

XZ Plane

For rotation on the xz plane (also described as rotation around the y axis or yaw), you simply start with equations very similar to the ones
used for the xy plane. You then simply substitute all of the y and z terms.

\[ x_1 = \cos(a) \cdot x_0 - \sin(a) \cdot z_0 \]
\[ y_1 = y_0 \]
\[ z_1 = \sin(a) \cdot x_0 + \cos(a) \cdot z_0 \]
\[ w_1 = w_0 \]

I’m not going to continue taking these equations step by step to form the matrix. By now, you’ve seen enough examples of that and you should have no problem doing it on your own, if you are so inclined. The matrix version of these equations is shown in Figure 10.7.

### YZ Plane

And finally, here are the equation for rotation on the yz plane, also described as rotation around the x axis, or “pitch.”

\[ x_1 = x_0 \]
\[ y_1 = \cos(a) \cdot y_0 - \sin(a) \cdot z_0 \]
\[ z_1 = \sin(a) \cdot y_0 + \cos(a) \cdot z_0 \]
\[ w_1 = w_0 \]

![Rotation matrix:XY](image)

![Rotation matrix:XZ](image)
As you can see, these are still very similar to the other rotation equations, with y used in place of x and z used in place of y. The matrix version of these equations is shown in Figure 10.8.

**Scaling**

The final matrix I’m going to demonstrate is a scaling matrix. This matrix multiplies x, y, and z by a constant value, although each one might be multiplied by a different number. Here are the equations for this process:

\[
x_1 = sx \times x_0 \\
y_1 = sy \times y_0 \\
z_1 = sz \times z_0 \\
w_1 = w_0
\]

The matrix for this is pretty simple-looking, and is shown in Figure 10.9.

To be sure, there are lots of matrices that you can make, but translation, rotation, and scaling are the three most important operations. To combine them, you simply multiply the matrices in the order that the operations are performed.

---

**Figure 10.8**  *Rotation matrix: YZ*

**Figure 10.9**  *Scaling matrix*
D3DMATRIX

Now that you’ve read at length about matrices, it is time to actually find out how to represent them in Direct3D. The type used for matrices is called, to no one’s surprise, D3DMATRIX.

typedef struct _D3DMATRIX {
    union {
        struct {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
} D3DMATRIX;

This structure allows you to look at the values stored in the matrix in two ways. One way enables you to use the _11, _12, and so on members. The numbers indicate row first and column second. The second way is to use the m member, which is a two-dimensional array of values. Again, it is row first and column second. If you had a matrix variable called M, this is how your equations for transformation would look:

//matrix M
x1 = M._11 * x0 + M._21 * y0 + M._31 * z0 + M._41 * w0
y1 = M._12 * x0 + M._22 * y0 + M._32 * z0 + M._42 * w0
z1 = M._13 * x0 + M._23 * y0 + M._33 * z0 + M._43 * w0
w1 = M._14 * x0 + M._24 * y0 + M._34 * z0 + M._44 * w0

So, if you can create a matrix, you can definitely put them into a D3DMATRIX variable. The D3DMATRIX type is used quite a bit in the next chapter.

D3DXMATRIX

Like vectors in the last chapter, matrix operations are complicated beasts. If you did it all yourself, there would be a lot of memorizing involved. So, to help you, the DirectX SDK includes the D3DXMATRIX class. This class is derived from D3DMATRIX, and has a number of
operators and helper functions you can use to make working with matrices just that much easier.

For one thing, because \texttt{D3DXMATRIX} is derived from \texttt{D3DMATRIX}, you can use them almost interchangeably.

\begin{verbatim}
D3DMATRIX M1;
D3DXMATRIX M2;
M2 = M1;
M1 = M2;
\end{verbatim}

Also, there are a number of operators defined for \texttt{D3DXMATRIX}. You can add, subtract, and multiply by other matrices. You can also multiply or divide by a single number. Division of one matrix by another does not exist.

\begin{verbatim}
D3DXMATRIX m1, m2, m3;
float scalar;
m3 = m1 + m2;
m3 = m1 - m2;
m3 = m1 * m2;
m2 = m1 * scalar;
m2 = m1 / scalar;
\end{verbatim}

Also, the +=, -=, *=, and /= are also defined to do much the same thing, but to only one matrix.

\begin{verbatim}
D3DXMATRIX m1, m2, m3;
float scalar;
m1 += m2;
m1 -= m2;
m1 *= m2;
m1 *= scalar;
m1 /= scalar;
\end{verbatim}

There are also unary + and - signs defined for \texttt{D3DXMATRIX}. In addition, the == and != operators are defined for comparing matrices. This gives you a pretty complete set of matrix operations, and saves you a lot of work.

In order to use \texttt{D3DXMATRIX}, you need to include \texttt{d3dx8math.h} and \texttt{d3dx8.lib} in your projects.
In addition to the nifty operators, D3DXMATRIX has a number of functions to help you with common types of matrices. There are a large number of these, but I’m only going to talk about a few. Namely, these are D3DXMatrixIdentity, D3DXMatrixRotationX, D3DXMatrixRotationY, D3DXMatrixRotationZ, D3DXMatrixScaling, and D3DXMatrixTranslation. These are all of the types of matrix you have seen through the course of this chapter.

The D3DXMatrixIdentity function is useful when you are multiplying matrices. The identity matrix gives you a starting point.

```c
D3DXMATRIX* D3DXMatrixIdentity(
    D3DXMATRIX* pOut
);
```

This function has only one parameter, a pointer to a D3DXMATRIX (you cannot send a D3DXMATRIX pointer here). The returned value is a pointer to the same matrix you sent in, which is useful for performing a number of matrix operations on a single line.

The three rotation functions describe the rotations as X, Y, and Z. A quick way to remember which does what is that whatever letter is in the function name is the one that isn’t affected by the matrix. Therefore, D3DXMatrixRotationX affects Y and Z, but not X, and so on.

```c
D3DXMATRIX* D3DXMatrixRotationX(
    D3DXMATRIX* pOut,
    FLOAT Angle
);
D3DXMATRIX* D3DXMatrixRotationY(
    D3DXMATRIX* pOut,
    FLOAT Angle
);
D3DXMATRIX* D3DXMatrixRotationZ(
    D3DXMATRIX* pOut,
    FLOAT Angle
);
```

Each function takes two parameters, an in/out pointer to a D3DXMATRIX and an angle by which you want to rotate. Again, these functions return the same pointer you send in.
The scaling and translation matrices are pretty easy.

```c
D3DXMATRIX* D3DXMatrixScaling(
    D3DXMATRIX* pOut,
    FLOAT sx,
    FLOAT sy,
    FLOAT sz,
);
```

To `D3DXMatrixScaling`, you send the matrix you want to set up as a scaling matrix along with the three scaling values. The value returned is the pointer to the same matrix you sent to it.

```c
D3DXMATRIX* D3DXMatrixTranslation(
    D3DXMATRIX* pOut,
    FLOAT x,
    FLOAT y,
    FLOAT z,
);
```

And finally, `D3DXMatrixTranslation` takes a pointer to the matrix you are setting and the three translation values.

Although this is not a comprehensive list of the available functions, it will enable you to do just about every operation you need to do.

**Summary**

This is the third chapter in a row without example programs. I know you must be anxious to put all of the theory you have been exploring to use. Unfortunately, the thing about Direct3D (or any 3D API) is that it requires a whole lot of knowledge before you can do much of anything. However, don’t worry. In the next chapter, you get away from theory and back to code.
CHAPTER 11

Geometry Pipeline
In the last three chapters I’ve taken you through a lot of theory about geometry transformation. It’s time to put that theory to work and get back to writing code. This is not a long chapter, but it is a very important one.

At the end of this chapter you should:

- Understand, concretely, what happens in the geometry pipeline
- Know how to set the various transformation matrices
- Know how to multiply the various transformation matrices

The Pipeline

In Chapter 8 you took an abstract look at what the geometry transformation pipeline does to points. You will now be looking at it concretely, and reading about what Direct3D actually does when called upon to transform points. You will from now on be working with untransformed vertices, so your vertex format has to change to reflect this.

The vertices begin in model space, representing all vertices in relation to the center of mass of the model. The word *model* is an abstract term here; it doesn’t mean an actual 3D model.

When some sort of primitive rendering function is called, such as DrawPrimitive, DrawIndexedPrimitive, DrawPrimitiveUP or DrawIndexedPrimitiveUP, the geometry pipeline is set into motion.

The first thing Direct3D does is to multiply by the world matrix to get the world coordinates. Then, it multiplies by the view matrix to get the camera coordinates. Then, it multiplies by the projection matrix to get cube space coordinates. Finally, the output is clipped and scaled to the viewport before being sent to the rasterizer to be drawn on the screen.

There is one extra thing that is done right after the point goes through the projection matrix. If \( w \) is not equal to 1, then \( x, y, \) and \( z \) are all divided by \( w \) as part of the projection.
I like to work backwards through the pipeline when determining how to set up my matrices for world, view, and projection. Because I am not doing traditional 3D, I don’t use the matrices commonly used for 3D. Instead, I rely on the fact that I know matrices are based on a set of equations. I take the goal of what the output should be, and I determine what I want the input to be, and then I come up with equations for how to get from the input to the output. This is a lot easier if you work backwards (just like when you are solving a maze in a puzzle book, it is often easier to go from the end to the beginning rather than the other way around).

So, let’s do that just for an example. You’re going to start at the end of the pipeline, make a few decisions along the way, and then come up with your matrices.

First, the main goal of this exercise is to set up the pipeline so that it renders an 8 8 grid of squares, which can be used as a checkerboard or chessboard. This is 2D, so you can ignore Z. In model space, each board square is 1.0 units tall and 1.0 units wide, so it has the coordinates (0.0,0.0), (1.0,0.0), (0.0,1.0), and (1.0,1.0), with the z component always equaling 0.0. I do mean for these to be actual squares, not just rectangles, so the matrices should make each one a square. The size of the square doesn’t matter much, but for the purposes here you want it to take up as much of the screen as possible. Because the height of a screen is shorter than the width, it means that the board will take up the entire vertical span of the screen.

So if your viewport is 640 480 and you want the central 480 480 area of the viewport reserved for your board, its corners would be at (80,0), (560,0), (80,480), (560,480). These are in pixel coordinates. To convert them to the cube space coordinates, subtract half the width and half the height from x and y, and then divide by half the width and half the height of the coordinate, and finally multiply y by 1. Half of the width, in this case, is 320, and half the height is 240.

Let’s do the upper-left corner.

\[
\left(\frac{80-320}{320}, -\frac{0-240}{240}\right)
\]
\[
\left(-\frac{240}{320}, \frac{240}{240}\right)
\]
\[
\left(-0.75, 1\right)
\]
And now the lower-right corner, which you expect will be the same as
the upper-left corner, only negative.

\[(560-320)/320, -(480-240)/240\]
\[(240/320, -240/240)\]
\[(0.75, -1)\]

And lo and behold, the opposite corners are virtually the same, the
only difference being negative signs.

Next, take a quick look at your model space to world space transfor-
mation. You’ll get back to the cube space coordinates in a little bit.

For model space, recall that each square ranges from \((0.0, 0.0)\) to
\((1.0, 1.0)\). When arranged in an 8 8 grid (thus making the “world” in
as far as this example is concerned), this will take up the area between
\((0.0,0.0)\) \((8.0,8.0)\). These are the coordinates that eventually have to
end up as \((0.75, 1)\) and \((0.75, -1)\).

You can do this all in one matrix, if you so desire. You just have to
come up with the equations for it.

For the x coordinate, 0.0 maps to 0.75, and 8.0 maps to 0.75. The
widths (right to left) are 8.0 for the world coordinates and 1.5 for the
cube space coordinates. You simply multiply by 1.5/8.0, then add
0.75, and you have your x equation, like so:

\[x_1 = 0.1875 \times x_0 - 0.75\]

0.1875 is what 1.5/8.0 comes out to. For the y coordinate, the height
(bottom-top) is 8.0 for the world coordinates, and 2 for the cube
space coordinates. 0.0 maps to 1.0. For the y coordinate, you do much
the same thing; add 1 and multiply by 2.0/8.0, like so:

\[y_1 = -0.25 \times y_0 + 1\]

Because w is 1, you use -0.75 and 1 as the w coefficients, and add the
other terms for each equation, as well as the equations for z and w,
which do not change.

\[x_1 = 0.1875 \times x_0 + 0 \times y_0 + 0 \times z_0 -0.75 \times w\]
\[y_1 = 0 \times x_0 - 0.25 \times y_0 + 0 \times z_0 + 1 \times w\]
\[z_1 = 0 \times x_0 + 0 \times y_0 + 1 \times z_0 + 0 \times w_0\]
\[w_0 = 0 \times x_0 + 0 \times y_0 + 0 \times z_0 + 1 \times w_0\]

Figure 11.1 shows these equations in their matrix form.
Setting up a Transformation

Okay, so now it’s time to do something for real. You’ve read long enough about setting up the foundations and the theory. To set up a transformation matrix in Direct3D, you first fill in a D3DMATRIX or a D3DXMATRIX structure, and then you call IDirect3DDevice8::SetTransform.

HRESULT IDirect3DDevice8::SetTransform(
    D3DTRANSFORMSTATETYPE State,
    CONST D3DMATRIX* pMatrix
);

On success, this function will return D3D_OK. On failure, it will return D3DERR_INVALIDCALL. The pMatrix is a pointer to a D3DMATRIX structure, but a D3DXMATRIX structure pointer will work here as well. The State parameter tells Direct3D which transformation stage you are setting. There are a number of values that you can use in this parameter, but this example uses only three. They are D3DTS_WORLD, D3DTS_VIEW, and D3DTS_PROJECTION. As you might expect, these are the transitions from model to world, world to view, and view to projection.

You haven’t seen an example program for a while, and it’s way past time for a new one. Example 11.1 (X:\Source\Example11_1) on the CD shows you how to make the checkerboard I described earlier with only eight vertices (four for each color square) and matrices. This demonstrates one of the really cool uses of matrices: if you have a multitude of models (and although each square only consists of two triangles, it does qualify as a “model” in the broadest sense of the word), you can
repeat it numerous times in various places by simply using a different set of matrices. Figure 11.2 shows the output of Example 11.1.

For Example 11.1, notice that you are now using untransformed coordinates, which means you have dropped the \texttt{rhw} member of \texttt{CustomVertex}. Also, in this example I'm not using texture coordinates, but you could easily accommodate them if you wanted to.

If you wanted to get the contents of a transformation stage, you would use \texttt{IDirect3DDevice8::GetTransform}.

\begin{verbatim}
HRESULT IDirect3DDevice8::GetTransform(
    D3DTRANSFORMSTATETYPE State,
    D3DMATRIX* pMatrix
);
\end{verbatim}

This function returns \texttt{D3D_OK} on success and \texttt{D3DERR_INVALIDCALL} on failure. The first parameter specifies which matrix you want to retrieve—(\texttt{D3DTS_WORLD}, \texttt{D3DTS_VIEW}, or \texttt{D3DTS_PROJECTION})—and the second parameter is a pointer to a \texttt{D3DMATRIX} (a \texttt{D3DXMATRIX} will work as well).
into which the contents of the specified matrix are copied. You probably won’t need to use this function most of the time, as you will have stored the contents of all the matrices into variables anyway.

**Multiplying Transformation Matrices**

You can multiply any of the transformation stages by another matrix without having to keep those matrices as variables, doing the multiplying yourself, then using `IDirect3DDevice8::SetTransform`.

For an example of this, I’m going to create the checkerboard again, this time using three matrices. The world matrix remains the same, but I’m adding a view matrix to center the board (a simple matter of translating x and y by -4 to give you a range from -4 to +4 for x and y). For the projection matrix, this means you only have to scale x by 0.1875 and y by -0.25 and it’ll work just fine.

As an additional piece of processing, each time you render the scene in this example you will rotate the projection matrix by one degree around the z axis. However, you don’t have to perform this rotation manually; you can use the `IDirect3DDevice8::MultiplyTransform` function.

```cpp
HRESULT IDirect3DDevice8::MultiplyTransform(
    D3DTRANSFORMSTATETYPE State,
    CONST D3DMATRIX* pMatrix
);
```

Like the other functions dealing with matrices, this function returns `D3D_OK` if successful and `D3DERR_INVALIDCALL` if it fails. The first parameter holds the transformation stage you want to multiply, and `pMatrix` is a pointer to a `D3DMATRIX` or `D3DXMATRIX` by which you want the stage multiplied.

Example 11.2 does exactly this. It sets up a projection, view, and world matrix (the world matrix still changes for each square drawn), and a rotation matrix by which the projection matrix is multiplied each frame. Without having to do the bizarre trigonometry you’ve had to do in the past, the checkerboard rotates around the center. Cool, eh? Figure 11.3 shows the output of this example.
Summary

Now you have officially moved from transformed vertices to untransformed vertices in a concrete way. Sure, you had to slog through three chapters of boring theory, but now, at the end of the tunnel, you know how to do any sort of arbitrary transformation using matrices. As always, if you can find a set of equations for it, you can make it happen.

NOTE

I just wanted to note here at the end that you can certainly mix your transformed and untransformed vertices within the same scene. You just have to switch the vertex shader and use a different vertex type, which is no big deal and gives you a lot of power in how you use Direct3D.
CHAPTER 12

Z Buffering
In the first 11 chapters of this book I mentioned the z axis many times, but I haven’t really said too much about it, because it just wasn’t necessary at the time. This chapter is where that all changes, and you learn how to use z buffering, a powerful tool for much more than simply making 3D scenes. Even in 2D there is usually some sort of layering going on within the 2D scene. For example, in a chess game, the pieces are “above” or “closer than” the board.

Your objectives for this chapter are to:

- Understand what z buffering is
- Know how to set up z buffering
- Know how to make use of z buffering

**What Is Z Buffering?**

Your monitor screen is two-dimensional, and is likely to remain so for a very long time. There is no third dimension for computer graphics. 3D APIs use trickery and subterfuge to fool your eyes into thinking that there’s a third dimension. Z buffering is one of those tricks.

Until now, you haven’t been much concerned with the z part of the vertex. You have learned that the viewport’s z range is from 0.0 to 1.0, and that anything outside of that is not drawn. However, with matrices, you know how easy it is to use a different range of z and then make it fit into the range you desire.

Typically, 0.0 means near and 1.0 means far. You don’t have to do it this way, of course, that’s just the default. You can use 1.0 to represent near and 0.0 to represent far if you really want to; you’ll see more of that later. This section first looks into how z buffering works.

Recall the pixel formats you learned about quite a while ago. You looked at formats like D3DFMT_D16 and D3DFMT_D32. These depth formats (the D stands for depth) are used with z buffers.
What exactly is a z buffer? Well, like the screen and the back buffers, a z buffer is just a collection of pixels, but instead of storing color information, it stores depth information. When all bits of a depth pixel are 0, the z is 0.0. When all bits of a depth pixel are 1, the z is 1.0. Easy, right?

Each pixel on the display has a corresponding pixel in the depth buffer (I use the term depth buffer interchangeably with z buffer). When you clear the viewport and are using a depth buffer, you also clear the depth buffer to a certain value, typically 1.0. Next, when rendering a triangle, something extra happens for each pixel. The z value of the pixel being drawn is tested against the z buffer. If it passes the test (the default test is to determine whether the pixel has a lower z value than the value stored in the z buffer at that point), the pixel is drawn. If the test fails, the pixel is not drawn.

In programmer terms, then, z buffering is nothing more than a bunch of if statements. If NewZ < CurrentZ, then draw the pixel. If not, draw nothing.

So, if z buffering is nothing more than an application of comparisons and if statements, then you can easily find an unconventional use for it. And so you shall.

**Setting up for Z Buffering**

Setting up for z buffering can be really easy. Or, if you’re one of those people who likes difficulty, it can be difficult. The method you use is totally up to you. This chapter covers both methods.

**The Easy Way to Set up Z Buffering**

The easy way to set up z buffering involves the code near the creation of the IDirect3DDevice8 object. If you call the D3DPRESENT_PARAMETERS structure, there are two members that relate to depth buffers, a BOOL named EnableAutoDepthStencil and a D3DFORMAT named AutoDepthStencilFormat. The purpose of these members should be pretty obvious, but I’ll explain them anyway.

The EnableAutoDepthStencil member must be TRUE if you want the device to automatically be created with a depth buffer, or FALSE if you
don’t. So far, you have been specifying FALSE, because you have not been using depth buffers.

The AutoDepthStencilFormat should be D3DFMT_D16_LOCKABLE, D3DFMT_D32, D3DFMT_D15S1, D3DFMT_D24S8, D3DFMT_D16, D3DFMT_D24X8, or D3DFMT_D24X4S4. Because you aren’t setting up a stencil buffer here, you shouldn’t use any of the formats with an S in them (it would be pointless). This leaves D3DFMT_D16_LOCKABLE, D3DFMT_D32, D3DFMT_D16, and D3DFMT_D24X8. Because being able to lock a depth buffer is costly in performance, remove D3DFMT_D16_LOCKABLE from the list as well. This leaves three of the most commonly available depth formats, D3DFMT_D16, D3DFMT_D24X8, and D3DFMT_D32.

Now, because of differences in graphics cards, you won’t always be able to use all three of these. Luckily, D3DFMT_D16 is nearly universal, so you can normally get away with using it. However, you should still determine whether the depth format you want supports the device you are making. You can do this by using IDirect3DDevice8::CheckDeviceFormat, like so:

```c
hr = g_pd3ddev->CheckDeviceFormat ( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, D3DFMT_R5G6B5, D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, D3DFMT_D16 );
if ( FAILED ( hr ) )
{
    //failed
}
```

If this check passes, you still have another check to perform, using IDirect3DDevice8::CheckDepthStencilMatch.

```c
HRESULT IDirect3DDevice8::CheckDepthStencilMatch( 
    UINT Adapter, 
    D3DDEVTYPE DeviceType, 
    D3DFORMAT AdapterFormat, 
    D3DFORMAT RenderTargetFormat, 
    D3DFORMAT DepthStencilFormat
);
```

On success, this function returns D3D_OK. On failure, it returns D3DERR_INVALIDCALL or D3DERR_NOTAVAILABLE. D3DERR_NOTAVAILABLE indicates that the depth stencil format is not a match.
The first three parameters are pretty self-explanatory. They are the adapter number (Adapter), the type of device (DeviceType), and the format of the adapter you are using (AdapterFormat). The last two parameters (RenderTargetFormat and DepthStencilFormat) are the format of the render target (this is usually the same format as the adapter) and the depth format to check for compatibility, respectively.

If, for example, you were checking to see whether D3DFMT_D16 was available for a full-screen application, and you were using D3DFMT_R5G6B5 on the default adapter and the HAL device, this is how the function call would look:

```cpp
hr = g_pd3ddev->CheckDepthStencilMatch ( D3DADAPTER_DEFAULT , D3DDEVTYPE_HAL , D3DFMT_R5G6B5 , D3DFMT_R5G6B5 , D3DFMT_D16 );
if ( FAILED ( hr ) )
{
    //failure
}
```

And this is the easy way? All of these checks are something of a pain in the neck. It would be much easier just to make a function that performs both checks and returns a simple true or false, wouldn’t it?

Well, I think so, anyway.

```cpp
bool CheckZBuffer ( IDirect3DDevice8* pDevice , UINT Adapter , D3DDEVTYPE DevType , D3DFORMAT AdapterFormat , D3DFORMAT DepthFormat )
{
    HRESULT hr ;
    //check device format
    hr = pDevice->CheckDeviceFormat ( Adapter , DevType , AdapterFormat ,
D3DUSAGE_DEPTHSTENCIL , D3DRTYPE_SURFACE , DepthFormat ) ;
    //if check fails, return false
    if ( FAILED ( hr ) ) return ( false ) ;
    //check depth stencil match
    hr = pDevice->CheckDepthStencilMatch ( Adapter , DevType ,
AdapterFormat , AdapterFormat , DepthFormat ) ;
    //if check fails, return false
    if ( FAILED ( hr ) ) return ( false ) ;
    //otherwise, return true
    return ( true ) ;
}
```
After you have found a depth format that passes both tests, you simply put that value into the AutoDepthStencilFormat member of D3DPRESENT_PARAMETERS, and go ahead and create the device. You need do nothing more.

This also works in a call to IDirect3DDevice8::Reset.

The Hard Way to Set up Z Buffering

For the hard way, you create the depth buffer yourself, and then tell the device to use it. You might want to do this when you have already created the device and don’t want to call IDirect3DDevice8::Reset, or when you have changed your render target to a surface other than the back buffer, with perhaps a different size or format.

Even if you decide to go the hard way, you still need to perform the same checks in order to find a suitable depth format. The next step is to create the z buffer itself. This is done with IDirect3DDevice8::CreateDepthStencilSurface.

```
HRESULT IDirect3DDevice8::CreateDepthStencilSurface(
    UINT Width,
    UINT Height,
    D3DFORMAT Format,
    D3DMULTISAMPLE_TYPE MultiSample,
    IDirect3DSurface8** ppSurface
);
```

This function returns D3D_OK if successful or D3DERR_INVALIDCALL or D3DERR_OUTOFVIDEOMEMORY if not. The first parameter (Width) is the desired width of the surface in pixels. The second parameter (Height) is the desired height in pixels. The size of the z buffer must be the same size as the render target, so if you are making a depth buffer to be used with a 640 480 render target, the z buffer must also be 640 480. The third parameter (Format) is the format of the depth buffer, such as D3DFMT_D16. The fourth parameter (MultiSample) is the multisample type, which you aren’t using, so put D3DMULTISAMPLE_NONE. The last parameter (ppSurface) is a pointer to a pointer to an IDirect3DSurface8 object, which is filled in with a pointer to the newly created depth buffer.
Once the surface has been created, you can pair it with the current render target or a completely different render target by calling IDirect3DDevice8::SetRenderTarget.

HRESULT IDirect3DDevice8::SetRenderTarget(
    IDirect3DSurface8* pRenderTarget,
    IDirect3DSurface8* pNewZStencil
);

You’ve seen this function before, so I’ll just briefly go over it. The pRenderTarget parameter is a pointer to a new surface object to use as the render target. If NULL is passed into this parameter, the current render target is not changed. The pNewZStencil parameter is a pointer to the surface object to use as the new depth buffer. The current depth buffer is removed even when this parameter is NULL. So, if you created a depth buffer surface called pZSurface and you wanted to change the current depth buffer to use this new surface, this is how it would look:

    g_pd3ddev->SetRenderTarget ( NULL , pZSurface ) ;

But what happens to the old depth buffer (if there was one)? Well, if it was automatically created when the device was, and you haven’t increased its reference count above 1, it disappears. Poof. Gone. Forever. If you want to use it later, you’re out of luck.

So, what if you do want to use it later? In that case, you’ll want to grab it from the device prior to changing to another depth buffer, using the IDirect3DDevice8::GetDepthStencilSurface function.

HRESULT IDirect3DDevice8::GetDepthStencilSurface(
    IDirect3DSurface8** ppZStencilSurface
);

This function returns D3D_OK on success and D3D_INVALIDCALL on failure. The ppZStencilSurface is a pointer to a pointer to an IDirect3DSurface8 object that gets filled in with a pointer to the currently-in-use depth buffer.

And that’s how you set up z buffering the hard way.
Reference Counting
Now, a few words about reference counting. At the end of your program, any COM objects with a reference count of 1 or more aren’t deleted (at least in theory). They hog up memory and resources and degrade the performance of your computer. It becomes your responsibility as a programmer to make sure that by the end of the program everything has a reference count of 0.

If you go the easy way and have the depth buffer automatically created when the device is created, this is how it remains through the rest of the program. As long as you have released the device you’re okay, because the depth buffer had a reference count of 1, and releasing the device also released the depth buffer. Therefore, both reference counts go to 0 and the object is deleted.

However, if you call IDirect3DDevice8::GetDepthStencilSurface, you have increased the reference count of the depth buffer to 2. At some point, you need to release it. If a new depth buffer is put into place, the count drops to 1. If you then reassign the depth buffer, it goes back up to 2. See what I’m saying here? Reference counts can trip up even the most experienced programmers.

Similarly, if you go the hard way, creating the depth buffer sets the reference count to 1. Setting it as the device’s depth buffer brings that count up to 2. If it remains with the device until the device is destroyed, it will go back down to 1. You still need one call to Release in order to make the count 0 and delete the object.

So be careful and try to think ahead. Reference counts can easily get out of hand, especially if you aren’t paying attention (like all those times that you are up to four in the morning coding with blurry vision).

Making Use of Z Buffers
After you’ve set up the z buffer, using it is pretty easy, and for the most part, you don’t need to worry about it any more. There are a few more details regarding z buffers that you need to know about to use them effectively, though.
The first thing is that you need to clear the z buffer every frame. This is done with the IDirect3DDevice8::Clear statement, like so:

```
g_pd3ddev->Clear ( 0 , NULL , D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER , 0 , 1.0 , 0 ) ;
```

This line clears the back buffer to color 0 (black) and clears the z buffer so that each pixel has a z value of 1.0. By default, Direct3D writes any pixel with a z value less than or equal to the current z of the depth buffer.

Of course, there are times when you don’t want that behavior. You might instead want 0.0 to mean “far” and 1.0 to mean “near.” Direct3D will accommodate you. Setting that up is a simple matter of setting a render state.

```
g_pd3ddev->SetRenderState ( D3DRS_ZFUNC , D3DCMP_GREATEREQUAL ) ;
```

Any of the D3DCMP_* constants (which you learned about in Chapter 7, when you learned about alpha testing) can be used. The previous line of code sets up the z buffer so that 0.0 means far and 1.0 means near.

Another handy render state dealing with z buffers is D3DRS_ZENABLE. With it, you can turn z buffering on and off. To turn off z buffering, set this render state to D3DZB_FALSE. To turn it on, set this state to D3DZB_TRUE.

Example 12.1 (X:\Source\Example12_1) on the CD demonstrates z buffering. In this example, there are two squares, one red and one green, which rotate around the center of the screen. They are rotating in opposite directions so that they collide twice in each revolution. The red square is being drawn first, and the green square is being drawn second. However, if you look at Figure 12.1, you can see (sort of—it’s black and white, of course) that the red square appears in front of the green square, all because the red square has a lower z value (0.25) than the green square (0.75). Hooray for z buffering!

Using z buffers is not limited to untransformed coordinates, however. You can also use them with transformed coordinates. This way, you can assign various z coordinates to different layers, and you can have 2D sprites behind and ahead of one another, without having to worry about the order you draw them in. If you switched around Example
12.1 so that the z function was D3DCMP_GREATEREQUAL, and instead cleared out the z buffer to 0.0, you’d get the green square in front of the red square.

And of course, the 0.0 to 1.0 range of Z doesn’t have to look that way to you. You can easily make a matrix that has Z range from -10.0 to 10.0 in model space, and the matrices fit it into the 0.0 to 1.0 range by the time it gets through the pipeline.

Summary

As you can see, z buffering isn’t hard. It’s not even very challenging. Direct3D has made it so easy that it’s hard to imagine a time when it didn’t exist. However, there is no need for to use it as it was intended (that is, to show 3D scenes). Indeed, there are many opportunities to use z buffering in a 2D scene as well.
CHAPTER 13

Lighting
Up until now, you’ve been coloring your triangles by yourself, which is fine for many applications. However, Direct3D enables you to simulate light sources; doing so can add realism and lets you add some interesting effects that are rather difficult to implement manually. The Direct3D lighting system has only a vague similarity to real light, which is why it’s described as a simulation. Your goal for this chapter are to

- Understand the Direct3D lighting system in the abstract
- Understand the D3DMATERIAL8 structure
- Understand the D3DLIGHT8 structure
- Know how to make use of point lights
- Know how to make use of directional lights
- Know how to make use of spotlights
- Know how to make use of ambient light
- Know how to make use of specular light
- Know how to make use of emissive light

Direct3D Lighting

Setting up and using other parts of Direct3D are rather straightforward and simple. The same is true of lights. However, there are a number of new structures and functions that goes along with lighting, and you have to make some modifications to other structures that you’ve been using all along.

The first—and possibly the most significant—change when adding Direct3D lighting is that which occurs to the CustomVertex structure. In order to use some forms of lighting properly you need to add the normal vector, like so:

```c
#define CUSTOM_VERTEX_FVF D3DFVF_XYZ | D3DFVF_NORMAL
struct CustomVertex
```
Also notice that the diffuse component (which you have been using until now) is gone. It is replaced by something else when you use lighting. To get the normal for a vertex of a triangle, you can either set it yourself or determine the vectors between each of the vertices in the triangle, and then take the cross product of those vectors. You would then normalize the result. I’ll have some code that demonstrates this a little later.

There are four types of “light” that you can use in Direct3D: ambient, diffuse, specular, and emissive. They have only a remote relationship to actual light, as I said before. Ambient light is the light that encompasses the entire scene. Diffuse light is much like the diffuse component of the custom vertex structure you used in earlier chapters. Specular light deals with “shininess.” Emissive light deals with the objects that seem to “glow” with their own light.

**D3DMATERIAL8**

Much of how lighting works in Direct3D relies on two structures. One of those structures is D3DMATERIAL8. This structure defines how something reflects light of different types.

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

This structure is comprised of four D3DCOLORVALUE structures (which you have not yet seen) and a single float value. As you might expect, each of the D3DCOLORVALUE members correspond to a different type of light, indicated by the name of the member. So, if you are using diffuse light, you use the Diffuse member. The odd-ball float in the
structure, Power, is used with specular light, which you’ll learn about later.

The **D3DCOLORVALUE** structure is also pretty simple.

typedef struct _D3DCOLORVALUE {
    float r;
    float g;
    float b;
    float a;
} D3DCOLORVALUE;

This structure contains four floats that correspond to red, green, blue, and alpha. Because they are floating point numbers and not integers, you don’t use values 0 through 255 for colors. Instead, you use 0.0 through 1.0 to indicate color, with 0.0 meaning the same thing as 0, and 1.0 representing the same thing as 255.

A *material* specifies how something responds to light being shined on it. You don’t use all of the components of a material with every one you use. The material abstracts the substance of which the polygons are made. For example, a piece of plastic has a different material than a piece of wood, and a piece of metal has again a different material.

For the time being, let’s consider only diffuse light, because it is the most readily understandable and makes discussing light simple. The rest of the components work in the same way, and you’ll learn about them a bit later.

The value of the **Diffuse** member of a material is much like asking “if you have a diffuse white light reflecting off of a polygon with this material, what color will you see?” When setting up a material, you first clear it to all 0, with `memset` or `ZeroMemory`, and then set up whatever components you like. For example, if you were setting up a material that reflected red when a white light was shining on it, this is what the code would look like:

```c
D3DMATERIAL8 material ;
memset ( &material , 0 , sizeof ( D3D8MATERIAL ) ) ;
material.Diffuse.r = 1.0f ;
material.Diffuse.g = 0.0f ;
material.Diffuse.b = 0.0f ;
```
Because you aren't working with alpha blending, you can ignore the alpha portion of the Diffuse member. As you can see, the red component is set to 1.0 with the green and blue set to 0.0, so whenever white light shines on this material, red will be reflected back.

After a D3DMATERIAL8 structure is set up, you have to let Direct3D know that you intend to use it. The function for doing that is IDirect3DDevice8::SetMaterial.

```c
HRESULT IDirect3DDevice8::SetMaterial(
    CONST D3DMATERIAL8* pMaterial
);
```

If successful, this function returns D3D_OK. If it fails, you get a D3DERR_INVALIDCALL. The only parameter is a pointer to a D3DMATERIAL8 structure. Once you set a material, that material remains in use until you call this function again.

If you want to retrieve the current material, use IDirect3DDevice8::GetMaterial.

```c
HRESULT IDirect3DDevice8::GetMaterial(
    D3DMATERIAL8* pMaterial
);
```

This function returns either D3D_OK or D3DERR_INVALIDCALL, just like SetMaterial does. The pMaterial parameter is filled in with the current material.

**D3DLIGHT8**

The other really important lighting structure is D3DLIGHT8. It contains all of the properties of a light source.

```c
typedef struct _D3DLIGHT8 {
    D3DLIGHTTYPE Type;
    D3DCOLORVALUE Diffuse;
    D3DCOLORVALUE Specular;
    D3DCOLORVALUE Ambient;
    D3DVECTOR Position;
    D3DVECTOR Direction;
} D3DLIGHT8;
```
There is a lot of stuff here. This section takes it slowly, because a “light source” can mean so many things.

**Type**

The first, and perhaps the most important member is the **Type**. There are three types of light sources that Direct3D deals with. These are listed and briefly explained in Table 13.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DLIGHT_POINT</td>
<td>The light source is a single point and light is sent out equally in all directions.</td>
</tr>
<tr>
<td>D3DLIGHT_SPOT</td>
<td>The light source is a spotlight. Light is given out in a cone shape.</td>
</tr>
<tr>
<td>D3DLIGHT_DIRECTIONAL</td>
<td>The light source is directional. It seems to come from everywhere, but only in a single direction.</td>
</tr>
</tbody>
</table>
light source emits. If, for example, you were creating a light that emits pure white diffuse light, this is part of the code that would do it.

```c
D3DLIGHT8 light;
memset (&light, 0, sizeof (D3DLIGHT8));
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
//set up other members here
```

Setting up the Ambient and Specular components of the light works much the same way.

**Position and Direction**

Depending on the Type member, Position and Direction might not be significant. Table 13.2 lists the types of light, and indicates whether or not these two members are used.

As you can see, only spotlights have both position and direction. A point light has only a position, and a directional light has only a direction.

Although the meanings of point, spot, and directional are fairly clear, it doesn’t hurt to have a real-world example of each. Light bulbs and candle flames are good examples of point lights. A flashlight is a good example of a spotlight (other than the obvious example, a spotlight). The sun is the best example of a directional light. Directional lights are typically just light sources that are very powerful and so far away

<table>
<thead>
<tr>
<th>Type</th>
<th>Position</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DLIGHT_POINT</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>D3DLIGHT_SPOT</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>D3DLIGHT_DIRECTIONAL</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
that it doesn’t much matter where you are, the light is always coming in at the same angle.

It is important to note here that the position of a light is expressed in world coordinates. Lights are positioned within the entire world, rather than in model space.

**Range**

The Range parameter is kind of a weird one, because it has no corresponding real-world analogy. This member specifies the maximum distance from a light that still has an effect. If something is farther away from the light source than the number specified in Range, the light does not affect that object.

In the real world, you know that light keeps traveling until it hits something, bounces off, and hits something else. However, at large distances, the amount of light is barely detectable, because all of the photons have been scattered. Treat the Range member as though it were the distance at which the light no longer has a significant effect, and you’ll sleep better (and don’t tell any of your Physics-savvy friends).

The maximum value for Range is the square root of a constant called `FLT_MAX`, which, as you might expect, is the maximum value for a float. Directional lights ignore the Range member.

**Attenuation0, Attenuation1, and Attenuation2**

These three constants control the light’s attenuation, or change in intensity over distance. If you turn on a light in your house, you know that the room will be brightest right near the light, and darkest at the point farthest from the light, because light intensity decreases with the square of the distance. This is attenuation. It is only important for lights that have a position, so for directional lights all of these members can just be zero.

These three values are actually coefficients of an equation that Direct3D uses to calculate how a light has decreased in intensity over distance.

\[
\text{Intensity}_1 = \frac{\text{Intensity}_0}{(\text{Attenuation}_0 + \text{Attenuation}_1 \times d + \text{Attenuation}_2 \times d \times d)}
\]
In this equation, \( \text{Intensity}_0 \) is the original intensity of the light (one of the color components of one of the types of light emitted, a number between 0.0 and 1.0). The \( d \) represents distance. So, if you want to have a light that doesn’t change intensity, you can use 1.0 for \( \text{Attenuation} \). If you want to have intensity drop off linearly, you can use 1.0 for \( \text{Attenuation}_1 \), and if you want to have intensity drop off parabolically, you can use 1.0 for \( \text{Attenuation}_2 \). Using numbers above or below 1.0 lets the light fade faster or slower with distance. Using more than one of these members can make for some interesting and odd lighting effects. The values you use are very much the product of experimentation. I definitely suggest trying different attenuation values to see the different effects first-hand.

**Theta, Phi, and Falloff**

These three members (the last of the bunch) are used only with spotlights. Anyone who remembers their Algebra II class can tell that theta and phi, because they are Greek letters, are angles of some kind. I’ve never been sure why Greek letters have been used for angles. It’s just one of those things.

The \( \Phi \) member contains the outer cone size. This value has to be between 0 and \( \pi \). The lower values make for a tighter cone, and the higher values make for a wider cone. The spotlight only has an effect within this angle.

The \( \Theta \) member contains the inner cone size. This value has to be between 0 and the value placed into \( \Phi \). The inner cone of a spotlight has full intensity. Outside of the inner cone and towards the outer cone, the intensity decreases towards zero, according to the value of the Falloff member.

The math for the Falloff member is rather complex. If the value is 1.0, then from the inner cone to the outer cone is a smooth, linear decrease in intensity. If the value is higher than 1.0, then the intensity starts to decrease sharply at first, and then slows down as it approaches zero. If the value is lower than 1.0, the intensity starts to decrease slowly at first, and then speeds up as it approaches zero. The documentation says that values other than 1.0 are hard to notice (“the effect is subtle”), and suggests using 1.0 at all times. Because values other than 1.0 incur a performance penalty, this is probably a good idea.
The spotlight members are perhaps the most physically accurate part of the description of a light. If you shine flashlight against a wall, you can see that there is a part of the light that is brighter (the inner cone), and a part of the light that slowly dims to nothing (the outer cone). If you have maglight (a light where you can change the angle in which light is emitted), you can even change these values by twisting the flashlight.

**Setting up Lights**

Setting up a light is the hardest part. After that, it’s a simple matter of telling Direct3D what the light is and then turning it on or off.

The IDirect3DDevice8::SetLight function puts the light into the device’s list of lights.

```cpp
HRESULT IDirect3DDevice8::SetLight (
    DWORD dwIndex ,
    CONST D3DLIGHT8* pLight 
);
```

This function returns D3D_OK on success and D3DERR_INVALIDCALL on failure. You can create as many lights as you like. However, Direct3D can only use eight lights at a time, so be frugal. The dwIndex parameter is the number of the light you are setting (the lowest allowed value is 0) and pLight is a pointer to the light you want to have at this index.

To retrieve a light, you can use IDirect3DDevice8::GetLight.

```cpp
HRESULT IDirect3DDevice8::GetLight (
    DWORD dwIndex ,
    D3DLIGHT8* pLight 
);
```

This function is much like IDirect3DDevice::SetLight. The parameters and return values are nearly the same. The only difference is that pLight is the destination for the light information at that index.

After you have set a light, it is still off. Using IDirect3DDevice::SetLight is sort of like plugging in a lamp. It is installed, but not in use. You have to flip the switch to turn it on. Direct3D’s version of flipping the switch is making use of the IDirect3DDevice8::LightEnable function.
HRESULT IDirect3DDevice8::LightEnable (
    DWORD dwIndex ,
    BOOL bEnable
) ;

This function returns D3D_OK on success and D3DERR_INVALIDCALL on failure. The dwIndex is the index to the light and bEnable is non-zero if you want the light to be on, or zero if you want the light off.

On the flip side, you can determine whether a light is enabled by calling IDirect3DDevice8::GetLightEnable.

HRESULT IDirect3DDevice8::GetLightEnable (
    DWORD dwIndex ,
    BOOL* bEnabled
) ;

This function returns D3D_OK on success and D3DERR_INVALIDCALL on failure. The dwIndex parameter is again the index of the light. The bEnabled parameter is a pointer to a BOOL that gets filled in with the current enabled state of the light. An enabled light is non-zero and a disabled light is zero.

## Point Lights

You will likely use points lights or directional lights above spotlights much of the time, unless you are in need of some special effects. To demonstrate point lights, I have thrown together a small example. Example 13.1 (X:\Source\Example13_1) on the CD shows a single point light shining onto the checkerboard used in Chapter 11.

The original set up for the point light looks like this.

```c
memset ( &g_Light , 0 , sizeof ( D3DLIGHT8 ) ) ;
g_Light.Type = D3DLIGHT_POINT ;
g_Light.Diffuse.r = 1.0f ;
g_Light.Diffuse.g = 1.0f ;
g_Light.Diffuse.b = 1.0f ;
g_Light.Position.x = 4.0f ;
g_Light.Position.y = 4.0f ;
g_Light.Position.z = -5.0f ;
```
g_Light.Attenuation0 = 1.0f ;
g_Light.Attenuation1 = 0.0f ;
g_Light.Attenuation2 = 0.0f ;
g_Light.Range = sqrt ( FLT_MAX ) ;

You know that in the checkerboard example (at least the first one),
the world coordinates ranged from 0.0 to 8.0 for x and y, and z is zero.
Initially, the point light exists at (4,4,-5), which is in the middle of the
board, but a small distance away in the z direction. The light is pure
white (you can try other values if you’re interested in seeing how they
turn out) and attenuation is constant.

This is not a static demo. You can use the arrow keys to move the light
around. You can use the PageUp and PageDown keys to move the
light away and towards the board. Also, you can see out how the differ-
ent attenuation constants affect the light shining onto the board.
Pressing 1, 2, or 3 changes from \texttt{Attenuation0} to \texttt{Attenuation1} to
\texttt{Attenuation3}. Figure 13.1 shows this in action.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{point_light_example.png}
\caption{Point light example}
\end{figure}
Note that there is no longer a diffuse color component of the CustomVertex structure. Instead, two materials represent the different squares of the board. I’m still only using diffuse light at this moment. Also, the CustomVertex structure now has a normal vector. In the case of the board, all of these are (0, 0, -1), because the squares are oriented in the same direction.

**Directional Lights**

The directional light example isn’t nearly as impressive as the point light example, mainly because everything in the scene is facing the same way. Directional lights are much cooler-looking in 3D scenes. The initial setup for this direction light looks like the following:

```c
memset ( &g_Light , 0 , sizeof ( D3DLIGHT8 ) ) ;
g_Light.Type = D3DLIGHT_DIRECTIONAL ;
g_Light.Diffuse.r = 1.0f ;
g_Light.Diffuse.g = 1.0f ;
g_Light.Diffuse.b = 1.0f ;
g_Light.Direction.x = 0.0f ;
g_Light.Direction.y = 0.0f ;
g_Light.Direction.z = 1.0f ;
```

Setting up a directional light is rather easy, because it has no position and needs no attenuation (because it has no position, objects cannot be distant from it, nullifying the need for any sort of attenuation). Like the point light, this light is pure white.

In any case, Example 13.2 (X:\Source\Example13_2) shows a small example of using a directional light. The only controls are the arrow keys, and they simply rotate the direction around the x and y axes, respectively. The effect is underwhelming. As the direction moves away from the normal of the board, the light just dims.

**Spotlights**

Now spotlights are just neat. Much more impressive than either point lights or directional lights. However, due to the fact that the board has rather large squares, a spotlight may look a bit blocky. For this reason,
you should probably combine a spotlight with some other kind of dim light.

Example 13.3 (X:\Source\Example13_3) demonstrates a spotlight on a checkerboard. This example has the most complicated setup for a light thus far.

```cpp
memset (&g_Light, 0, sizeof (D3DLIGHT8));
g_Light.Type = D3DLIGHT_SPOT;
g_Light.Diffuse.r = 1.0f;
g_Light.Diffuse.g = 1.0f;
g_Light.Diffuse.b = 1.0f;
g_Light.Position.x = 4.0f;
g_Light.Position.y = 4.0f;
g_Light.Position.z = -10.0f;
g_Light.Direction.x = 0.0f;
g_Light.Direction.y = 0.0f;
g_Light.Direction.z = 1.0f;
g_Light.Attenuation0 = 1.0f;
g_Light.Attenuation1 = 0.0f;
g_Light.Attenuation2 = 0.0f;
g_Light.Range = (float)sqrt (FLT_MAX);
g_Light.Falloff = 1.0f;
g_Light.Phi = PI / 6.0;
g_Light.Theta = PI / 9.0;
```

Because a spotlight has both position and direction, both must be filled in. I placed the spotlight at the center of the board, but a little farther away than the point light (to avoid some of the blockiness associated with the squares). The direction points right towards the board. Attenuation starts out as constant. I place the falloff at 1.0, and that cannot be changed with the program’s controls. Finally, the outer cone is PI/6 (30 degrees), and the inner cone is PI/9 (10 degrees).

Just about everything can be changed with the controls for this program. All of the same controls (arrow keys, Pageup, Pagedown, and 1, 2, and 3) from the point light example are still effective here. In addition, there are a number of other controls that affect the properties particular to the spotlight. By pressing Q and W, you can modify the size of the outer cone. By pressing A and S, you can change the inner cone’s size. By using I, J, K, and L, you can change the direction of the
spotlight (they act sort of like auxiliary arrow keys). Figure 13.2 shows an example of this.

**Ambient Light**

One of the other three types of light, as far as Direct3D is concerned, is ambient light. It is rather easy to use (just as easy as diffuse light, and sometimes even easier).

The simplest way to use ambient light is to set a render state called `D3DRS_AMBIENT`. This render state is a `DWORD` that represents a color. If, for example, you wanted all ambient light in your program to be `D3DCOLOR_XRGB (128, 128, 128)`, you would do this:

```c
  g_pd3ddev->SetRenderState ( D3DRS_AMBIENT , D3DCOLOR_XRGB ( 128 , 128 , 128 ) ;
```

In order for the ambient light to show up, of course, you have to fill out the members of your `D3DMATERIAL8`'s Ambient color value. These val-
ues can be different than the diffuse values. If you do this, the object will appear differently in ambient light than in diffuse light, and different again when both types of light are combined.

Example 13.4 (X:\Source\Example13_4) shows a simple example of using ambient light in this manner. This example is similar to Example 13.1, with all of the same controls as that example, and two new controls: F1 turns the ambient light to dark gray, and F2 turns the ambient light to black. Figures 13.3 and 13.4 show the differing output when ambient lighting is off and on.

Another way to do ambient light is to set up the Ambient member of the D3DLIGHT8. It works in almost exactly the same way as using the render state. Example 13.5 (X:\Source\Example13_5) demonstrates this method of doing ambient light. The code for setting up the light is as follows:

```c
memset ( &g_Light , 0 , sizeof ( D3DLIGHT8 ) ) ;
g_Light.Type = D3DLIGHT_POINT ;
g_Light.Diffuse.r = 1.0f ;
g_Light.Diffuse.g = 1.0f ;
g_Light.Diffuse.b = 1.0f ;
g_Light.Ambient.r = 0.5f ;
g_Light.Ambient.g = 0.5f ;
g_Light.Ambient.b = 0.5f ;
g_Light.Position.x = 4.0f ;
g_Light.Position.y = 4.0f ;
g_Light.Position.z = -5.0f ;
g_Light.Attenuation0 = 1.0f ;
g_Light.Attenuation1 = 0.0f ;
g_Light.Attenuation2 = 0.0f ;
g_Light.Range = sqrt ( FLT_MAX ) ;
```

Again, this is the point light example, modified to meet the new needs. Some of the controls have changed as well. The F1, F2, and F3 keys now turn on and off the red, green, and blue components, respectively, of the ambient light. When all three components are on the board looks much like Figure 13.4. When all three components are off the board looks much like Figure 13.3. When some are on and others are off you can see some interesting color shades.

Ambient light, as you have seen, is pretty easy to implement if you already know how diffuse light is implemented. Also, you can achieve
**Figure 13.3**  CD example 13.4, ambient off

**Figure 13.4**  CD example 13.4, ambient on
some pretty neat effects when combining it with diffuse light. Ambient
light is something you’re going to want to experiment with in exam-
ple programs.

**Specular Light**

Specular light is not as easy to use as diffuse or ambient light. It makes
things shiny, so it isn’t necessary for all situations. It’s also really, really
weird, so for this part, I’m going to show you how to set it up, show
you an example, and then leave you to experiment with it.

First off, you need to set the D3DRS_SPECULARENABLE render state to TRUE. Without it, specular highlighting is off.

Next, you need to set up the Specular members of your D3DMATERIAL8
and D3DLIGHT8 structures. Additionally, you have to fill in the D3DMATER-
IAL8’s Power member to something that isn’t 0. After that, you can go
ahead and use the light, and you’ll have specular highlights.

I have placed the code to demonstrate specular highlighting in
Example 13.6 (X:\Source\Example13_6). It is again the point light
demo, this time with specular lighting enabled. The code for setting
up the materials and light is as follows:

```c
//set up materials
memset ( &g_Material [ 0 ] , 0 , sizeof ( D3DMATERIAL8 ) ) ;
memset ( &g_Material [ 1 ] , 0 , sizeof ( D3DMATERIAL8 ) ) ;
g_Material [ 0 ].Diffuse.r = 0.75f ;
g_Material [ 0 ].Diffuse.g = 0.75f ;
g_Material [ 0 ].Diffuse.b = 0.75f ;
g_Material [ 0 ].Ambient.r = 0.75f ;
g_Material [ 0 ].Ambient.g = 0.75f ;
g_Material [ 0 ].Ambient.b = 0.75f ;
g_Material [ 0 ].Specular.r = 0.75f ;
g_Material [ 0 ].Specular.g = 0.75f ;
g_Material [ 0 ].Specular.b = 0.75f ;
g_Material [ 0 ].Power = 2.0f ;
g_Material [ 1 ].Diffuse.r = 0.5f ;
g_Material [ 1 ].Diffuse.g = 0.5f ;
g_Material [ 1 ].Diffuse.b = 0.5f ;
g_Material [ 1 ].Ambient.r = 0.5f ;
```
g_Material[1].Ambient.g = 0.5f;
g_Material[1].Ambient.b = 0.5f;
//set up light
memset(&g_Light, 0, sizeof(D3DLIGHT8));
g_Light.Type = D3DLIGHT_POINT;
g_Light.Specular.r = 1.0f;
g_Light.Specular.g = 0.0f;
g_Light.Specular.b = 0.0f;
g_Light.Diffuse.r = 1.0f;
g_Light.Diffuse.g = 1.0f;
g_Light.Diffuse.b = 1.0f;
g_Light.Ambient.r = 0.5f;
g_Light.Ambient.g = 0.5f;
g_Light.Ambient.b = 0.5f;
g_Light.Position.x = 4.0f;
g_Light.Position.y = 4.0f;
g_Light.Position.z = -5.0f;
g_Light.Attenuation0 = 1.0f;
g_Light.Attenuation1 = 0.0f;
g_Light.Attenuation2 = 0.0f;
g_Light.Range = sqrt(FLT_MAX);

I haven’t included a figure here because this effect doesn’t translate well to black and white. Run the example and see for yourself. At certain distances some of the squares of the board have a pinkish glow about them.

**Emissive Light**

And finally, you come to emissive light. By now you can probably guess how to use it. Just as you do with any other type of light, you set up the material’s Emissive member. Except there is no Emissive member of D3DLIGHT8.

Well, the only thing to do is to try it out. Example 13.7 (X:\Source\Example13_7) uses emissive light on one of the materials; the other material doesn’t have emissive light. If you run it, you will see something that looks like Figure 13.5. Half of the squares seem to glow with a white light. The other half responds to the light normally.
Summary

As you have seen, there is a great deal you can do with Direct3D’s lighting. Lighting is one of the easier variables to implement. Of course you’ll need to experiment, but I’m confident you can find some unusual and graphically stunning uses for this aspect of Direct3D.
CHAPTER 14

Alpha Blending
A discussion of Direct3D (or any 3D API, for that matter) just isn’t complete without a section on alpha blending. You learned about alpha a little bit already, but only in the context of alpha testing, which does not bring into play the interesting effects you can achieve with translucency. This chapter takes a look at alpha blending and some of the neat stuff you can do with it. Your objectives for this chapter are to:

- Know what alpha blending is
- Know how to set up for alpha blending
- Know how to use alpha blending

**Alpha Blending Defined**

The main purpose of alpha blending is to create the illusion of translucency. You can see through translucent objects, but they aren’t totally invisible. A good example is colored glass. You can see through the glass, but whatever is behind it is modified by the color of the glass through which you’re looking.

And so, enter the importance of the alpha component of a color. Although alpha blending is customizable, as all things are in Direct3D, it typically involves combining two images. The alpha components are between 0.0 and 1.0, and these are represented as color values between 0 and 255. The low value is completely transparent, and the high value is completely opaque, so the values in between are somewhat translucent.

You can think of the properties of an object as being a certain percentage transparent and a certain percentage opaque. These two percentages add up to 100 percent. If something is 100 percent opaque, it is 0 percent transparent, and if it is 100 percent transparent, it is 0 percent opaque.
However, consider an object that is 25 percent opaque and 75 percent transparent, like a lightly tinted blue glass.

The blue glass has 0.0 for red and green, and 1.0 for blue. It is 25 percent opaque, so its alpha component is 0.25. Translated into the normal colors, it looks like this:

```c
DWORD dwColorGlass = D3DCOLOR_ARGB(64, 0, 0, 255);
```

Now, try putting it in front of some other object, like something that is a pastel reddish color. The color of the object behind the glass is:

```c
DWORD dwColor = D3DCOLOR_XRGB(255, 128, 128);
```

Notice there is no alpha component for this color, because it is completely opaque. Internally, it is represented with a 255 for its alpha component.

To mix these colors, take the color behind the alpha blended object (the reddish color) and multiply it by the transparent value (1.0; the opaque value). Then take the alpha blended color and multiply it by the opaque value. Finally, add these two colors together to get the final blended color.

The pastel color is multiplied by approximately 3/4, so you will get red, green, and blue components of approximately 192, 96, and 96, respectively.

The blue glass color is multiplied by approximately 1/4, so you will get red, green, and blue components of about 0, 0, and 64, respectively.

Next, you add them, and so you get red, green, and blue components of 192, 96, and 160, which is sort of a pastel lavanderish color. This is what you’d expect if you looked at a pastel red object through blue tinted glasses.

Of course, this isn’t all that you can do with alpha blending, just a common example of it. When doing just about any sort of alpha blending, you multiply the destination color by a certain value, multiply the source color by a certain value, and perform some sort of arithmetic combination of the source and destination colors.
Setting up for Alpha Blending

I’ve got good news and bad news for you. The good news is that setting up alpha blending is very easy. The bad news is that if you are using alpha blending, you cannot use z buffering, because z buffering will screw it up. You must instead sort your own objects from far to near. When using Direct3D to do 3D, this is something of a pain. Because you are using Direct3D for 2D and are often sorting from back to front anyway, it’s a bit simpler.

Setting up alpha blending for a Direct3D device is simply a matter of render states. There are four of these: D3DRS_ALPHABLENDENABLE, D3DRS_SRCBLEND, D3DRSDESTBLEND, and D3DRS_BLENDOP. Their uses are fairly obvious, but let’s take this one step at a time.

The first is D3DRS_ALPHABLENDENABLE. As with most of the other “ENABLE” render states, you set the value to either TRUE or FALSE. I appreciate such consistency, don’t you?

So, to turn on alpha blending, do the following:

```c
    g_pd3ddev->SetRenderState ( D3DRS_ALPHABLENDENABLE , TRUE ) ;
```

And to turn it off, simply replace the TRUE with a FALSE. There, that was easy.

The next two render states take a little more planning and forethought before use. They are D3DRS_SRCBLEND and D3DRS_DESTBLEND. Essentially, these render states control the values used to multiply the source and destination colors before performing the actual blending.

This render state expects a value from the D3DBLEND enumeration. There are several of these, and they are listed and briefly explained in Table 14.1.

Note that in Table 14.1, the word *inverted* is used a lot. In alpha blending, each component is a value between 0.0 and 1.0 (even though they are between 0 and 255). The inverted value is simply 1.0 minus the original value. So, if you have a red value of 0.6, the inverted red value is 0.4.
So, the default value for D3DRS_SRCBLEND is D3DBLEND_ONE, and the default value for D3DRS_DESTBLEND is D3DBLEND_ZERO. With this setup, you always get the value of the source pixel drawn, which is what you’ve been using up until now.

In the example with the blue glass and the red object behind it, you would set D3DRS_SRCBLEND to D3DBLEND_SRCALPHA, and D3DRS_DESTBLEND to D3DBLEND_INVSRCCOLOR. With these values, the blue would be multiplied by the source alpha (0.25), and the destination would be multiplied by 1-alpha (0.75), which is just the way you wanted.
With the various destination alpha components, be aware that if the format of the surface has no alpha component in it, the alpha at any particular point is always 1.0.

The third and final render state is D3DRS_BLENDOP. This render state expects a value from the D3DBLENDOP enumeration. Table 14.2 lists and briefly explains the constants in this enumeration.

The default value for D3DRS_BLENDOP is D3DBLENDOP_ADD.

## Table 14.2 D3DBLENDOP Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DBLENDOP_ADD</td>
<td>Add source and destination components.</td>
</tr>
<tr>
<td>D3DBLENDOP_SUBTRACT</td>
<td>Subtract destination components from source components.</td>
</tr>
<tr>
<td>D3DBLENDOP_REVSUBTRACT</td>
<td>Subtract source components from destination components.</td>
</tr>
<tr>
<td>D3DBLENDOP_MIN</td>
<td>Take the minimum of the source and destination components.</td>
</tr>
<tr>
<td>D3DBLENDOP_MAX</td>
<td>Take the maximum of the source and destination components.</td>
</tr>
</tbody>
</table>
Using Alpha Blending

Ready for an example? I’ve got a quick one. Example 14.1 is a simple demonstration of alpha blending, using D3DBLEND_SRCALPHA for the source blend, D3DBLEND_INVSRCALPHA for the destination blend, and D3DBLENDOP_ADD for the blending operation. It features two squares that rotate around the center of the screen and periodically overlap. The green square has an alpha of 1.0, and the red square has an alpha of 0.5. Figure 14.1 shows this example.

As with lighting, effective use of alpha blending has very much to do with experimentation. To get the effect you’re looking for you often have to go through the many blending operations and source and destination blends. The important point is that you now know what you have to do in order to start experimenting.

Although Example 14.1 might not seem visually impressive, you can always put a texture on the primitives if you want. That will make the effect much more interesting. You already know how to put textures on primitives, and doing so when using alpha blending is no different.
than doing so normally. You can even combine alpha blending with
alpha testing to create partially transparent alpha blended primitives.
All of these effects can work together, but in the example I just show
them to you individually.

Summary

One of the coolest alpha blending effects, in my opinion, is what I call
the “transporter” effect. You start with an object in one position with
an alpha of 1.0. You change the alpha over a number of frames until it
is down to 0.0, and it looks like the object has disappeared. You then
move the object and start increasing the alpha again until it reaches
1.0 again. It makes it look as though the object slowly disappears in
one location and reappears in another. And this is only one of the
many types of effects alpha blending makes easier for you.
PART THREE

Bringing It All Together
15 Models
16 Particle Systems
17 Point in a Poly
18 Wrap up
CHAPTER 15

MODELS
In 2D games, when you want to make an animated character or object, you need to create enough images so that the object can be viewed from many angles and usually in several poses. If you need a lot of detailed characters and objects, this quickly adds up to a lot of art. Art takes up space on the hard drive and it takes up video memory when it is in use.

But what if I told you that you could reduce the amount of art needed for your game so far that you only need one texture for each character type or object? This is one of the things that Direct3D can do. If you create a model of the object or creature, and apply a texture to that model, you can then use matrices to rotate, translate, scale, spindle, and mutilate the model, and you only have to work off one model for each pose (rather than a bunch of 2D images for each pose and each facing). And if you really wanted to get fancy, you could put together pieces of a creature (an arm here, a leg here, a head there) and only have to work off of a few models using skeletal animation.

Here are your objectives for this chapter:

- Understand what a model is
- Understand what an X file contains
- Know how to make a simplified X file loader

**What Is a Model?**

A *model*, in regards to Direct3D, contains a bunch of triangles that, when put together, approximate the appearance of an object or creature. Typically, a file contains information about a model’s vertices and what indexes to the vertices make up its polygons. In many cases it will also store information about the color of the vertices and the material and textures of the vertices. In short, a model has all the pieces of information you need in order to draw the object or creature in question.
The X Files

Okay. Forget about Mulder and Scully and the new guy. When I talk about X files, I’m talking about the file format used to describe models. It is a simple, text-based format. Most 3D modeling programs can export to an X file.

To start off, let’s examine an X file that contains a model of a cube. I created this model with a program called Nendo. Nendo might not be as full-featured as some other programs, but for a small-time developer it is more affordable.

```
xof 0302txt 0064
Header {
  1;
  0;
  1;
}
Material cube0_material {
  1.000000;1.000000;1.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
}
Mesh cube0 {
  24;
  -10.000000;10.000000;10.000000;;
  10.000000;10.000000;-10.000000;;
  10.000000;-10.000000;10.000000;;
  -10.000000;-10.000000;-10.000000;;
  10.000000;10.000000;10.000000;;
  -10.000000;10.000000;10.000000;;
  10.000000;-10.000000;10.000000;;
  -10.000000;10.000000;10.000000;;
  10.000000;10.000000;10.000000;;
  -10.000000;10.000000;-10.000000;;
  -10.000000;10.000000;-10.000000;;
  10.000000;10.000000;-10.000000;;
  -10.000000;10.000000;-10.000000;;
  10.000000;10.000000;-10.000000;;
  -10.000000;10.000000;-10.000000;;
}
```
-10.000000;-10.000000;-10.000000;;
10.000000;-10.000000;-10.000000;;
10.000000;-10.000000;10.000000;;
-10.000000;-10.000000;10.000000;;
10.000000;10.000000;-10.000000;;
10.000000;10.000000;10.000000;;
10.000000;-10.000000;10.000000;;
10.000000;-10.000000;-10.000000;;
-10.000000;10.000000;10.000000;;
-10.000000;10.000000;-10.000000;;
-10.000000;-10.000000;-10.000000;;
6;
4;0,1,2,3;;
4;4,5,6,7;;
4;8,9,10,11;;
4;12,13,14,15;;
4;16,17,18,19;;
4;20,21,22,23;;
MeshMaterialList {
1;
6;
0,
0,
0,
0,
0,
0;
{cube0_material}
}
MeshNormals {
24;
-0.639602;0.639602;0.426401;;
0.639602;0.639602;0.426401;;
0.639602;-0.639602;0.426401;;
-0.639602;-0.639602;0.426401;;
0.639602;0.639602;-0.426401;;
-0.639602;0.639602;-0.426401;;
The X Files
Ack! All of that for a cube? If you have that dreading feeling in the pit of your stomach, don’t worry. It’s not nearly as bad as it looks. You’ll get through this; really, you will.

The very first line in a typical X file looks like the one in the cube X file.

xof 0302txt 0064

If you’ve worked with file formats in the past, you probably have already guessed that this is the part of the file that identifies it as an X file. If not, now you know. There are four parts to this line, xof (trailing space included), 0302, txt (trailing space included again), and 0064. Each portion of this line has a meaning. The xof means that this is an X file (the of stands for object file). The 0302 is actually two parts, 03 and 02, which designate the major and minor version numbers of the format. 0302 indicates that this is a version 3.2 file. The txt indicates that this is the text version of the file as opposed to the binary version, which would have a bin. Finally, the 0064 specifies the number of bits in a float that are used. If you are loading an X file yourself, you don’t strictly have to follow this. You can use a float (32 bits) instead of a double (64 bits), and nobody will be able to tell the difference.
The rest of the file contains a bunch of things that look sort of like functions, like the few lines immediately following the first line.

Header {
  1;
  0;
  1;
}

This is called a template. There are a number of pre-defined templates that are used in making an X file. This one is called the header template. It describes some aspect of the model in the X file. In the case of the header template, the first 1 specifies the major version number, the 0 specifies the minor version number, and the second 1 is a flag. The meaning of these numbers is not particularly important.

Notice that a template starts with some sort of name, in this case Header, and then has an open brace ({}), followed by some data that is separated either by a comma or semicolon, and then a close brace (}). This is typical of templates. It declares what it is, and then defines them between braces.

An important thing to note is that the white space (newline characters and spaces) is not important here; it just aids in readability. The exact same header could all fit on a single line like so:

Header{1;0;1;}

However, that sort of thing becomes hard to read, and part of the reason for using text-based X files is to make them human-readable, or at least moderately so.

See? This isn't that hard when you take it in bite-sized chunks. Let's look at the next template:

Material cube0_material {
  1.000000;1.000000;1.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
}

The X Files

211
Okay. This is a Material definition template. Unlike the Header template, it gives a name to the object, in this case cube0_material. It has five lines within it, and the last one is blank. Because this is a Material, you might assume this has to do with the members of a D3DMATERIAL8 structure. If so, you are correct.

The first line after the brace contains the ambient color for the material, in RGBA format. Notice that there are two semicolons at the end of the line. The four numbers that make up the red, green, blue, and alpha components of the ambient color count as a single item, even though each of the components is separated by a semicolon as well. Because each component has a value of 1.0, it seems that the ambient color for this material is D3DCOLOR_RGBA (255, 255, 255, 255).

The next line, a value of 0.0 all by itself, is the specular power factor, which is to be placed in D3DMATERIAL8's Power member.

The next line is the specular color, but with no alpha component, just RGB. Again, two semicolons appear at the end. The next line is the emissive color, formatted in the same way as the specular color.

The next line, which is currently blank, would be a string containing the name of a texture, if you had one. If you did have a texture name, it would look something like this:

TextureFilename { "cube0.bmp"; }  

This is like a template within a template. There are quite a few of these, especially with the Mesh template that you will see in a minute. These are optional factors in whatever is being described by the template. The TextureFilename template is reasonably straightforward; it contains a single string that is a filename which stores the texture used by the material in question.

The next template is the Mesh template. This one contains a lot of information, and several templates within it. Let’s start at the beginning: The first item is just a number, 24. This is the number of vertices in the mesh. After this number, there are 24 lines describing each of the vertices. Here’s a shortened look at what this part of the file looks like:

24;  
-10.000000;10.000000;-10.000000;,,  
10.000000;10.000000;-10.000000;,,
The 24 is immediately followed by a semicolon, indicating that it is a piece of data all on its own. The next 24 lines of the file contain three numerical values, each of which ends with a semicolon, but the line itself ends with a comma except for the 24th line. This is how X files indicate an array of values for something. You have likely guessed the meaning of each of the lines in the vertex array. They are the x, y, and z coordinates for each of the vertices.

But why 24? Last time I counted, a cube has a total of eight vertices. The reason you have 24 vertices is because of lighting, indirectly. In order for lighting to work on your model, each vertex must have a normal vector. However, each vertex is part of three faces of the cube, and so each vertex doesn’t have just one normal vector, it has three. So, eight vertices, three normals a piece, for a total of 24 vertices. Get it? Another way to look at it is that the model doesn’t model the cube as a whole, but each of the faces instead. There are six faces, and each face has four vertices, and six times four is again 24. In a smoother type of object, like a sphere, you might have only one normal per vertex, but not so for a cube.

When reading in an X file, the best place for this data is in a vertex buffer or in an array of vertex data, if that’s what you plan on using.

The next part of the mesh description describes each of the faces of the model, the polygons.

6;
4;0,1,2,3;;
4;4,5,6,7;;
4;8,9,10,11;;
4;12,13,14,15;;
4;16,17,18,19;;
4;20,21,22,23;;

If you’ve been following along, you have probably determined the logic behind the format of an X file, and you know that the 6 must be
a number of elements, and each of the other lines is one of the items in that element. That’s exactly how this part of the mesh works. First is the number of faces, followed by a description of each face.

However, each of the elements in the array of faces is also an array, just to add something more interesting. Take the second line:

4;0,1,2,3;;

This array contains the number of vertices in the face (4), followed by the indexes into the vertex list (0, 1, 2, 3) that make up this face. For this simple cube model each face consists of four vertices. In Direct3D, however, you can only render triangles, so you’ll have to take this and divide it into triangles yourself. If you wanted to just have faces with three sides you would need to divide the model into triangles. In the case of a cube this would give you a total of 36 vertices and 12 faces. There are a number of ways to model a cube, it seems.

A polygon with four vertices isn’t really that hard to divide into triangles, though. You can simply use 0, 1, 2 and 2, 3, 0 as the two triangles. Doing more than four vertices on a face is a little more complicated, so for the purposes of loading X files manually, you’ll limit the number of vertices to four per side for the loader. If you want more capability, there are D3DX objects for loading in X files; you’ll read about that in the next chapter.

The next part of the mesh is the MeshMaterialList template. With the single cube you have only one material, so this part of the file isn’t very interesting but is still important. Here’s another look at it:

MeshMaterialList {
    1;
    6;
    0,
    0,
    0,
    0,
    0;
    {cube0_material}
}
The first line of this template is just a 1. That specifies how many materials are in use in the entire mesh. In the case of this simple cube (which winds up being anything but simple as far as the X file is concerned), you have only one. The next element, the 6, is the number of faces. It is followed by an array of indexes into the material list (the first and only material being indexed as a 0). It’s pretty easy to see that if you had a cube with three differently-colored sides, you could easily have three materials, and the numbers in the array would range from 0 to 2.

The last part of this mesh contains the vertex colors, in the MeshVertexColors template, which I have abbreviated here:

```plaintext
MeshVertexColors {
  24;
  0;1.000000;1.000000;1.000000;1.0;,
  1;1.000000;1.000000;1.000000;1.0;,
  2;1.000000;1.000000;1.000000;1.0;,
  ...,
  21;1.000000;1.000000;1.000000;1.0;,
  22;1.000000;1.000000;1.000000;1.0;,
  23;1.000000;1.000000;1.000000;1.0;,
  ;
}
```

You’re probably beginning to notice a pattern. The MeshVertexColors template contains an array. The first line is the number of elements in that array—24 in this case (one for each vertex). The individual items in this array contain five numbers. The first number is an index into the vertex array, and ranges from 0 to 23. The second, third, and fourth correspond to the red, green, and blue values for the vertex. The last number is the alpha value for the vertex.

What this X file contains is a very detailed mathematical description of the model (the cube). Even in a simple model like this one, there is a lot of mathematical information to rebuild it. The main goal when creating a model-loading program is to take the information in the file and convert it into some data that is understandable to the computer, and more importantly, to the API you’re using.
Making a Simple X File Loader

Even though you’re going to spend some time making an (extremely simplified) X file loader, I’m not actually suggesting that you build one yourself—just as I have never suggested making your own .BMP file loader, because there are already functions for doing so. Treat this more as an exercise in appreciating all of the work that goes into loading an X file into memory.

Because you’re dealing with text-based X files, you have to build a tokenizer/parser/lexer (something that takes the contents of a text file and gives it meaning to the computer). You have to take all textual data and often convert it into numerical data. Now’s the time to brush up on some of your own C function knowledge, such as `atoi` and `atof`. Those two functions, and the `strtok` function, are all you really need in order to make a simple model-loading program.

The STL Vector Template

Another bit of standard C++ knowledge that will be useful in this endeavor is knowing how to use the STL (Standard Template Library) vector template. Without going overboard, let’s take a moment to look at some of the more useful aspects of the STL vector.

First, inclusion. The header in which the vector template is found is in a file called “vector” (no .h), so including the capability to use the vector template is done like so:

```cpp
#include <vector>
```

Often, this will be followed by another line:

```cpp
using namespace std ;
```

Don’t worry if you have never used the `using` statement before. It’s just there to make the declarations look a bit cleaner.

Next, declaration. A vector acts and looks very much like an array. In fact, that’s the whole point of a vector (not to be confused with the
xyz vectors in Direct3D). Why is this template called vector and not something like “array”? I don’t know. Take it up with Bjarne.

So, back to declaration. If you need a vector that contains int values, this is what it might look like:

```cpp
vector < int > vecInts ;
```

Okay, not too bad, right? The bit in between the `<` and `>` signs is the type, so if you wanted to have a vector of something else, like a D3DMATERIAL8 structure, you could make something like that, too.

```cpp
vector < D3DMATERIAL8 > vecMaterials ;
```

If you wanted to confuse the issue, you could even have a vector of a vector of something, like so:

```cpp
vector < vector < int > > vecVecInts ;
```

A vector of a vector is like a two-dimensional array. You could in theory go on forever like this, and make vectors within vectors within vectors, but you get the idea.

When a vector is initially created, it is empty. You can check for emptiness with the member function of the same name, like so:

```cpp
if (vecMaterials.empty ( ) )
{
    //materials list is empty
}
```

To add an element to the end of the vector, you use the `push_back` member function, like so:

```cpp
vecMaterials.push_back ( material ) ;
```

To see how many elements are in the vector, you use the `size` member function.

```cpp
int nElementCount = vecMaterials.size ( ) ;
```

And finally, to randomly access a particular element, you use operator `[ ]`, just like you would an array.

```cpp
material = vecMaterials [ index ] ;  //index>=0 and index<vecMaterials.size ( )
```
And that’s about all you need in order to make use of vectors. If you’re interested in learning more, I suggest finding a good book on STL. Using the STL is a good way to save on development time, because you don’t have to implement the standard containers yourself.

**Getting the Data from File into Memory**

But back to making an X file loader. To start with, you’ll want to get all of the data from the file into memory, so that you can start working on it. For this I suggest the use of some standard WIN32 functions, because they are easy and will do the job just fine.

```c
//open the file for reading
HANDLE hFile = CreateFile ( "filename.x" , GENERIC_READ , 0 , NULL , OPEN_EXISTING , FILE_ATTRIBUTE_NORMAL , NULL ) ;
//retrieve the size of this file
DWORD dwFileSize = GetFileSize ( hFile , NULL ) ;
//allocate a string large enough to hold the entire contents of the file, plus the null terminator
char* pFileString = new DWORD [ dwFileSize + 1 ] ;
//clear out the string to all 0s
memset ( pFileString , 0 , dwFileSize + 1 ) ;
//read the file, place contents into the string
DWORD dwBytesRead = 0 ;
ReadFile ( hFile , pFileString , dwFileSize , &dwBytesRead , NULL ) ;
//close the file
CloseHandle ( hFile ) ;
```

After this code you’ll have the entire contents of the file in `pFileString`, and at that point you’ll be ready to start processing the contents. Ideally, you want to make something that could load any X file ever made, with error checking and so on. Unfortunately, you just don’t have that kind of time or the space to make it happen. Instead, you’ll simply make a loader that can load in the basic cube model you’ve been looking at this chapter. From there, it shouldn’t be too hard to start adding functionality. This is just to get you started.
Tokenizing Text

For one thing, you need to abstract some of the data found in an X file into some sort of containing structure or class. For now, you can skip over the first 16 bytes of the file, like so:

```c
char* pPosition = pFileString + 16;
```

The first 16 bytes, as you recall, are the things that make the X file an X file in the first place. In a more robust loader, you would make sure that the header was valid and so on.

Now you need to start tokenizing—taking a bunch of text and making it into words or tokens. You need to define the tokens and the separators. In the case of an X file, most of the tokens will be numbers (or at least string representations of numbers), as well as string values. Spaces, newline characters, braces, commas, and semicolons do not matter. You can set up `strtok` so that you get only numbers and strings, and everything else will be ignored.

```c
pToken = strtok ( pPosition , "\r\n\t ,;{}" ) ;
```

This line tells `strtok` to ignore all of the white space and punctuation characters, and that you want to start the tokenization at `pPosition`. A pointer to the first token is placed into `pToken`. It will point to a string that contains `Header`. After this initial token, the remaining tokens can be retrieved using `strtok` again, like so:

```c
while ( pToken )
{
    //do something with the token
    pToken = strtok ( NULL , "\r\n\t ,;{}" ) ;
}
```

Of course you won’t do it exactly like this, but this is how basic tokenization using `strtok` goes. You could even go as far as to put together a token list in a vector `< char* >` if you wanted to.

The rest of the task is placing meaning onto each of the tokens. Based on the rules of the templates in an X file, once you see `Header`, you can expect the next three tokens to be numerical values. In this case, these are 1, 0, and 1, which are the major, minor, and flag values (WORD,
WORD, and DWORD according to the documentation). To abstract the header, you might come up with a simple structure like so:

```c
struct XHDR
{
    WORD major;
    WORD minor;
    DWORD flags;
};
```

And you could have the very beginnings of an X file loading function.

```c
void LoadXFile ( char* sFileName )
{
    //code to load file into string goes here

    //data
    XHDR hdr;

    //start tokenizing
    char* sToken = strtok ( pPosition , "\r\n\t ,;{}" ) ;
    while ( sToken )
    {
        //which token is this?
        if ( strcmp ( sToken , "Header" ) == 0 )
        {
            LoadXHeader ( &hdr ) ;
        }
        //find next token
        strtok ( NULL , "\r\n\t ,;{}" ) ;
    }
}
void LoadXHeader ( XHDR* pheader )
{
    //major
    char* pMajor = strtok ( NULL , "\r\n\t ,;{}" ) ;
    pheader->major = atoi ( pMajor ) ;
    //minor
    char* pMinor = strtok ( NULL , "\r\n\t ,;{}" ) ;
```
pheader->minor = atoi ( pMinor ) ;
   //flags
   char* pFlags = strtok ( NULL , "\r\n\t ,;{}" ) ;
   pheader->flags = atoi ( pFlags ) ;

There’s nothing particularly useful about the header (at least as far as you’re concerned). In fact, if you wanted to, you could simply ignore it.

The next tokens that will come out of the file are Material, cube0_material, and 11 floating point numerical values that describe a material. You could add some code to the LoadXFile function, like so (the new code is shown in bold):

```c
void LoadXFile ( char* sFileName )
{
   //code to load file into string goes here

   //data
   XHDR hdr ;
   XMATERIAL mat ;

   //start tokenizing
   char* sToken = strtok ( pPosition , "\r\n\t ,;{}" ) ;
   while ( sToken )
   {
      //which token is this?
      if ( strcmp ( sToken , "Header" ) == 0 )
      {
         LoadXHeader ( &hdr ) ;
      }
      if ( strcmp ( sToken , "Material" ) == 0 )
      {
         LoadXMaterial ( &mat ) ;
      }
      //find next token
      strtok ( NULL , "\r\n\t ,;{}" ) ;
   }
}
```
This introduces a new type that I have not yet defined, called XMATERIAL. I see it as a structure containing a string (of the name of the material) and a D3DMATERIAL8 structure, like so:

```c
struct XMATERIAL
{
    char* sName;
    D3DMATERIAL8 material;
};
```

And you would next make the LoadXMaterial function, reading in the tokens one at a time and placing them in the respective parts of the XMaterial structure, like so:

```c
void LoadXMaterial ( XMATERIAL* pMaterial )
{
    //clear out structure
    memset ( pMaterial , 0 , sizeof ( XMATERIAL ) ) ;
    //name
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMaterial->sName = strdup ( sToken ) ;
    //read in data about the material
    //diffuse color
    //r
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMaterial->material.Ambient.r = atof ( sToken ) ;
    //g
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMaterial->material.Ambient.g = atof ( sToken ) ;
    //b
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMaterial->material.Ambient.b = atof ( sToken ) ;
    //a
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMaterial->material.Ambient.a = atof ( sToken ) ;
    //specular power
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMaterial->material.Power = atof ( sToken ) ;
    //specular color
    //r
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
}
```
pMaterial->material.Specular.r = atof ( sToken ) ;
//g
sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
pMaterial->material.Specular.g = atof ( sToken ) ;
//b
sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
pMaterial->material.Specular.b = atof ( sToken ) ;
//emissive color
//r
sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
pMaterial->material.Emissive.r = atof ( sToken ) ;
//g
sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
pMaterial->material.Emissive.g = atof ( sToken ) ;
//b
sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
pMaterial->material.Emissive.b = atof ( sToken ) ;
}

One thing is for certain—parsing a file makes for a whole lot of code. Handling the Material template (provided you don’t have to worry about the TextureFilename template), is just as straightforward as making the Header template loader.

The Mesh

After the material has been loaded, the next part of the file contains the mesh for the cube itself, so the next tokens you should expect to see are Mesh and cube0, followed by a whole lot of numbers. Because the mesh is broken up into smaller pieces, you can take them one at a time and work from there. First, you need some sort of structure into which you can store the mesh. For the time being, consider only the vertex list; you can add to it later.

You need two pieces of information in whatever structure you plan on storing the mesh. The first, like the XMATERIAL structure, is the name of the mesh itself. The second is an array of vertex data. For simplicity, you can define your own structure for storing vertices, like so:

```c
struct XVERTEX
{
    float x ;
}
float y;
float z;
};

Simple enough, right? Now, you can create a structure that can contain both the name of the mesh as well as an array of XVERTEX pointers. To do this, you can use an STL vector:

```c
struct XMESH
{
    char* sName;
    vector<XVERTEX*> vecVertices;
};
```

I can hear you cringing already. Don’t worry, this will be easy. You don’t have to be an STL expert to make use of STL.

You can add a few lines of code to the main X file loading function, like so (new code appears in bold):

```c
void LoadXFile ( char* sFileName )
{
/* code omitted*/
XMESH mesh;
// start tokenizing
char* sToken = strtok ( pPosition , "\r\n\t ,;{}" ) ;
while ( sToken )
{
    // which token is this?
    if ( strcmp ( sToken , "Header" ) == 0 )
    {
        LoadXHeader ( &hdr ) ;
    }
    if ( strcmp ( sToken , "Material" ) == 0 )
    {
        LoadXMaterial ( &mat ) ;
    }
    if ( strcmp ( sToken , "Mesh" ) == 0 )
    {
        LoadXMesh ( &mesh ) ;
    }
    // find next token
    strtok ( NULL , "\r\n\t ,;{}" ) ;
}```
The next logical step is to make a `LoadXMesh` function.

```c
void LoadXMesh ( XMESH* pMesh )
{
    //read in the name
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMesh->sName = strdup ( sToken ) ;
    //read in the number of vertices
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    int nVertexCount = atoi ( sToken ) ;
    //read the vertices themselves
    XVERTEX* pVertex ;
    while ( nVertexCount )
    {
        pVertex = new XVERTEX ;
        //x
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        pVertex->x = atof ( sToken ) ;

        //y
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        pVertex->y = atof ( sToken ) ;

        //z
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        pVertex->z = atof ( sToken ) ;
        //add vertex to list
        pMesh->vecVertices.push_back ( pVertex ) ;
        //decrement number of vertices to read.
        nVertexCount — ;
    }
}
```

Well, that wasn’t too painful, I hope. The only weird part is using the vector, which is only painful if you don’t often work with STL. You must admit, however, that it makes the task of storing an array of unknown size a lot easier.

But you aren’t done with the mesh yet! You only have a single part of it, the vertex list. This is going to be a little trickier than storing the
vertex list, because the face list is an array of faces, and the faces
themselves are an array of indexes into the vertex list. Rather than
working from the top down, you can design this from the bottom up,
by starting with a single face. You can store a single face in a vector,
and so make a type that contains a vector of DWORD (which is the type
used for indexes), like so:

```c
struct XFACE
{
    vector < DWORD > vecIndices ;
};
```

You might wonder why I have put nothing but a vector into a struc-
ture. Doing so allows you to avoid some otherwise very ugly declara-
tions. Now that you can contain a face in a structure, you can make a
function to read them in.

If you recall, a face looks something like the following:

```
4;0,1,2,3;,
```

Don’t count the number that indicates the number of faces in the
face list, because that’s handled later. You only have to deal with the
individual faces. The format is a number indicating how many indexes
there are, followed by that many indexes. That’s simple enough to
read in, right?

```c
void LoadXFace ( XFACE* pFace )
{
    //read in number of indexes
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    DWORD dwIndices = atoi ( sToken ) ;

    //read the indexes
    DWORD dwIndex ;
    while ( dwIndices )
    {
        //read an index
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        dwIndex = atoi ( sToken ) ;
        //add index to the list
        pFace->vecIndices.push_back ( dwIndex ) ;
        //decrease number of indexes left
```
Now you simply have to add the capability to store an array of XFACEs to the XMESH type, and you can do that by simply adding a vector < XFACE* > to your declaration of XMESH, like so (bold code is new code):

```c
struct XMESH
{
    char* sName ;
    vector < XVERTEX* > vecVertices ;
    vector < XFACE* > vecFaces ;
} ;
```

And then you add a provision to LoadXMesh that also reads in the face array, like so:

```c
void LoadXMesh ( XMESH* pMesh )
{
    //read in the name
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pMesh->sName = strdup ( sToken ) ;
    //read in the number of vertices
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    int nVertexCount = atoi ( sToken ) ;
    //read the vertices themselves
    XVERTEX* pVertex ;
    while ( nVertexCount )
    {
        pVertex = new XVERTEX ;
        //x
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        pVertex->x = atof ( sToken ) ;
        //y
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        pVertex->y = atof ( sToken ) ;
        //z
        sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
        pVertex->z = atof ( sToken ) ;
        //add vertex to list
```
pMesh->vecVertices.push_back ( pVertex ) ;
//decrement number of vertices to read.
nVertexCount -- ;
}

//load in the number of faces
sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
int nFaceCount = atoi ( sToken ) ;
XFACE* pFace ;
//while there are faces left
while ( nFaceCount )
{
   //allocate a new face
   pFace = new XFACE ;

   //load the face
   LoadXFace ( pFace ) ;
   //add the face to the list
   pMesh->vecFaces.push_back ( pFace ) ;
   //decrease the number of faces left
   nFaceCount -- ;
}

Now you’ve got the faces read in. The next bit is a little trickier, because it deals with an optional template. Because the optional templates can be in any order, and because they might not even be there at all, you have to be able to read them in any order, or completely skip them altogether. Also, it is likely that more than one mesh exists in the X file, and so the optional templates need a way to attach themselves to the appropriate mesh. The easiest way to handle this is to simply attach the information in any of the optional templates to the most recently read mesh.

So, keeping all of this in mind, let’s tackle the MeshMaterialLists template. It’s been a number of pages since you last looked at it, so here it is again.

MeshMaterialList {
  1;
  6;
  0,
  0,
The first token read is MeshMaterialList, which is added to the list of tokens to process in the main LoadXFile function. The code in bold is new.

```c
void LoadXFile ( char* sFileName )
{
    /* code omitted*/
    XMESH mesh ;
    //start tokenizing
    char* sToken = strtok ( pPosition , "\r\n\t ,;{}" ) ;
    while ( sToken )
    {
        //which token is this?
        if ( strcmp ( sToken , "Header" ) == 0 )
        {
            LoadXHeader ( &hdr ) ;
        }
        if ( strcmp ( sToken , "Material" ) == 0 )
        {
            LoadXMaterial ( &mat ) ;
        }
        if ( strcmp ( sToken , "Mesh" ) == 0 )
        {
            LoadXMesh ( &mesh ) ;
        }
        if ( strcmp ( sToken , "MeshMaterialLists" ) == 0 )
        {
            AddXMaterialList ( &mesh ) ;
        }
        //find next token
        strtok ( NULL , "\r\n\t ,;{}" ) ;
    }
}
```
Here you are adding information to the mesh, rather than loading the material list all on its own, because the material list is a part of the mesh. This means you have to add some stuff to the XMESH type, but before you do that you can determine what you need to do to store the information in the mesh material list.

The first token after the template name is the number of materials used. The materials are listed by name at the end of the template. This means you need some way of storing an array of strings. vector < char* > works well for something like that.

The second token is the number of faces again, followed by an index into the material array for each face. This is no big deal to read in, because it’s just like the index array of XFACE. So, the XMATERIALLIST type should probably look something like this:

```c
struct XMATERIALLIST
{
    vector < char* > vecNames ;
    vector < DWORD > vecMaterials ;
} ;
```

And you can add an XMATERIALLIST* to XMESH, like so:

```c
struct XMESH
{
    char* sName ;
    vector < XVERTEX* > vecVertices ;
    vector < XFACE* > vecFaces ;
    XMATERIALLIST* MaterialList ;
} ;
```

You indicate the lack of a material list with a NULL value in the XMESH structure. Now for a function that reads in a material list:

```c
void LoadXMaterialList ( XMATERIALLIST* pMaterialList )
{
    //read in number of materials used
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    DWORD dwMaterialCount = atoi ( sToken ) ;
    //read in number of faces
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    DWORD dwFaceCount = atoi ( sToken ) ;
    //read in the face
while ( dwFaceCount )
{
    // read in material ID
    sToken = strtok ( NULL, "\r\n\t ;,{}" ) ;
    DWORD dwMaterialID = atoi ( sToken ) ;
    // add to list of materials
    pMaterialList->vecMaterials->push_back ( dwMaterialID ) ;

    // decrease number of faces left
    dwFaceCount — ;
}

// read in the names of the materials
while ( dwMaterialCount )
{
    // read in name of material
    sToken = strtok ( NULL, "\r\n\t ;,{}" ) ;
    // duplicate string and add to name list
    pMaterialList->vecNames.push_back ( strdup ( sToken ) ) ;

    // decrease number of materials left
    dwMaterialCount — ;
}
}

Now you have to add the AddXMaterialList function, so that you can attach the material list to the XMESH.

void AddXMaterialList ( XMESH* pMesh )
{
    // allocate a new XMATERIALLIST
    pMesh->MaterialList = new XMATERIALLIST ;
    // load in the material list
    LoadXMaterialList ( pMesh->MaterialList ) ;
}

And now you’ve only got one left, the MeshVertexColors. For this one, you can create two new types, XVERTEXCOLOR and XVERTEXCOLORLIST. The XVERTEXCOLOR type will contain an individual piece of data, like so:

struct XVERTEXCOLOR
{
    DWORD dwVertexID ;
Reading in one of these is pretty simple.

```c
void LoadXVertexColor ( XVERTEXCOLOR* pVertexColor )
{
    //read in vertex ID
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pVertexColor->dwVertexID = atoi ( sToken ) ;
    //read in red
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pVertexColor->r = atof ( sToken ) ;
    //read in green
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pVertexColor->g = atof ( sToken ) ;
    //read in blue
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pVertexColor->b = atof ( sToken ) ;
    //read in alpha
    sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    pVertexColor->a = atof ( sToken ) ;
}
```

The `XVERTEXCOLORLIST` simply contains a vector of `XVERTEXCOLOR*`, like so:

```c
struct XVERTEXCOLORLIST
{
    vector < XVERTEXCOLOR* > vecVertexColors ;
} ;
```

And loading in this list is just like loading in any of the other lists you have dealt with so far.

```c
void LoadXVertexColorList ( XVERTEXCOLORLIST* pVertexColorList )
{
    //read in number of colors
    char* sToken = strtok ( NULL , "\r\n\t ;,{}" ) ;
    DWORD dwColors = atoi ( sToken ) ;
    //read in the colors
```
XVERTEXCOLOR* pVertexColor ;
while ( dwColors )
{
    pVertexColor = new XVERTEXCOLOR ;
    //read in color
    LoadXVertexColor ( pVertexColor ) ;
    //add to list
    pVertexColorList->vecVertexColors.push_back ( pVertexColor ) ;
    //decrease number of colors left
}

Now you need to add an XVERTEXCOLORLIST* to XMESH, like so:

struct XMESH{
    char* sName ;
    vector < XVERTEX* > vecVertices ;
    vector < XFACE* > vecFaces ;
    XMATERIALLIST* MaterialList ;
    XVERTEXCOLORLIST* VertexColorList ;
} ;

Like MaterialList, if the template doesn’t exist it’ll just be NULL.

Finally, you add the capability to add an XVERTEXCOLORLIST to the XMESH with a simple AddXVertexColorList function, modeled after the AddXMateriAllList function.

void AddXVertexColorList ( XMESH* pMesh )
{
    //create a new vertex color list
    pMesh->VertexColorList = new XVERTEXCOLORLIST ;
    //load the vertex color list
    LoadXVertexColorList ( pMesh->VertexColorList ) ;
}

You then add a call to this function in the LoadXFile function.

void LoadXFile ( char* sFileName )
{
    /* code omitted*/
    XMESH mesh ;
Grouping the Types

Now you’ve got all the code necessary to get the darn thing loaded, but you still aren’t quite done yet because you need to group all of the various types—XMESH, XMATERIAL, XHDR—into a single type so that you can have all of the data in one place, and also support more than one material and mesh.

struct XFILE
{
    XHDR Header ;
}
And finally, you need to adapt the `LoadXFile` function so that it uses this type.

```cpp
void LoadXFile ( char* sFileName , XFILE* pXFile )
{
    //open the file for reading
    HANDLE hFile = CreateFile ( "filename.x" , GENERIC_READ , 0 , NULL ,
OPEN_EXISTING , FILE_ATTRIBUTE_NORMAL , NULL ) ;
    //retrieve the size of this file
    DWORD dwFileSize = GetFileSize ( hFile , NULL ) ;
    //allocate a string large enough to hold the entire contents of the
    //file, plus the null terminator
    char* pFileString = new DWORD [ dwFileSize + 1 ] ;
    //clear out the string to all 0s
    memset ( pFileString , 0 , dwFileSize + 1 ) ;
    //read the file, place contents into the string
    DWORD dwBytesRead = 0 ;
    ReadFile ( hFile , pFileString , dwFileSize , &dwBytesRead , NULL ) ;
    //close the file
    CloseHandle ( hFile ) ;
    //temporary materials and mesh
    XMATERIAL* pMaterial ;
    XMESH* pMesh ;
    //start tokenizing
    char* sToken = strtok ( pPosition , "\r\n\t ,;{}" ) ;
    while ( sToken )
    {
        //which token is this?
        if ( strcmp ( sToken , "Header" ) == 0 )
        {
            LoadXHeader ( &pXFile->hdr ) ;
        }
        if ( strcmp ( sToken , "Material" ) == 0 )
        {
            pMaterial = new XMATERIAL ;
            LoadXMaterial ( pMaterial ) ;
            pXFile->vecMaterials.push_back ( pMaterial ) ;
        }
    }
}
```
if ( strcmp ( sToken , "Mesh" ) == 0 )
{
    pMesh = new XMESH;
    LoadXMesh ( pMesh ) ;
    pXFile->vecMeshes.push_back ( pMesh ) ;
}
if ( strcmp ( sToken , "MeshMaterialList" ) == 0 )
{
    AddXMaterialList ( pMesh ) ;
}
if ( strcmp ( sToken , "VertexColorList" == 0 )
{
    AddXVertexColorList ( pMesh ) ;
}
//find next token
strtok ( NULL , "\r\n\t ,;{}" ) ;
}

Blech! We’ve been looking at a lot of code lately. Wasn’t this supposed to be a simple X file loader? Unfortunately, nothing about X files is simple, and so the loader has to be complicated beyond all reason. Even now that you have a way to completely load the information (for the limited number of templates that are in the simple cube X file), you still don’t have it in the format that Direct3D needs to understand it. You next have to move the vertex and index data into a vertex buffer and index buffer, or into arrays that you use with the IDirect3DDevice8::DrawIndexedPrimitiveUP function.

I’m leaving the rest of the task up to you. You know how to load vertex buffers and index buffers already. It’ll be a good exercise.

Summary

As you have seen, loading in an X file is not trivial. Of course, nobody ever said you have to load it by yourself. That’s why a number of D3DX classes and functions exist. For more information about those classes, take a look at the DirectX documentation.
CHAPTER 16

Particle Systems
Particle systems are one of those things that you don’t really need but probably want to include anyway, because they’re really neat. They can be used for a number of interesting effects, including explosions, sparks, fire, streams of water, rain, and snow. They can be done using either 2D math or 3D math, which makes them ultimately adaptable.

So, let’s get started. Your goals for this chapter are to:

- Understand what particle systems are
- Understand issues for dealing with particle systems
- Know how to model a simple particle that uses Newtonian physics
- Know how to store a particle system in a container
- Know how the particle system’s life cycle works
- Know how to render a particle list

What Are Particle Systems?

A particle system is simply a collection of very small objects called particles. I’m sure you had that part figured out already. A particle can be just about anything from a single point to a small line to a single polygon to even a small 3D model. In the game Asteroids, the asteroids themselves essentially make up a particle system. The particles are simply very large.

Particle systems have rules. There is typically a particle generator, the source where all particles in the system come from, some sort of application of physics while the particle is in existence, and a time after which the particle no longer exists, which is the “death” rule for the particular particle.
So, the basic life cycle of a particle is to be born (at the generator), to move according to the physics rules, and finally to die (usually after a certain amount of moving). Of course, there is quite a bit more to it than that, or else there wouldn’t be so many resources that discuss particle systems and the techniques for doing them.

Particle System Issues

One of the main issues concerning particle systems is that of storage in memory. Because a system can contain anywhere between zero and an infinite number of particles (in theory, at least), you need a variably-sized container to store them. This immediately brings up the issue of speed. The container you use has to insert and delete items quickly, and it should be reasonably fast in order to loop through all of the particles when it comes time to draw them. My personal container of preference in these cases is a linked list.

Because this chapter uses linked lists, all of the items in the list are dynamically allocated. This brings up another speed issue. Allocating memory takes time. Deallocating memory also takes time. If you suddenly need to spawn 1000 particles, you are going to take a performance hit. So you need a way to minimize the number of memory allocations and deallocations. You can do this by allocating all of your particles beforehand, and placing them into another linked list that you use as sort of a “pool” of particles that are not currently in use. You can then simply take pointers to your particles out of this pool and place them into the particle list. When particles die, you remove them from the list and place them back into the pool. If the pool is out of particles at a particular moment, you have a choice: You can either allocate a new particle on the fly right then and there, or you can simply disallow the creation of new particles, keeping in mind that allocating on the fly will incur some performance hits.

A Particle Type

For the purposes of this discussion, I have created a structure for containing all of the information pertinent to a particle. The information used to store a particle will vary from system to system, just as needs
vary, but generally speaking you will have a basic particle with most of
the following information:

```c
struct PARTICLE
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Velocity;
    D3DXVECTOR3 Acceleration;
    float Drag;
    DWORD Color;
    DWORD TimeLeft;
};
```

This structure is pretty much the minimum for a particle. Other
things you might add are mass and spin (if you are using an object
larger than a single pixel for a particle).

In my particle type, I've got all of the basic necessities for implement-
ing Newtonian physics: Position, Velocity, Acceleration, and Drag. I've
also got Color, which is useful for making differently-colored particles,
and one member called TimeLeft. Each time a particle is shown,
TimeLeft decreases by one. When it reaches 0, the particle will die. It's
a simple way to manage the life cycle of a particle.

Here is your basic Newtonian physics review, ultra-simplified: You have
Position, Velocity, Acceleration, and Drag. Position indicates where in
world space (although you could also use model space if you so
desired) you are. Velocity determines which way the particle is going
per unit time (the unit of time in this case is one frame). In each
frame, you add Velocity to the Position to determine the new posi-
tion. Acceleration is the change in velocity per unit time, and so in
every frame, Acceleration is added to Velocity to make the new veloc-
ity (this is done prior to adding Velocity to Position).

Finally, Drag simulates air friction. It ranges from 0.0 to 1.0. Before you
add Velocity to Acceleration, you multiply Velocity by Drag. If Drag is
1.0, Velocity doesn’t change. If Drag is 0.0, Velocity is also set to zero,
making Acceleration essentially the Velocity of the particle. A Drag
anywhere between 0.0 and 1.0 gives the particle a “terminal veloc-
ity”—a maximum velocity based on the acceleration and the drag
coefficient.
Here’s a basic breakdown of how the physics system works per frame.

1. Multiply Velocity by Drag, and store the new value in Velocity.
2. Add Acceleration to Velocity, and store the new value in Velocity.
3. Add Velocity to Position, and store the new value in Position.

Pretty simple, right? Perhaps you would rather look at equations. I know I would.

// P, V, A, and D are Position, Velocity, Acceleration, and Drag
V *= D;  // New Velocity = Old Velocity times Drag
V += A;  // New Velocity = Old Velocity plus Acceleration
P += V;  //New Position = Old Position plus Velocity

With Drag other than 1.0, you’ll wind up with a maximum velocity for your particles. You might intentionally want to do that, to keep your particles from flying away too far. The terminal velocity is based on Acceleration and Drag.

V1 = ( V0 * D ) + A ; // combine drag and acceleration equations

In this case, you want V1 and V0 to be equal (no change in velocity), so:

V0 = ( V0 * D ) + A ;
V0 - V0 * D = A ;
V0 * ( 1 - D ) = A ;
V0 = A / ( 1 - D ) ;

So, if you have an Acceleration of (1, 1, 1), and a drag of .9, you get:

V0 = (1,1,1) /(1-0.9) = (1,1,1)/(0.1) = (10,10,10)

**NOTE**

This equation works for values of Drag between 0.0 and 1.0, inclusively. A value of 1.0 will give you division by zero (an infinite terminal velocity), and a value of 0.0 will give you the same value as Acceleration. It also works for equations outside of the range 0.0 to 1.0, but doing that will give you some strange results, especially some of the negative values. Consider yourself warned.
Storing a Particle System

You’ll want to have some sort of container to store all of your particles. Actually, you will want three containers—one for live particles, one for dead particles, and one for dying particles—so that you can minimize the number of memory allocations at runtime.

For this I like to use the STL list template, because it is variably-sized and pretty fast when adding and removing entries. Another candidate is the vector template, but it has a slower insertion time and so is unsuitable for these purposes.

Declaring a list of particles is just as easy as declaring a vector, which you looked at last chapter.

```cpp
list < PARTICLE* > lstParticlesAlive ;
list < PARTICLE* > lstParticlesDying ;
list < PARTICLE* > lstParticlesDead ;
```

Adding a particle to the list is no problem; you simply use the `push_back` function, just like you did with vector.

```cpp
//add a particle to the list
lstParticlesAlive.push_back ( pParticle ) ;
```

Removing a particle from the list is also rather easy. You simply use the `remove` member function.

```cpp
//remove particle from the list
lstParticlesAlive.remove ( pParticle ) ;
```

The only sticky part with using lists is accessing a particular particle. The list doesn’t have the handy operator [] that does this for you, and getting random access to the member of a list is somewhat painful and slow. This is the trade-off between vectors and lists. Vectors have good random access, but slow insertion. Lists have poor random access, but fast insertion.

Luckily, you don’t much care about being able to randomly access these particles. If you can just access the list from beginning to end all will be just fine. And you can do that with an iterator, as follows:

```cpp
list < PARTICLE* >::iterator iter ;
PARTICLE* pParticle ;
for ( iter = lstParticlesAlive.begin ( ) ; iter != lstParticlesAlive.end
```
To determine how many items are in a list, use the size member function. Just don’t use it often, because it is slow.

```cpp
int nItemCount = lstParticlesAlive.size();
```

Finally, to determine whether a list is empty, you use the empty member function, like so:

```cpp
if (lstParticlesAlive.empty())
{
    //list is empty
}
```

And that’s all you really need to be able to work with the STL list template. Sure, there are a bunch of other functions, but you don’t need them and why draw out the discussion? There are plenty of references on STL if you want to know more; *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, is a good one.

So, why three lists? Wouldn’t just two be enough? Yes, it might be; however, using two lists makes the main loop that looks at live particles a little more complicated, and I wanted it to look as simple as possible.

When you are looping through the “alive” list, removing a particle during the loop requires that the iterator not move forward when at the point of that particle, otherwise particles would be skipped. By simply adding the particle to the “dying” list, you don’t need any special code for this; you simply add it to that list. After the “alive” loop has gone all the way through, you loop through the “dying” list, remove those particles from the alive and dying lists, and move them over to the dead list.

**A Particle’s Life Cycle**

Particles come to life, move about, and die. It’s a rather pointless existence, really. Of course, I’ve met human beings with even less purpose
to their existence, so maybe being a particle wouldn’t be so bad! As for the particle system, you start with all of the particles in the “dead” list. You allocate however many particles you plan on using for the system, say 10,000, and perform all of the allocations at one time, like so:

```cpp
int counter;
PARTICLE* pParticle;
for (counter = 0; counter < PARTICLECOUNT; counter++)
{
    pParticle = new PARTICLE;
    lstParticlesDead.push_back(pParticle);
}
```

PARTICLECOUNT is whatever constant you’re using to control the size of your particle list. You want a nice big number—big enough that you don’t often run out of particles, but not so big that you have to increase your program’s memory requirements.

On cleanup, you have to take care of all the particles currently in the alive and dead lists (but not the dying list, because that is just a transient location). For this, you simply loop through each of those lists and delete each of the particles.

```cpp
list<PARTICLE*>*::iterator iter;
PARTICLE* pParticle;
//loop through alive list
for (iter = lstParticlesAlive.begin(); iter != lstParticlesAlive.end(); iter++)
{
    pParticle = *iter;
    delete pParticle;
}
for (iter = lstParticlesDead.begin(); iter != lstParticlesDead.end(); iter++)
{
    pParticle = *iter;
    delete pParticle;
}
```
And lastly, you clean out all of the lists with the `clear` member function, like so:

```cpp
lstParticlesAlive.clear();
lstParticlesDead.clear();
lstParticlesDying.clear();
```

And then you’re all cleaned up.

Once you have the initialization and cleanup code for the particle system as a whole put together, you can start giving birth to particles and letting them live out their (pointless) lives.

## Birth of a Particle

Now you’ve got a choice to make. The choice is about what happens when there are no particles in the dead list to bring to life. Do you allocate a new particle and add it to the list? Or do you not add particles, which means the alive list is just out of luck when there are no particles available? This is all up to you, and each choice comes with its drawbacks and advantages. If you allocate new particles when the dead list is out, all of your effects with particle systems will look the same—even if you run out of particles—but at the same time, you’re allocating memory at runtime, which can have a detrimental effect on performance. Also, you’re eating up more memory for the particle system.

If you limit the particle to a strict number, some of your effects using particles won’t look as good when you run low, but at the same time you won’t be allocating the extra memory. So, it’s another trade-off. Often you will have to experiment with the scheme as well as the initial number of particles in the list to begin with in order to find the optimal solution.

And so, when a particle is born, you first determine whether the dead particle list has any particles in it. If it does, you grab the first one, set it up, and add it to the alive list. Optionally, if a particle is not available on the dead list, you can allocate a new particle right then and there, set that particle up, and add it to the alive list.

```cpp
list< PARTICLE* >::iterator iter ;
PARTICLE* pParticle ;
```
//check for a particle in the dead list
if ( !lstParticlesDead.empty() )
{
    //a particle exists in the dead list
    //grab the first one
    iter = lstParticlesDead.begin();
    pParticle = *iter;
    //set up the particle here*
    //remove particle from dead list
    lstParticlesDead.remove(pParticle);
    //add particle to alive list
    lstParticlesAlive.push_back(pParticle);
}
else
{
    //no particles exist in the dead list
    //allocate new particle
    pParticle = new PARTICLE;
    //set up new particle here*
    //add particle to alive list
    lstParticlesAlive.push_back(pParticle);
}

And, as stated earlier, you might want to eliminate the else clause from this if statement, because memory allocations are expensive. Then again, you might not care. If you do have this “allocate on demand” type of thing you can forego the initial allocation of particles on the dead list, because the system will take care of any allocations on its own.

**Live Particles**

Once a particle has been born, it is moved to the alive list. While on the alive list, it is updated constantly, typically once per frame. The acceleration, velocity, and position are all updated, as is the TimeLeft member, which is decreased by one each time it is updated. If TimeLeft becomes zero, it is moved to the dying list, which is much like a nursing home for elderly particles.
So, to update all of the alive particles, you do the following:

```cpp
//make sure particle list is not empty
if ( !lstParticlesAlive.empty() )
{
    list< PARTICLE* >::iterator iter;
    PARTICLE* pParticle;
    //loop through the particles
    for ( iter = lstParticlesAlive.begin(); iter != lstParticlesAlive.end(); iter++ )
    {
        //apply drag to velocity
        pParticle->Velocity *= pParticle->Drag;
        //add acceleration to velocity
        pParticle->Velocity += pParticle->Acceleration;
        //add velocity to position
        pParticle->Position += pParticle->Velocity;
        //decrease the time left for this particle
        pParticle->TimeLeft --;
        //check for death
        if ( pParticle->TimeLeft == 0 )
        {
            //add this particle to the dying list
            lstParticlesDying.push_back(pParticle);
        }
    }
}
```

Notice that particles that have just died are still on the alive list. Depending on your design, you might not want to render these, and so that will affect the timing of your dead handler.

**Death of a Particle**

During the processing of live particles, if `TimeLeft` reaches zero, the particle is added to the dying list. At some point after that, you need to handle any dead particles that might have occurred. This is done simply by looping through the dying list, removing the particles on
that list from the alive list, placing those particles on the dead list, and finally cleaning out the dying list when you are all through.

```cpp
// if the dying list has particles in it
if ( !lstParticlesDying.empty() )
{
    list< PARTICLE* >::iterator iter ;
    PARTICLE* pParticle ;
    // loop through all the particles on the list
    for ( iter = lstParticlesDying.begin() ; iter != lstParticlesDying.end() ; iter ++ )
    {
        pParticle = *iter ;
        // remove the particle from the alive list
        lstParticlesAlive.remove( pParticle ) ;
        // add the particle to the dead list
        lstParticlesDead.push_back( pParticle ) ;
    }
    // clear out the dying list
    lstParticlesDying.clear() ;
}
```

So, it’s not like death at all, but rather reincarnation (won’t the particles be happy!).

**Rendering the Particles**

Up until now, this discussion of particles has been purely design based. You still don’t have a method of rendering the particles to the screen. Without one, well, there’s just no *point.* Get it?

First, you’re going to create a vertex structure for your particles. There isn’t much there, really, just a position and a color, so you’ll make the custom vertex like so:

```cpp
#define PARTICLE_VERTEX_FVF D3DFVF_XYZ | D3DFVF_DIFFUSE
struct ParticleVertex
{
    D3DVECTOR position ;
    DWORD color ;
} ;
```
This code assumes that you are using Direct3D for geometry transformation, of course. If you aren’t, you can simply add the RHW to it; don’t forget to set it to 1.0 when you’re ready to render.

Now you have each particle in its own little structure, but setting up a single particle vertex would be rather inefficient, so you should set them up in batches of 100 or so. The Position member of PARTICLE corresponds directly to the position member of ParticleVertex, and the Color member corresponds to color, so making copies is not a problem.

Set up an array somewhere.

```cpp
//set the number of vertices for rendering
const int PARTICLEVERTEXCOUNT = 100;
//set up an array for particle vertices
ParticleVertex g_vertParticles[PARTICLEVERTEXCOUNT];
```

Now, when rendering, start filling in information at `g_vertParticles[0]`. When you reach the end of the array, render it and then start over. After you have looped through all the other particles, render whatever is left (if anything).

```cpp
//if the alive particle list is not empty...
if (!lstParticlesAlive.empty())
{
    //set vertex shader for particle vertices
    g_pd3ddev->SetVertexShader(PARTICLE_VERTEX_FVF);
    list<PARTICLE*>::iterator iter;
    PARTICLE* pParticle;
    int nVertex = 0;
    //loop through particles
    for (iter = lstParticlesAlive.begin(); iter != lstParticlesAlive.end(); iter++)
    {
        pParticle = *iter;
        //add particle to vertex list
        g_vertParticles[nVertex].position = pParticle->Position;
        g_vertParticles[nVertex].color = pParticle->Color;
        //increment vertex number
        nVertex++;
        //if we have filled up the array
```
if ( nVertex == PARTICLEVERTEXCOUNT )
{
    //draw all of the particles in the array
    g_pd3ddev->DrawPrimitiveUP ( D3DPT_POINTLIST ,
        PARTICLEVERTEXCOUNT ,
        g_vertParticles ,
        sizeof ( ParticleVertex ) ) ;
    //set vertex index back to 0
    nVertex = 0 ;
}
}

//render remaining particles, if any
if ( nVertex != 0 )
{
    //draw remaining particles
    g_pd3ddev->DrawPrimitiveUP ( D3DPT_POINTLIST ,
        nVertex ,
        g_vertParticles ,
        sizeof ( ParticleVertex ) ) ;
}
}

Note that updating the particle list is separate from rendering the particle list. This is primarily for instructional purposes, because it is easier to see how something is working if there is only one thing working at a time. However, you probably don’t want to loop through the particle list more than once per frame, because when the particle list is large doing so can take some time. In such a case you might want to combine both the update and rendering loops into a single loop; doing this makes it a little harder to read, of course, but quite a bit more efficient.

//make sure particle list is not empty
if ( !lstParticlesAlive.empty ( ) )
{
    //set vertex shader for particle vertices
    g_pd3ddev->SetVertexShader ( PARTICLE_VERTEX_FVF ) ;
    int nVertex = 0 ;
    list < PARTICLE* >::iterator iter ;
    PARTICLE* pParticle ;
/loop through the particles
for ( iter = lstParticlesAlive.begin ( ) ; iter != lstParticlesAlive.end ( ) ; iter ++ )
{
    //apply drag to velocity
    pParticle->Velocity *= pParticle->Drag ;
    //add acceleration to velocity
    pParticle->Velocity += pParticle->Acceleration ;
    //add velocity to position
    pParticle->Position += pParticle->Velocity ;
    //decrease the time left for this particle
    pParticle->TimeLeft -- ;
    //check for death
    if ( pParticle->TimeLeft == 0 )
    {
        //add this particle to the dying list
        lstParticlesDying.push_back ( pParticle ) ;
    }
    //add particle to vertex list
    g_vertParticles [ nVertex ].position = pParticle->Position ;
    g_vertParticles [ nVertex ].color = pParticle->Color ;
    //increment vertex number
    nVertex ++ ;
    //if we have filled up the array
    if ( nVertex == PARTICLEVERTEXCOUNT )
    {
        //draw all of the particles in the array
        g_pd3ddev->DrawPrimitiveUP ( D3DPT_POINTLIST ,
            PARTICLEVERTEXCOUNT ,
            g_vertParticles ,
            sizeof ( ParticleVertex ) ) ;
        //set vertex index back to 0
        nVertex = 0 ;
    }
}
//render remaining particles, if any
if ( nVertex != 0 )
{
    //draw remaining particles (nVertex == the number remaining)
A Particle System Example

There’s just one more thing to do, and that’s try it. Example 16.1 (X:\Source\Example16_1) on the CD uses almost the exact code found in this chapter. It generates particles along the top of the screen and has them fall towards the bottom of the screen, making a simulation of rain. I think it’s pretty neat.

Of course, you can adapt this program to make just about any sort of particle effect you can imagine: rain, snow, explosions, sparks, fire, smoke, and so on. Have fun with it.

Summary

In conclusion, particle systems are neat! You knew that already. Also, they aren’t too hard to put together, provided you think about what you want to do beforehand. They are fairly simple to implement and can produce outstanding effects. Still, particles are very much an effect that you have to experiment with—there isn’t a cookbook of particle effects that you can draw from. This chapter looked at the basics, but, with a little work, you’ll find they can do a lot more than make little blue dots that simulate rain.
CHAPTER 17

Point in a Poly
There’s one more item I think you probably need to make your 2D in Direct3D repertoire complete. Throughout the book you’ve been learning how to draw objects, but you’ve spent no time selecting objects. Although it’s a huge topic, I think I can still shed some light by discussing how to determine whether a point is in a polygon.

If you’re using transformed and lit coordinates with rectangular quads, you don’t really need this chapter; simply determine whether the position is within the bounding rectangle of the object, and if it is, determine whether it is on the object by looking at a two-dimensional Boolean array or something similar. That’s just a standard 2D hit selection.

For this chapter, you have only one objective: Know how to determine whether a 2D point is in a 2D triangle

**Point in a Triangle**

By polygon, I strictly mean “triangle,” because that’s all you have been dealing with in the book. Other polygons can always be broken down into component triangles.

There is a nice, simple way to determine whether a point is within a triangle. The trick is as follows: from the point you’re testing, extend a ray in any direction. If this ray intersects the triangle an odd number of times, it is inside the triangle. If the ray intersects the triangle an even number of times, it’s outside of the triangle. Don’t believe me? See Figure 17.1.

This works for 2D, 3D, and so forth. For simplicity, though, this chapter examines only 2D, using the screen coordinates, but moving the method into 3D isn’t that hard, really.

The easiest ray you can use for something like this is a horizontal ray—there is the least amount of math involved. For a point, for example (4.0, 3.0), extending a ray outwards is as simple as this:

\[
\begin{align*}
Y &= 3.0 \\
X &\geq 4.0
\end{align*}
\]
There’s no slope, no rise over run, and practically nothing to calculate. For the sides of the triangle it gets a little more complicated, but not too much so. The equation for a line based on two points is as follows:

\[(x_2 - x_1) \times (y_2 - y_1) = (x - x_1) \times (y_2 - y_1)\]

This is modified from the standard point-slope configuration, but it’s a little backwards for these purposes. The coordinates of one end of the line are \((x_1, y_1)\) and \((x_2, y_2)\) are the coordinates of the other end. I know that in reality that the lines go on forever, but you are only concerned about the portion of the line in between these two end points.

There are essentially three cases of this equation that you have to look for. The first is when \(x_1\) and \(x_2\) are equal (making the line vertical), the second is when \(y_1\) and \(y_2\) are equal (making it horizontal) and the third is when neither of these is true.

In the first case, the vertical line, the equation winds up just being:

\[X = x_1 \text{ OR } X=x_2 \text{ (}x_1 \text{ and } x_2 \text{ have the same value anyway).}\]

Testing for an intersection is really easy in this case. You must simply make sure that \(x_1\) is greater than the \(x\) component of the point you are testing, and you must make sure that the \(y\) of the point you are testing is between \(y_1\) and \(y_2\).

//vertical line hit test

**Figure 17.1** *Even/odd crossings allow you to determine whether a point is within a triangle.*
if ( x < x1 && y > y1 && y < y2 )
{
    // a hit!
}

In the second case, a horizontal line, you don’t really have to do anything; the line itself is not strictly inside the polygon. The end point test, a little later on, takes care of this test for you.

In the third case it gets slightly more complicated, but not very much so, because you are using a horizontal ray. You simply take the linear equation and solve for x, like so:

\[
(x_2 - x_1) * (y - y_1) = (y_2 - y_1) * (x - x_1)
\]

\[
(x_2 - x_1) * (y - y_1) / (y_2 - y_1) = (x - x_1)
\]

\[
(x_2 - x_1) * (y - y_1) / (y_2 - y_1) + x_1 = x
\]

Into the left side of the equation, you plug in your point’s y value and get the x value of the line at that point. If this x value is between x1 and x2 and is greater than your point’s x value, it’s a hit.

// non-horizontal or vertical line
float line_x = (x_2 - x_1) * (y - y_1) / (y_2 - y_1) + x_1;
if (line_x > x_1 && line_x < x_2 && line_x > x)
{
    // a hit
}

You’re still not quite done yet—you still have to test the end points. For each end point, simply check that the y values are the same as the point you are testing, and that the x value of the end point is greater than the point’s x value. Using this method, you test each of the three lines, and if you get 0 hits or 2 hits, the point is not in the triangle. If you get one hit, it is inside the triangle. Because of the nature of triangles, you won’t get three hits (provided all of the points of the triangle are different).

**Summary**

While it takes a little math, determining whether or not a point is in a given polygon is no Herculean task. With a little bit of linear algebra, it’s actually quite easy.
CHAPTER 18

Wrap up
You’ve reached the end, so look to the future and think about the past, all while drinking milk and eating cookies. This has been a relatively short book, but I hope by this point you have a much better idea about how to use Direct3D, even if you’re only using it for two-dimensional graphics.

Where’s the Big Game?

Typically, in a book like this, the last chapter contains “the big game example.” I thought about it, but then I realized something: This book is not about you watching me make games. This book is about giving you the tools and tips you need to go out and make your own games. All of the programming examples in this book are intentionally rather simplistic. The idea is that if you can perform one of the many tasks done by Direct3D in a simple manner and understand it, you can then do it in a more complicated manner and still understand it. The main purpose of the text is to give you the understanding necessary to use the API.

There are some topics the book didn’t get a chance to cover. Direct3D is a rich API with a lot of really interesting features, like point sprites, vertex shaders, and pixel shaders. These topics, however, aren’t really fundamentals, which is more what these chapters have been concerned with.

Where You’ve Been

The overlying mission of the book was to take you from a 2D API and bring you into a 3D world—which you discovered was still a 2D world with some clever tricks applied to make people think it’s 3D.

What you’ve read here was a leg-up for you if you’ve been wanting to move towards a 3D API but always thought it was too difficult to learn.
It’s not all that difficult, if you take it bit by bit. However, if you have purchased other books about Direct3D, you’ll notice that they are always from the point of view of a 3D programmer, and from experience I know that this technique of writing will often leave a 2D programmer behind. I know it did when I was trying to learn all of this stuff.

At this point you should be able to pick up that Direct3D book that perplexed you in the past, read it, and have a much easier time understanding it. The fundamentals don’t change, just how you use them.

Where You’re Going

To make games, of course. To practice your skills as game programmers. To make that really cool game that sells a million and a half copies, which means you can buy the big mansion.

Well, okay—maybe that’s not realistic. I think the best we can hope for is to continue in the quest to become better programmers. At the pace of technology today, each day brings something new to learn, some new platform to work on, some new API to demystify. Often it might seem you can never get ahead in this race, and you’re always playing a game of catch-up.

Of course, the game of catch-up is part of the fun of being a programmer. It is an exciting time to be programming. I personally could not conceive of doing anything else. It’s fun and challenging, a puzzle-solving experience where the puzzle changes every day so that you never get bored.

Summary

Thanks for reading my book. I hope you’ve learned as much as I have (yes, authors also learn through the process of writing). Keep programming! If you run into any problems, you can reach me at tanstaaf@gamedev.net.
This page intentionally left blank
Index

A

absolute values, vectors, 133
acceleration, particle systems, 240–241
Adapter parameter, 9
adapters, video cards, 8–11
addition
  matrices, 144–146
  vectors, 131
address modes, textures, 99–102
AddXMaterialList function, 231
AddXVertexColorList function, 233
alpha blending
  colors, 196–197
  defined, 196–197
  hardware capability checking, 200
  inverted values, 198
  render states, 198–200
  setup, 198–200
  source/destination blend example, 201–202
  uses, 201–202
  z buffering concerns, 198
alpha testing
  cell divisions, 114–115
  color bits, 109
  color-control values, 113–114
  defined, 108
  enabling, 108
  image loading, 110–112
  render states, 108–109
ambient lighting
  defined, 177
  render states, 189
  setup, 189–192
Ambient member, 180–181
applications
  back buffer, 26–27
  Direct3D initialization, 5–6
  DirectDraw initialization, 5
  display reset, 32–35
  full-screen, 25
  top-level window, 25
arrays
  versus vertex buffers, 46
  X file elements, 214
atan function, 134
atan2 function, 134
attenuation, lighting, 182–183
Attenuation0 constant, 182–183
Attenuation1 constant, 182–183
Attenuation2 constant, 182–183

B

back buffers
  access methods, 76
  clearing z buffers, 173
  copying surface to, 79–80
  copying to the screen, 27–28
  described, 26
  surface access, 76–77
  swap chains, 76
BeginScene function, 59
behavior flag, 25
BehaviorFlags parameter, 25
bEnabled parameter, 185
bit flags, surface images, 84
bit shifting, colors, 81–82
bitmaps, image loading, 81–86
blending weights, positioning, 42
brackets, [ and ], particle lists, 242
C

candle flames, point light type, 181
capabilities, D3DCAPS8 members, 13
Cartesian coordinates, vector rotation, 135–136
CheckDepthStencilMatch function, 168–169
CheckDeviceFormat function, 19–21, 168
CheckDeviceType function, 21–22, 95
Clear function, 30–32, 173, 245
ClientToScreen function, 88
color argument constant, 106
Color member, 249
color-control values, alpha testing, 113–114
COLORREF variable, 81
ColorRefTo function, 82
colors
  alpha blending, 196–197
  alpha component, 109
  argument constants, 106
  bit shifting, 81–82
  diffuse, 43
  Direct3D descriptions, 30–31
  specular, 43
COM object, described, 7
containers, particle systems, 242–243
ConvertFromColorRef function, 82–83
CopyRects function, 79–80
Count parameter, 31
CreateDepthStencilSurface function, 170
CreateDevice function, 24–25, 76
CreateImageSurface function, 77–78
CreateIndexBuffer function, 51–52
CreateRenderTarget function, 90
CreateTexture function, 94
CreateVertexBuffer function, 46–47
cube space, described, 123–124
curly braces { and }, header templates, 211
CursorCaps member, 87

cursors
  eliminating Windows cursor, 88
  hardware support considerations, 87
  hotspots, 87
  image surface dimensions, 87
  making visible, 88
  positioning, 88
  update frequency, 88
CUSTOM_VERTEX_FVF macro, 40–41, 45
CustomVertex structure, 130, 176–177

d
D3D_ENUM_NO_WHOL_LEVEL constant, 9
D3D_OK (success) return, 9, 12
D3D_SDK_VERSION constant, 7
D3DADAPTER_DEFAULT, default adapter designation, 9
D3DADAPTER_IDENTIFIER8 member, 10
D3DBACKBUFFER_TYPE_MONO value, 76
D3DBLEND constant, 199
D3DBLENDOP constant, 200
D3DCAPS2_CANRENDERWINDOWED flag, 26
D3DCAPS8 member, 13
D3DCREATE_MIXED_VERTEXPROCESSING flag, 26
D3DCREATE_SOFTWARE_VERTEXPROCESSING flag, 26
D3DCREATE_VERTEXPROCESSING flag, 26
D3DCURSOR_DEFAULT flag, 88
D3DCURSORCAPS_COLOR flag, 87
D3DCURSORCAPS_LOWRES flag, 87
D3DDEVTYPE enumerated types, 12–14
D3DDISPLAYMODE member, 17–18
D3DERR_INVALIDCALL return, 9, 12
D3DERR_INVALIDDEVICE return, 12
D3DERR_OUTOFVIDEOMEMORY return, 12
D3DFMT_INDEX16 constant, 52–53
D3DFMT_INDEX32 constant, 52–53
D3DFMT_R5G6B5 constant, 81–82
D3DFORMAT enumerated types, 14–16
data loading, vertex buffers, 48–49
data representation, vertex formats, 39–40
DEDCOLOR variable, 30
depth buffer, 169–170
depth formats, z buffers, 169–170
depth/stencil surfaces, 90–91
diffuse color, vertex formats, 43
diffuse lighting, defined, 177
Diffuse member, 177–181
Direct3D
API purpose, 38–39
color descriptions, 30–31
cursors, 86–89
flexible vertex format (FVF), 39–40
game uses, 259
geometry transformation purpose, 39
initialization, 5–6
migrating from DirectX 7, 4
mission statement, 258–259
polygon rendering, 39
purposes, 38–39
resource types, 75
state-based rendering machine, 56–58
thinking outside the box, 32
Direct3DCreate8 function, 6–7
DirectDraw
API purpose, 38
Blt function, 79
display adapter determination, 8
IDirectDrawSurface7 object, 24
initialization, 5
DirectDrawCreateEx function, 6
Direction member, 181–182
direction, lighting, 181–182
directional light, direction only, 181
directional lighting, setup, 187
DirectX 7, migrating to Direct3D, 4
dirty rectangles, updating, 28
display adapters
determining available, 8–11
Direct3D 8 in window support checking, 26
enumerating modes, 16–18
hardware acceleration support checking, 21–22
resetting, 32–35
resource format checking, 19–21
returning current display mode, 19
display formats, D3DFORMAT enumerated types, 14–16
display mode
enumerating, 16–18
returning current, 19
dot product
vector scaling, 132–133
vectors, 136–137
drag, particle systems, 240–241
DrawIndexedPrimitive function, 71, 158
DrawIndexedPrimitiveUP function, 6–70, 158
drawing, primitives, 56–77
DrawPrimitive function, 70, 158
DrawPrimitiveUP function, 62–63, 158
dwIndex parameter, 185

E
e-mail addresses, author’s, 259
emissive lighting
defined, 177
setup, 193–194
Empty function, 243
EndScene function, 59
EnumAdapterModes function, 17
enumerated types
defined, 8
examination
determining available video adapters, 8–11
device capability, 6, 8–14
device types, 11–14
hardware abstraction level (HAL) device, 11
pure software device, 12
reference rasterizer device, 11–12

G
GDI, BitBlt function, 79
gometry pipeline
described, 158–161
model, 158
multiplying transformation matrices, 163–164
setting into motion, 158
transformation setup, 161–163
gometry transformation
coordinate spaces, 124–125
Direct3D purpose, 39
GetAdapterCount function, 9
GetAdapterDisplayMode function, 19
GetAdapterIdentifier function, 9
GetAdapterModeCount function, 16
GetAdapterMonitor function, 9
GetBackBuffer function, 76–77
GetBValue macro, 81
GetDepthStencilSurface function, 172
GetDeviceCaps function, 12
GetGValue macro, 81
GetLight function, 184
GetLightEnable function, 185
GetPixel function, 81
GetRenderState function, 58
GetRenderTarget function, 91
GetRValue macro, 81
GetSurfaceLevel function, 96
GetTransform function, 161
GetViewport function, 30

H
hardware abstraction level (HAL), device type, 11
hardware acceleration, support checking, 21–22
header template, X files, 211
Height parameter, 90
hotspots, cursors, 87
HRESULT return, 9, 12
IDirect3DDevice8 object
  behavior flags, 25
  creating, 24–28
  releasing, 28
  swap chains, 76
if statements, z buffers, 167
image surfaces, described, 77–78
index buffers
  creating, 51–52
  described, 51
  destroying, 53
  device notification, 53
  drawing indexed primitives with, 71
  loading, 52–53
  locking/unlocking, 52–53
  size settings, 52
indexed primitives
  drawing w/index buffers, 71
  user pointers, 67–69
initialization
  Direct3D, 5–6
  DirectDraw, 5
intensity, lighting, 182–183
intersections, triangle, 254–256
inverse trigonometric functions, vector direction, 134
inverted values, alpha blending, 198
iterators, particle systems, 242–243
Length parameter, 52
Levels parameter, 95
life cycle, particle systems, 243–248
light bulbs, point light type, 181
LightEnable function, 184–185
lighting
  ambient, 177, 189–192
  attenuation, 182–183
  diffuse, 177
direction, 181–182
directional, 181–182, 187
emissive, 177, 193–194
floats, 178
index of, 185
intensity, 182–183
normal vector, 176–177
point, 181–182
point lights, 181, 185–187
position, 181–182
range, 182
setup, 184–185
specular, 177, 192–193
spotlights, 181–184, 187–189
type sources, 180–181
line strips, primitive type, 60–61
linear algebra, texture mapping, 96–97
lines, primitive type, 59–62
lists
  face, 226–228
  material, 228–231
  particle declarations, 242–243
  vertex, 224–225
LoadBitmap function, 81
LoadIconTexture function, 111
LoadImage function, 81
LoadXFace function, 226–227
LoadXFile function, 224, 229, 233–236
LoadXMaterialList function, 230–231
LoadXMesh function, 225, 227–228
LoadXVertexColor function, 232
LoadXVertexColorList function, 232–233
Lock function, 52, 48
Lockable parameter, 90
LockRect function, 83
magnitude, vectors, 133
Material definition template, X files, 211–212
matrices
- addition, 144–146
- described, 142–144
- multiplication, 146–149
- rotation, 150
- scaling, 152
- translation, 149–150
matrix, defined, 142
MaxTextureHeight member, 95
MaxTextureWidth member, 95
mesh
- adding information, 230
- face list, 226–228
- MeshMaterialLists template, 228–229
- storing, 223–234
- templates, 228–234
- vertex list, 224–225
Mesh template, X files, 212–214
MeshMaterialList template
- X files, 214–215
- mesh, 228–229
model space, described, 120–121
models
- defined, 158
- grouping types, 234–236
- mesh token, 223–234
- tokenizing text, 219–223
- tokens, 219–234
- vector template, 216–218
- X files, 207–215
modes, display adapters, 16–18
modulation, defined, 103
multiplication
- matrices, 146–149
- transformation matrices, 163–164
- vector scaling, 132–133
MultiplyTransform function, 163
MultiSample parameter, 90
multi-texturing
- color argument constants, 106
- described, 102–103
- modulation, 103
mutually exclusive flag, 26

N
- non-square textures, hardware support issues, 95
- normal vector, lighting, 176–177
- normal, vertex coordinate, 42
- normalizing, vectors, 134–135

O
- objects
  - COM, 7
  - IDirect3D8, 5–8
  - IDirect3DDevice8, 5–6, 24–28, 76
  - IDirectDraw7, 5–6
- OffsetToLock parameter, 49
operators
- D3DXMATRIX class, 154
- D3DXVECTOR3 class, 139–140
- or (|), 42

P
- parameters
  - Adapter, 9
  - BehaviorFlags, 25
  - bEnabled, 185
  - Count, 31
  - D3DPOOL, 48
  - dwIndex, 185
  - Flag, 9, 49
  - Flexible Vertex Format, 48
  - Format, 52, 90
  - Height, 90
  - Length, 52
  - Levels, 95
  - Lockable, 90
  - MultiSample, 90
  - OffsetToLock, 49
  - pBits, 84
  - Pool, 52
particle, defined, 238

particle generator, described, 238

PARTICLE structure, 240–241

particle systems
adding to dying list, 247–248
alive list updating, 246–247
allocating new, 245–246
clearing lists, 245
combining update/rendering loops, 250–252
dead list allocations, 244
deleting particles, 244
described, 238–239
information storage, 239–241
issues, 239
iterators, 242–243
life cycle, 243–248
list declarations, 242–243
PARTICLE structure, 240–241
PARTICLECOUNT, 244
rendering, 248–252
storage containers, 242–243

vertex structure, 248–249

PARTICLECOUNT structure, 244

pBits parameter, 84

Phi member, 183

Pitch bytes, calculating, 84

Pitch member, 84

pixel formats, D3DFORMAT enumerated types, 14–16

pixels, plotting, 84–85

point lights
position only, 181
setup, 185–187

pointers
D3DLOCKED_RECT, 83–84
non-NULL, 32
pCursorBitmap, 87
pIdentifier, 9
pNewZStencil, 90
ppReturnedDeviceInterface, 28
user, 62–63
XVERTEX, 224

points
primitive type, 59–62
triangles, 254–256

polygon rendering, Direct3D purpose, 39

polygons, points in a triangle, 254–256

Pool parameter, 52

position
lighting, 181–182
particle systems, 240–241
vertex information, 40–41

Position member, 181–182, 249

Power member, 212

ppbData parameter, 49

pPresentationParameters parameter, 26

ppSurface parameter, 90

ppSurfaceLevel parameter, 96

ppTexture parameter, 95

pRects parameter, 31

pRenderTarget parameter, 90

Present function, 31–32
primitives  
applying textures, 97–99 
beginning/ending scenes, 59 
defined, 56 
drawing w/o vertex buffer, 62–67 
drawing w/vertex buffers, 70 
example programs, 64–67 
indexed, 67–69 
line strips, 60–61 
lines, 59–62 
points, 59–62 
rendering states, 56–58 
triangle fan, 60, 62 
triangle strips, 60–61 
triangles, 59–62 
types, 59–62 
vertices associations, 64

pure software device, device type, 12

R
Range parameter, 182 
range, lighting, 182 
rays, triangle intersection points, 254–256 
reciprocal of the homogenous W (RHW), 41 
vertex information, 41
RECT parameter, 84
reference counting, z buffers, 172 
reference rasterizer, device type, 11–12 
Release function, 50, 53
render state parameter, 57–59
render states
alpha blending, 198–200 
parameters, 57–59 
z buffers, 173
render targets
creating, 89–91 
z buffer paring, 171
Reset function, 34–35
resource formats, checking, 19–21
RHW (reciprocal of the homogenous W) 
vertex information, 41
rotation
matrices, 150 
vectors, 135–136
RType parameter, 20

S
scalar multiplication, vectors, 132–133
scalar, defined, 133
scaling
matrices, 152 
vectors, 132–133
scenes
beginning/ending, 59 
presenting, 31–32
ScreenToClient function, 88
SetCursorPosition function, 88
SetCursorProperties function, 87
SetIndices function, 53, 68
SetLight function, 184
SetMaterial function, 179
SetRenderState function, 56–58, 173, 189, 198
SetRenderTarget function, 90, 171
SetStreamSource function, 50
SetTexture function, 97–98
SetTextureStageState function, 100
SetTransform function, 161–163
SetVertexShader function, 45
SetViewport function, 29
shearing, preventing with vertical 
retrace, 28
ShowCursor function, 88
Size function, 243
SizeToLock parameter, 49
specular color, vertex formats, 43
specular lighting
defined, 177 
setup, 192–193
Specular member, 180–181, 192
spotlights
Faloff member, 183
Phi member, 183
position/direction, 181–182
setup, 187–189
Theta member, 183

**Standard Template Library (STL), vector**

**template**, 216–218

**statements**, if, 167

**states**, rendering, 56–58

**stencil/depth surfaces**, 90–91

**streams**, vertex buffer associations, 50

**stride**, vertex type, 63

**strings**, 10–11

**strips**, 60–61

**structures**

  CustomVertex, 130, 176–177

**subtraction**, vectors, 132

**sunshine**, directional light type, 181–182

**surfaces**

  back buffers, 76–77
  copying between, 78–80
cursors, 86–89
defined, 74
depth/stencil, 90–91
image, 77–78
image loading, 81–86
locking images, 83–84
pixel plotting, 84–85
render targets, 89–91
resource types, 74–75
swap chains, 76

**SVERTEXCOLORLIST structure**, 231–233

**swap chains**, defined, 76

**SwapEffect member**, 27

**T**

**targets**, render, 89–91

**templates**

  curly braces { and }, 211
  header, 211
  Material definition, 211–212
  Mesh, 212–214
  mesh, 228–234
  MeshMaterialLists, 214–215, 228–229
  MeshVertexColors, 215
  vector, 216–218
  X files, 211–215

**TestCooperativeLevel function**, 34

**text**, tokenizing, 219–223

**texture coordinates**

  specifications, 96–97
  vertex formats, 44–45

**texture mapping**, defined, 96

**textures**

  address modes, 99–102
  applying to primitives, 97–99
  color argument constants, 106
  coordinate specifications, 96–97
creating, 94–95
  data loading, 96
defined, 94
  MIP mapping, 95
  modulation, 103
  multi-texturing, 102–106
  non-square support, 95
  size concerns, 95
  video card support checking, 105
  width/height parameters, 95

**Theta member**, 183

**TimeLeft member**, 246–247

**tokenizing**, defined, 219

**tokens**

  cube0, 233
  mesh, 223–234
  X files, 219–234

**transformations**, geometry pipeline setup, 161–163

**transformed vertices**, 45–46

**translations**, matrices, 149–150

**triangle fan**, primitive type, 60, 62

**triangle strips**, primitive type, 60–61

**triangles**

  end point testing, 256
  points, 254–256
  primitive type, 59–62
Type member, 180
Type parameter, 76
type sources, lighting, 180–181

U
UINT (unsigned int) return, 9
Unlock function, 49, 52, 84
untransformed vertices, 45–46
updates, dirty rectangle, 28
usage flags, vertex buffers, 47–48
Usage parameter, 20, 47
user pointer
drawing primitives w/o vertex buffer, 62–63
indexed primitives, 67–69

V
values
color-control, 113–114
inverted, 198
variables
COLORREF, 81
D3DCOLOR, 30
vertex buffers, 47–48
vertex data, flexible vertex format (FVF), 39–40
vertex, defined, 38
vertex formats
data representation, 39–40
described, 38
device notification, 45
diffuse color, 43
flags, 40–41
normal, 42
position, 40–41
reciprocal of the homogenous W (RHW), 41
specular color, 43
stride, 63
texture coordinates, 44–45
transformed versus untransformed, 45–46
vertex formats flag, 40–41
vertex list, mesh, 224–225
vertical retrace, shearing prevention, 28
video cards
See also display adapters
texture operation availability, 105
view space, described, 123

viewport
  clearing, 30–31
  described, 28–29
  determining currently used, 30
  dimension settings, 29–30

W

white space, X files, 211
WHQLLevel member, 10
Width parameter, 90
windowed applications, back buffer, 26–27
windows
  back buffer, 26–27
  top-level, 25

Windows Hardware Quality Lab (WHQL), 10
world space, described, 121–122

X

X file loaders
  custom, 216
  grouping types, 234–236
  importing file data into memory, 218

X files
  array elements, 214
  grouping types, 234–236
  header template, 211
  identifiers, 210
  Material definition template, 211–212
  Mesh template, 212–214
  mesh token, 223–234
  MeshMaterialList template, 214–215
  MeshVertexColors template, 215
  models, 207–215
  Standard Template Library (STL), 216–218
  templates, 211–215
  text versus binary versions, 210
  tokenizing text, 219–223
  vector template, 216–218
  white space, 211
  X file loaders, 216

XHotSpot parameter, 87
XMATERIAL structure, 223–234
XMATERIALLIST structure, 230–231
XScreenSpace parameter, 88
XVERTEXCOLOR structure, 231–233
XY plane, matrices, 150
XZ plane, matrices, 150–151

Y

YHotSpot parameter, 87
YScreenSpace parameter, 88
YZ plane, matrices, 151–152

Z

z buffering
  alpha blending concerns, 198
  calls, 167–168
  clearing, 173
  creating, 170–171
  depth formats, 169–170
  described, 166–167
  device format checking, 168
  enabling/disabling, 173
  if statements, 167
  locking/unlocking, 168
  reference counting, 172
  render states, 173
  render target pairing, 171
  setup methods, 167–171
  squares, 173–174
  uses, 172–174