

WORDWARE APPLICATIONS LIBRARY



DIRECTX[®] 9 GRAPHICS

The Definitive Guide
to Direct3D[®]

ALAN THORN

W7
WORDWARE
Publishing, Inc.

DirectX[®] 9 Graphics: The Definitive Guide to Direct3D[®]

Alan Thorn

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Thorn, Alan.

DirectX 9 graphics : the definitive guide to Direct3D / by Alan Thorn.
p. cm.

Includes bibliographical references and index.

ISBN-13: 978-1-55622-229-7

ISBN-10: 1-55622-229-7 (pbk.)

1. Computer graphics. 2. DirectX. I. Title.

T385.T49576 2005

006.6'93--dc22

2005003234

CIP

© 2005, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard

Plano, Texas 75074

No part of this book may be reproduced in any form or by
any means without permission in writing from
Wordware Publishing, Inc.

Printed in the United States of America

ISBN-13: 978-1-55622-229-7

ISBN-10: 1-55622-229-7

10 9 8 7 6 5 4 3 2 1

0503

DirectX and Direct3D are registered trademarks of Microsoft Corporation in the United States and/or other countries. Other brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

This book is sold as is, without warranty of any kind, either express or implied, respecting the contents of this book and any disks or programs that may accompany it, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Wordware Publishing, Inc. nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by this book.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

To Jean and Bill Mills, Peter Thorn, and Reginald Stokes

This page intentionally left blank.

Contents

	Acknowledgments	xi
	Introduction	xiii
Chapter 1	Getting Started with DirectX.	1
	What Is DirectX?	2
	A More Precise Definition of DirectX	4
	Obtaining DirectX.	5
	Installing DirectX	5
	Configuring DirectX.	6
	Exploring DirectX.	7
	DirectX Utilities.	9
	Conclusion	13
Chapter 2	Starting DirectX — Your First Program	15
	Getting Started	16
	Step 1 — Create a Window	17
	Step 2 — Create a Direct3D Object	19
	Step 3 — Create a Direct3D Device.	20
	Creating a Device	21
	More on Direct3D Devices	27
	Step 4 — Configure the Message Loop	28
	The Game Loop	29
	Step 5 — Render and Display a Scene	30
	Direct3D Surfaces — IDirect3DSurface9	31
	Preparing to Present a Scene	32
	Beginning and Ending a Scene	34
	Step 6 — Shut Down Direct3D	36
	More on Surfaces	37
	Creating Surfaces	37
	Loading Images onto Surfaces.	40
	Copying Surfaces	42
	Presenting Images with the Back Buffer	44
	Lost Devices.	46
	Conclusion	49

Chapter 3	3D Mathematics.	51
	Coordinate Systems	52
	One-Dimensional (1D) Coordinate Systems	52
	Two-Dimensional (2D) Coordinate Systems	53
	More on 2D Coordinate Systems	54
	Three-Dimensional (3D) Coordinate Systems	56
	Geometric Transformations	57
	Translation	58
	Rotation	58
	Scaling	60
	Vectors	61
	Length of the Diagonal (Magnitude).	63
	Vector Addition.	64
	Vector Subtraction	65
	Vector Multiplication.	66
	Vector Normalization	67
	Vector Dot Product	67
	Vector Cross Product	68
	Matrices	70
	Matrix Components	71
	Matrix Addition	72
	Matrix Subtraction	72
	Matrix Multiplication (Scalar)	72
	Matrix by Matrix Multiplication	73
	Identity Matrix.	74
	Inverse Matrix	75
	Matrices for Geometric Transformations	76
	Matrix Translation	76
	Matrix Rotation	78
	Matrix Scaling	81
	Combining Transformations	82
	Planes	83
	Creating Planes from Three Points	84
	Creating Planes from Point and Normal.	85
	Classifying Points in 3D Using Planes.	86
	Plane and Line Intersection	87
	Conclusion	89
Chapter 4	Direct3D for 3D Graphics	91
	Getting Started	92
	Create and Define a Vertex Format	93
	Using Vertices	94
	Create a Vertex Buffer.	95
	Fill the Vertex Buffer	97
	Rendering the Triangle	99
	View, Projection, and Transform Matrices.	99

	Transformation Matrix	100
	View Matrix	102
	Projection Matrix.	103
	Setting the Stream Source	104
	Setting the FVF	106
	Drawing a Primitive	106
	Animating the Triangle.	109
	Drawing Other Primitives	109
	Indexed Primitives	111
	Setting Up Index Buffers	113
	Drawing Indexed Primitives	115
	Conclusion	118
Chapter 5	Materials, Lights, and Textures	119
	Lighting.	120
	Turning the Lights On and Off	121
	More on Lights	122
	Setting the Ambient Lighting	122
	Getting Started with Direct Lights.	123
	Direct3D Shading Modes.	123
	Materials	124
	Direct Lighting Types	126
	Textures	131
	Creating Textures	132
	Creating Blank Textures	132
	Creating Textures from Image Files	134
	Texture Mapping	135
	Setting the Active Texture	137
	Texture Filtering	138
	Texture Addressing Modes.	141
	Texture Alpha Blending	144
	2D Texturing	145
	ID3DXSprite — Drawing Textures in 2D	147
	Conclusion	150
Chapter 6	X Files — Loading and Saving Data.	151
	Introduction to X Files	152
	Structural Overview	154
	Header	154
	Templates.	154
	Data Objects	156
	Parent and Child Objects	157
	Data Objects and References.	157
	Standard Templates.	159
	Custom Templates	169
	Reading X Files Using DirectX.	170

	Preparing	170
	Registering Templates	171
	Opening a File	173
	Enumerating Top Objects	174
	Enumerating Child Objects.	175
	Processing Child Objects.	176
	Enumeration Overview	176
	Getting Object Data	178
	Object Names.	178
	Object Types	179
	Object Data	181
	Saving Data to X Files — Save Object	182
	Preparing	183
	Saving Data.	184
	Building the Tree	185
	Committing the Data	186
	Conclusion	186
Chapter 7	Meshes	187
	What Are Meshes?	188
	How to Make Meshes	188
	How to Export Meshes.	189
	Testing Your Mesh	190
	Meshes in Direct3D	191
	Loading Meshes from X Files	191
	Loading Meshes from X File Data Objects.	193
	Mesh Materials and Textures	196
	Rendering Meshes	199
	Cleaning Up Meshes	200
	More on Meshes	201
	Meshes and Vertex Buffers	201
	Meshes and FVFs	202
	Bounding Boxes and Spheres	204
	Rays Intersecting Meshes	209
	Vertex Interpolation	212
	Conclusion	214
Chapter 8	Cameras — First-Person and More	215
	The Problem	216
	Overview	217
	Looking Around	218
	Pitch	218
	Roll	219
	Yaw	219
	Combining Rotations	220
	Moving the Camera	221

	Making the Camera.	222
	Initializing the Camera Class.	223
	Moving the Camera.	223
	Rotating the Camera.	224
	Building the View Matrix.	225
	Test Drive.	228
	Viewing Frustum.	229
	Constructing the Frustum.	230
	Testing for a Point.	232
	Testing for a Cube.	232
	Testing for a Sphere.	234
	Testing for a Mesh.	234
Chapter 9	Timing and Animation.	237
	Time.	238
	Keyframe Animation.	240
	Hierarchical Animation.	242
	Linked Lists.	243
	Adding Items to the List.	244
	Clearing a Linked List.	245
	Object Hierarchies for Animations.	245
	Conclusion.	248
Chapter 10	Point Sprites and Particle Systems.	249
	Particle Systems Overview.	249
	Particles in Direct3D — Point Sprites.	250
	Creating Point Sprites.	251
	Rendering Point Sprites.	253
	Particle Systems.	254
	Creating a Particle System.	256
	Conclusion.	260
Chapter 11	Playing Video and Animating Textures.	261
	Playing Video Using DirectShow.	261
	The 1, 2, 3 of Playing a File.	262
	Creating the Filter Graph.	262
	Media Control and Event System.	263
	Loading a Media File.	263
	Configuring Events.	264
	Playing a File.	268
	Playing Video — Further Information.	269
	Animating Textures.	270
	Conclusion.	272

Chapter 12 **More Animated Texturing** 273

 Movie Files (MPG, AVI, and More) 274

 Playing Video on Textures in Theory 274

 Playing Video on Textures in Practice 275

 Creating the Base Video Renderer 276

 Implementing the Base Video Renderer 277

 Implementing the Constructor 278

 Implementing CheckMediaType 278

 Implementing SetMediaType 279

 Implementing DoRenderSample 282

 Preparing the Filter Graph 287

 Conclusion 293

Chapter 13 **Skeletal Animation** 295

 What Is Skeletal Animation? 295

 Skinned Meshes 296

 The Structure of a Skinned Mesh 297

 Loading a Skinned Mesh from an X File 299

 Bone Hierarchies 302

 Loading the Bone Hierarchy 306

 Mapping the Bone Hierarchy to the Mesh 310

 Updating the Mesh 311

 Rendering the Mesh 314

 Animating the Skeleton 314

 Loading Animations 319

 Playing Animations 325

 Conclusion 328

 Afterword 329

Appendix A **DirectX Q&A** 331

Appendix B **Recommended Reading** 343

Index 345

Acknowledgments

There are a great many people who, in one way or another, have helped my second book through to completion. I would like to take this opportunity to thank the following individuals:

Wes Beckwith, Beth Kohler, and the rest of Wordware Publishing for their encouragement and professionalism.

My mother, Christine; my father, Gary; and my sister, Angela. I would like to thank them for their advice and support.

The rest of my friends and family for their care and attention, and for their help throughout the years.

Cassandra Gentleman. I would like to thank Cassie, a striking and intelligent lady who's tolerated all my mood swings during the production of this book.

Oh, and finally, I would like to thank you, the reader, for purchasing my book. I hope you will find it valuable.

This page intentionally left blank.

Introduction

So you want to make computer games? Or perhaps you want to produce alternative media products and digital art? If you do, then this could very well be the book for you. That's right; this book is about a technology designed to create first-rate multimedia products, from games to other interactive media. It's called Direct3D, and it's part of the DirectX suite of APIs distributed freely by Microsoft.

DirectX is one of the slickest, fastest, and most powerful technologies in town. Using DirectX you can produce computer games and multimedia products of the highest caliber, making use of the latest features of today's hardware. Simply put, using DirectX you can make stunning computer games. The possibilities are, in case you hadn't guessed, endless.

Interested? I hope so. Before we get started, however, let me tell you some things about this book, such as its intended audience, what you may ideally need to know, and how to obtain the companion files.

Who Should Read This Book?

Whether we like it or not, nearly every book is aimed at a specific audience. Now, while I wouldn't want to discourage any person from learning what's contained within these pages, I would like to stress that this book may not be suitable for all audiences. For example, you may be a student at a college or university who is looking to try his hand at some side project, or you may already be an industry professional keen to learn more or change your career direction. Ultimately, to understand and appreciate the knowledge

contained with these pages, you will ideally be a confident programmer familiar with the following:

- Windows programming
- Windows API and GDI
- Microsoft Visual C++
- Basic understanding of COM (Component Object Model)
- Foundational knowledge of mathematics

Why Should You Read This Book?

There could be any number of reasons why you'd want to read a book like this. Typically, you may be a student and game player who is now keen to try making your own games. Additionally, you may be someone looking to skill up for a job in the game or entertainment industry. On the other hand, you could simply be curious about how games are made and want to know more. Whatever the reason, if you're already a C++ programmer, you're likely to find this book interesting and informative.

How Should You Read This Book?

This book has been designed to be both a reference and a tutorial in order to suit programmers both new to and experienced with DirectX. It is divided into chapters that focus on a specific feature of DirectX. This means it should be simple for experienced DirectX coders to flick back and forth through the book and find what they need. Those new to DirectX can read this book through from beginning to end, trying their hand at each topic as they go.

What Is Contained in the Companion Files?

The companion files can be downloaded from

www.wordware.com/files/dx9graphics. They include the following:

- All code featured in this book
- Panda DirectX X file exporter for 3D Studio MAX

What Do I Need to Compile the Code Featured in the Book and in the Companion Files?

All the code samples featured in this book were written in Microsoft Visual C++. Thus, to compile this code you will need Microsoft Visual C++ 6.0 or .NET, or above. The book has the following requirements:

- Microsoft Visual C++ 6.0 or above
- DirectX 9 SDK Update — February 2005
- DirectX 9 SDK Extras Pack (for Visual C++ 6.0)

This page intentionally left blank.



Chapter 1

Getting Started with DirectX

If you want to make your own games and multimedia applications, the kind of applications with fast-paced 3D graphics, smooth animation, and real-time lighting, then you've come to the right place. This book provides a detailed and comprehensive guide on how to get started with DirectX, and specifically it focuses on Direct3D. In short, this book offers a detailed reference for people new to game making, or for those with some knowledge who want to know more. Now, when I say "games," I mean computer/video games like Doom, Quake, Half-Life, Black & White, etc. That's right, I'm talking big leagues.

This chapter begins our long and interesting journey into the world of game development. As you read these pages I'd like you to think of yourself as a fledgling game developer, and as a developer you will need to make various decisions about how you'll use the technology I'll teach you for making games. In this chapter I'll discuss the basics of DirectX and how it's used in the development of games, and I'll discuss specifically why it's useful to you and why you'd want to use it. Overall, this chapter will answer the following questions:

- What is DirectX?
- What components does DirectX include?
- How can DirectX be obtained?

- How is DirectX installed?
- What's the difference between DirectX runtime and DirectX SDK?
- How do you set up DirectX for your development environment?
- What utility software comes with DirectX?

What Is DirectX?

Imagine this: You want to create an adventure exploration game. The game will be first-person perspective, meaning the player views the game world as he'd see it from the eyes of the game character. So, the game allows the player to look left, right, up, and down and also allows the character to walk forward, backward, left, and right. Furthermore, the character will be able to shoot a gun at creatures that try to attack. To create this game a developer must consider many factors. Some of these include the following:

■ **Engine**

A developer will need to make an engine for the game. This means you must code the physics of the 3D world, including features like gravity, wind, and rain. Furthermore, a developer must also perform all the calculations for lighting, like the lengths and angles at which shadows are cast from a light source — like the sun — at a particular point in 3D space.

■ **Graphics**

A developer will need to produce all the images and artwork used for the game. Then he or she must find a way to load all the 3D models — like game characters and scenery — into the game. Additionally, the developer must texture map the scenery (in other words, make it look real).

■ **Sound**

A developer is responsible for all the game sounds, from the sounds on option screens and menus to in-game sounds. The latter requires additional thought because in-game sounds can

use advanced technologies to enhance their realism, such as 3D sound. Using 3D sound, a developer can play sounds from selected speakers and pan sounds in various directions across multiple speakers. This can give the effect of a sound having a position in 3D space — such as a 3D game world — and as the player moves around, the sound will be heard differently. For example, as the player moves farther away from the sound, its volume will decrease, or as the player turns his head, the sounds heard in each ear will have reversed.

■ **Multiplayer**

Nowadays many games can be played on the Internet, meaning people can play against or cooperatively with people from other nations and other parts of the world. From the perspective of the developer, this gives us more issues to consider. Specifically, the developer must develop a system whereby each running game, from machine to machine, can communicate and transfer data to the other as the game is being played.

Whew! That's a lot of stuff to be thinking about; makes me dizzy just writing about it. As you're no doubt beginning to see, developing a first-person exploration game is not a simple process. It involves a combination of many disciplines — from graphics and music to programming and mathematics. Naturally, then, game developers are always on the lookout for ways their lives can be made easier. Enter DirectX...

Now, if we wanted to make that first-person exploration game, we could code everything from scratch — the sound system, the mechanism by which we texture map 3D objects, and the very system by which images are drawn to the screen. Or, we could use a library of premade functions and classes, which we can call upon to do a lot of the work for us. This, simply put, is what DirectX is.

A More Precise Definition of DirectX

DirectX is an SDK (software development kit) composed of a collection of COM libraries. These libraries provide a variety of functions and classes to make the process of developing games and multimedia applications simpler for the developer. Specifically, DirectX is a collection of smaller APIs. These are:

■ **Direct3D**

This API forms the main subject matter for this book. Direct3D is a graphics API that makes the process of drawing images to the screen simpler. Using Direct3D you can load 3D models, texture map models, represent and animate 3D worlds, and generally display flashy graphics. This component takes care of the visual aspect of games. In other words, more or less everything you see in a video game is developed with Direct3D.

■ **DirectShow**

DirectShow is responsible for media streaming. It can play movie and sound files, .MPG files, .MP3 files, and more. Think of it like a programmable media player. Using DirectShow you can load and play many kinds of media files.

■ **DirectInput**

DirectInput is the component that reads data from peripheral input devices such as the mouse, keyboard, and joystick.


■ **DirectSound**

This component allows developers to play digitized sounds. Using DirectSound, you can play sounds in a 3D environment. Unfortunately, DirectSound is beyond the scope of this book.

■ **DirectPlay**

DirectPlay is an API used to create and design multiplayer games, or games that need to communicate via networks.

*** NOTE.** DirectX is distributed in two modes by Microsoft: runtime and SDK. The runtime DirectX is typically downloaded and installed only by users of DirectX applications, not developers. In other words, the runtime package contains only that data that is required to run DirectX applications; it does not include the library, include, or help files that are needed to develop DirectX applications. The SDK is the full package. This includes all files needed to both develop and run DirectX applications.

 **TIP.** The DirectX SDK often installs the debug versions of the DirectX DLLs. This makes it simpler for developers to debug their DirectX applications, but results in a performance penalty. To solve this, you can install the DirectX runtime after installing the SDK. This replaces the debug DLLs with the retail versions.

Obtaining DirectX

So now that you've learned more about DirectX — in terms of what it is, what it does, and how it's useful — you'll probably want to know how you can get a copy. Fortunately, obtaining DirectX is really simple. You can download the most current version of DirectX from Microsoft's web site. The DirectX web site can be found at: <http://www.microsoft.com/windows/directx/default.aspx>.

Installing DirectX

Installing DirectX is a simple, one-time process. Once completed, the DirectX SDK will be ready to use. To begin the DirectX installation you just need to double-click on the setup.exe file after you download it. As installation begins, it'll look something like Figure 1.1.

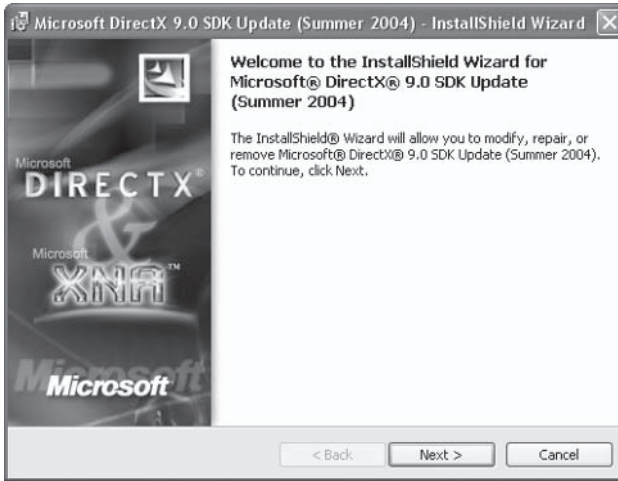


Figure 1.1

*** NOTE.** Before installing the latest version of DirectX, you should ensure any previous versions of DirectX have been uninstalled.

By clicking the Next button and moving from screen to screen, you can select a path to install and other options.

After installation has finished you'll want to restart your computer before coding anything.

Voilà! Installation complete. It's that simple.

Configuring DirectX

Now it's time to examine how you can set up your development environment, like Visual C++ 6.0 and .NET, to use Microsoft DirectX. In other words, you'll need to tell your development environment which directories to search when referencing the DirectX SDK libs and includes. Thankfully, most of the time the DirectX installer will automatically configure itself for use in your development environment. However, there may be occasions when this doesn't occur. In order to set up your environment you will need to perform the following steps:

For Visual C++ 6.0:

1. Click **Tools** | **Options** from the main menu.
2. Select the **Directories** tab.
3. Select **Include Files** from the right-hand drop-down list.
4. In the Directories list, add a path to the top of the list that points to the DirectX\Include path.
5. Select **Library Files** from the right-hand drop-down list.
6. Again, in the Directories list, add a path to the top of the list. This time, provide the DirectX\Lib path.
7. Click **OK**.

For Visual C++ .NET:

1. Click **Tools** | **Options** from the main menu.
2. Click the **Projects** category.
3. Click the **VC++ Directories** option.
4. Select **Include Files** from the top right-hand drop-down list.
5. Add a path to the DirectX\Include folder using the New tool button.
6. Select **Library Files** from the top right-hand drop-down list.
7. Add a path to the DirectX\Lib folder using the New tool button.
8. Click **OK**.

Exploring DirectX

Once the DirectX SDK has been installed, it's always a good idea to check out the newly installed files and folders. DirectX will have been installed to a path of your choosing; typically this may be C:\DXSDK or a similar path. Try opening up this folder using the standard Windows Explorer to take a look. You'll notice at least the

folders shown in Figure 1.2. The following list describes each folder's purpose.

■ **Developer Runtime**

This folder houses the redistributable runtime version of DirectX. In other words, this folder includes the DirectX installation that your users will need in order to run your DirectX applications.

■ **Documentation**

This folder is where the help files and legal documentation are located.

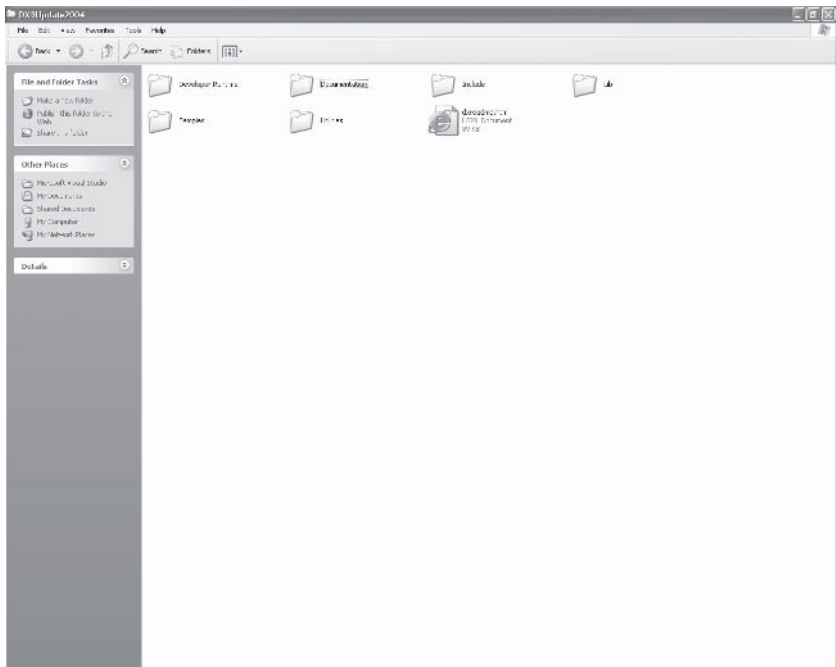


Figure 1.2

■ **Include**

The Include directory contains all the DirectX source files, some or all of which you'll need in order to develop DirectX applications.

- **Lib**

This folder contains the libraries (.lib files) that your development projects will reference.

- **Samples**

The Samples folder is where you'll find all the example projects and other media that have been included in the SDK to demonstrate how DirectX works. Take care, you could get lost here, as there are lots of them.

- **Utilities**

The Utilities folder is home to a number of tools distributed with the SDK. These are small applications intended to make the developer's life a little easier.

DirectX Utilities

The DirectX SDK comes with a number of utility applications. These can be found on the Windows Start menu in the DirectX program group. The purpose of these utilities is to assist the developer in producing DirectX software. The utilities can also be found in the Utilities folder where DirectX was installed. The following paragraphs briefly describe each utility.

- **D3DSpy**

This utility is a debugging tool that can monitor running DirectX applications and provide developers with a report on the current status of DirectX. It can list the calls made to the API and the current values of DirectX's internal variables.

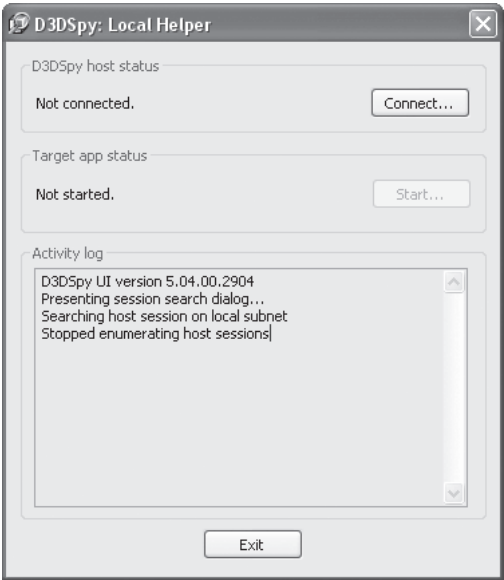


Figure 1.3

■ **DirectX Caps Viewer**

DirectX Caps Viewer is an abbreviation for the DirectX Capabilities Viewer. This application is like the standard system information tool that accompanies Windows. It provides detailed information about DirectX and the computer’s relevant hardware components.

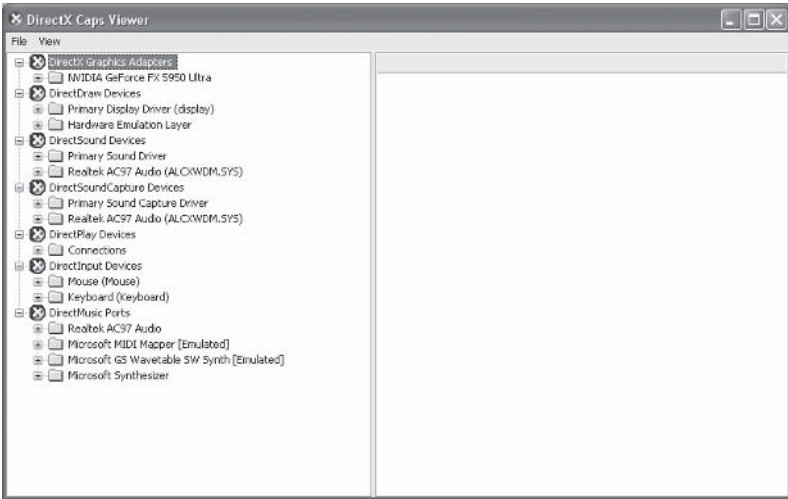


Figure 1.4

■ DirectX Diagnostic Tool

The DirectX Diagnostic Tool provides information about DirectX and pertinent system information. You can launch this window by clicking Start | Run and typing DXDIAG.

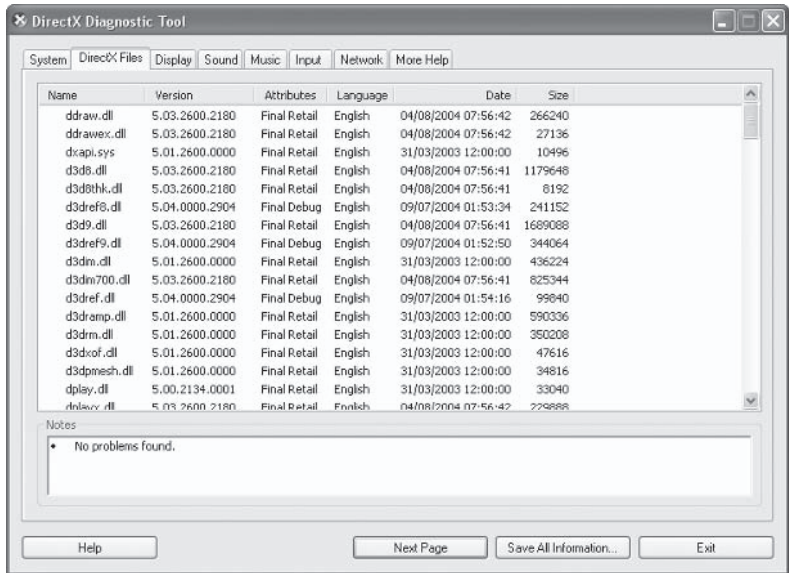


Figure 1.5

■ DirectX Error Lookup

The DirectX Error Lookup tool takes a DirectX error code — in binary or hexadecimal format — and converts it into a human readable string that describes the error.

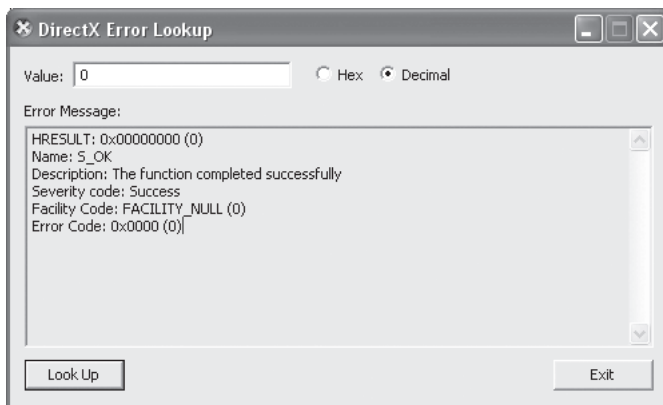


Figure 1.6

■ **DirectX Texture Tool**

This tool is a miniature image converter that can convert images like JPEGs, bitmaps, and Targas into DDS, the native DirectX image format.

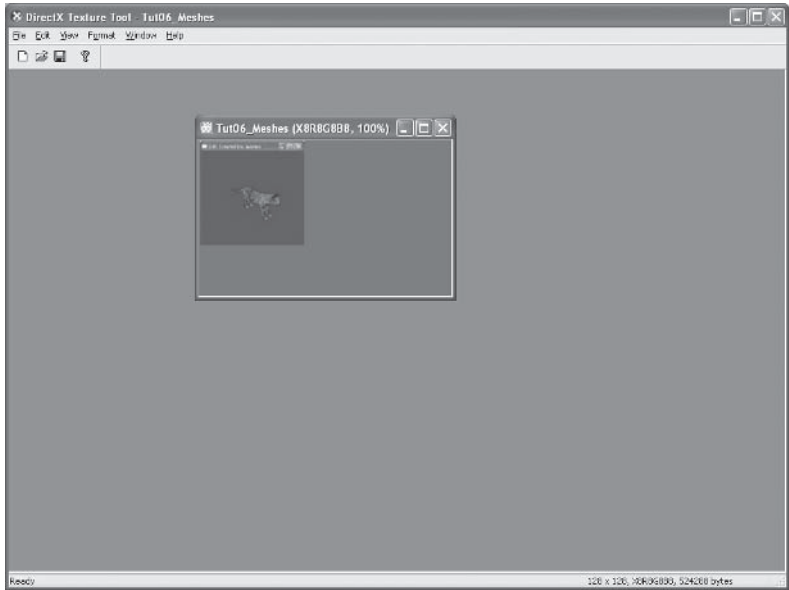


Figure 1.7

■ **Mesh Viewer**

The DirectX MeshView is a small utility for viewing 3D meshes (models) that have been saved in the DirectX X file format (explained later). This is a good way to test your models and see if they will appear properly in Direct3D.

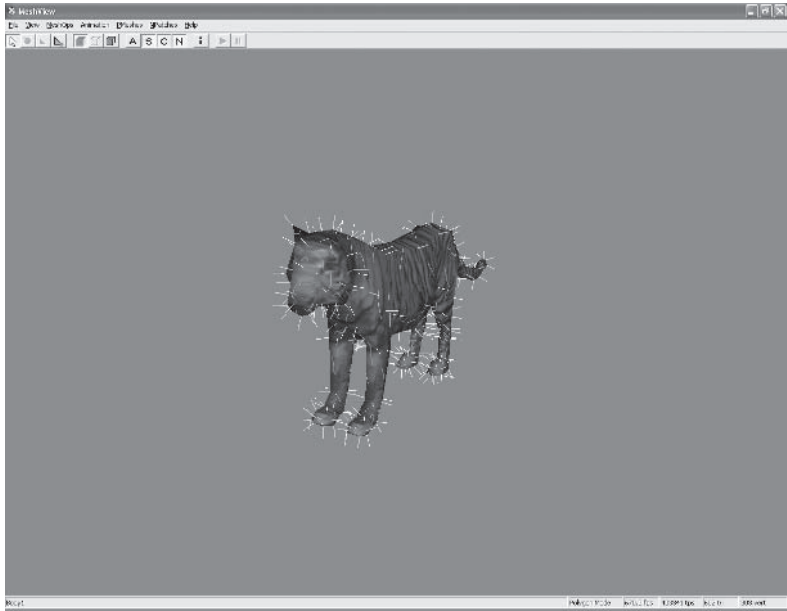


Figure 1.8

Conclusion

This chapter was a real walk in the park; nice and simple stuff. It explained how to get started using DirectX. Specifically, it explained how DirectX is a software development kit composed of a variety of APIs. These APIs are designed to make game and multimedia development simpler for developers.

The next chapter looks more closely at DirectX, and it specifically focuses on Direct3D. That's right — the next chapter starts our journey into the world of great, fast-paced graphics.

This page intentionally left blank.



Chapter 2

Starting DirectX — Your First Program

The previous chapter explained how DirectX is an SDK (software development kit) developers use to produce multimedia products, like games. It mentioned specifically how Direct3D is a component of DirectX that is used to render cutting-edge graphics, both 2D and 3D. Finally, it demonstrated how the DirectX SDK can be obtained from Microsoft's web site, then installed and configured for your development environment.

This chapter is the first of many to focus on how Direct3D works and should help you code your first Direct3D application. This application won't be anything spectacular, though. It'll simply draw a bitmap image in the top-left corner of the screen, but it will demonstrate some important concepts. In this chapter you will learn the following:

- How to create and configure a Direct3D application
- The purpose of a Direct3D device
- How to structure a game message loop
- How to draw images to the screen
- How to handle lost devices
- How to free Direct3D interfaces

Getting Started

This chapter demonstrates the most basic and essential features of Direct3D. Specifically, it shows you how to structure Direct3D applications, including the message loop, and explains how images can be loaded from a file and drawn onto the screen. No flashy 3D stuff, just plain, simple 2D for the moment. From this you will get a feel for how Direct3D works, making this a great starting point for any developer.

To make any Direct3D program we perform the following coding steps, which are examined as the chapter progresses:

1. **Create a window**

This is a standard window, identified by a window handle. To create a window you can use the normal Win API functions or other libraries like MFC. The purpose of creating a main window, as should be obvious, is to receive messages and provide a canvas upon which Direct3D will render images. In other words, *Direct3D will draw all its data onto this window.*

2. **Create a Direct3D object**

This is the main object that all Direct3D applications have. It is created when the application starts and destroyed when the application ends. By having this object it's like saying, "Hey, I'm a Direct3D application." Its primary use will be to create other objects, which will do stuff like drawing images, etc., for us.

3. **Create a Direct3D device**

One of the other objects a Direct3D object (step 2) creates for us is a Direct3D device. A Direct3D device represents the system's graphics card and uses it to draw images. In other words, *we use the Direct3D device to actually draw images in our app's window.*

4. **Configure the message loop**

Once applications create their main window they usually enter a message pump. Here, the window often waits around for messages and redraws itself on WM_PAINT. Direct3D applications

don't hang around, though; they repeatedly redraw themselves whenever no other messages need processing. Redrawing is performed by the Direct3D device (step 3) and this occurs many times a second.

5. **Render and display a scene**

This is like the drawing we'd normally do on `WM_PAINT`. On each redraw the scene is rendered (displayed). By this I mean we draw our data using the Direct3D device. As mentioned in step 4, this process occurs many times per second whenever there are no messages to handle. In this chapter we will be drawing a simple image. Later, we'll be rendering 3D models, etc.

6. **Shut down Direct3D**

As the application ends, the message loop finishes and all created objects (steps 2 and 3 for example) must be released. If you've used COM objects before, you'll be familiar with the standard `Release` method.

Step 1 — Create a Window

So you want to draw a simple image in Direct3D? An image loaded from a file? First things first. All Direct3D applications begin by creating a main application window; this is just a standard window and will be identified by its window handle. Direct3D will later use this window as a canvas, like a giant sketchpad or a surface onto which it'll continually draw (render) a scene.

*** NOTE.** When I use the word "scene" I mean a final image. This could be an image loaded from a file or it could be an image snapshot taken from a complex 3D world. No matter from where the image is taken or how the image is created, a "scene" refers to the final image the user/player will see on the screen every frame.

There are many ways to create a window, and it's assumed you already know at least one of them. However, for convenience and thoroughness, I have provided an example. A window is often created in the WinMain function of an application, like this:

```
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
{
    // Register the window class
    WNDCLASSEX wc = {sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                    GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                    TEXT("WindowClass"), NULL};
    RegisterClassEx(&wc);

    // Create the application's window
    HWND hWnd = CreateWindow(TEXT("WindowClass"), TEXT("Test"),
                            WS_OVERLAPPEDWINDOW, 100, 100, 512, 512,
                            GetDesktopWindow(), NULL, wc.hInstance, NULL);
```

The above code is probably not new to you. It creates a window 512x512 pixels in size and stores its window handle in hWnd. The created window does not contain a menu but does contain both minimize and maximize buttons. After this, an application would proceed to update and show the window before entering the message pump. This stage is shown later in the section titled “Step 4 — Configure the Message Loop.” Once a window has been created and displayed, an application then proceeds by creating a Direct3D object.

***NOTE.** You can create a window using other methods too. You could even use MFC. For your convenience, a sample project that creates a window has been included in the companion files and can be found in Code\Chapter2\Proj.vcproj.

Step 2 — Create a Direct3D Object

A Direct3D application must begin by creating a Direct3D object. This is a COM interface of type `IDirect3D9`. Applications typically declare this as a global object that is instantiated at application startup and destroyed at application end. You only need one of these objects and it represents the lifespan of a Direct3D application. Creating this object is the equivalent of saying, “Hey, I’m a Direct3D application.” Its primary purpose is to create other objects, such as a Direct3D device, as we’ll see later. To create a Direct3D object we must call the `Direct3DCreate9` function. It accepts one parameter, the current DirectX version number. Its syntax and parameter can be seen below.

```
IDirect3D9 *WINAPI Direct3DCreate9(
    UINT SDKVersion
);
```

UINT SDKVersion

The value of this parameter should be `D3D_SDK_VERSION`.

*** NOTE.** Don’t forget the Direct3D include files: `#include <d3d9.h>` and `#include <d3dx9.h>`.

This function is really simple to use. We just pass it `D3D_SDK_VERSION` to confirm we want to use the current version of DirectX. On successful completion, the function returns a pointer to a valid Direct3D object, which we’ll need to hold onto throughout our application. We’ll use it later. The code to create a Direct3D object would look like this:

```
IDirect3D9* g_pD3D = NULL;
if(NULL == (g_pD3D = Direct3DCreate9(D3D_SDK_VERSION)))
    return E_FAIL;
```

*** NOTE.** Rather than using `IDirect3D9*`, you can also use the typedef `LPDIRECT3D9`.

There, now we have a proper Direct3D application. This object can then be employed to help set up our application, ready for drawing something. This is done by first creating a Direct3D device.

Step 3 — Create a Direct3D Device

In the previous section we saw how a Direct3D object (IDirect3D9) is the lifeline of a Direct3D application. It's mostly created at the beginning and destroyed at the end, and it's used to create other objects. In this section we use IDirect3D9 to create the most important of these objects: the Direct3D device. This is a COM interface of type IDirect3DDevice9 and it's essential if you expect to actually draw anything on the screen. It represents a computer's graphics card or some other graphical hardware. It's important to note that one Direct3D device can only represent one piece of hardware; if you wanted to program two cards, you'd create two devices, and so on. Each device has a unique ID to distinguish it from any others. This book focuses on using only one device, the primary graphics device attached to the system.

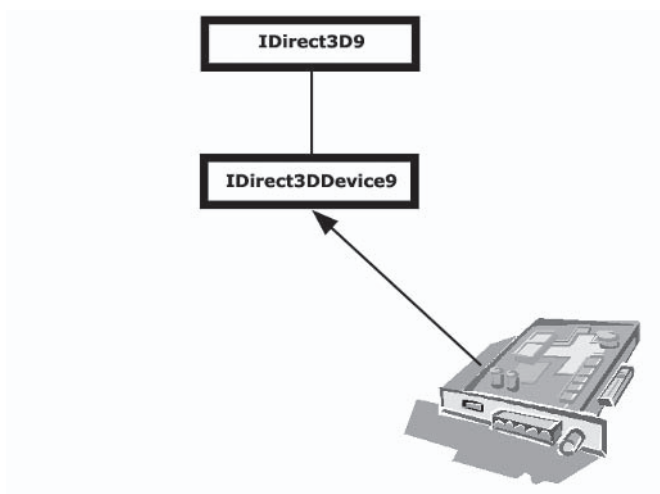


Figure 2.1: Direct3D devices

*** NOTE.** Graphic cards with dual heads are considered two devices.

In Direct3D, a device can be one of the following two types:

■ **HAL device**

The primary device type, and the one used in this book, is a HAL device. This stands for hardware abstraction layer. This device type includes the main 3D accelerated cards used to play games.

■ **Reference device**

This is a software-exclusive device that supports every Direct3D feature. However, these devices are often slow and are typically used only by developers to test features that will later be implemented in hardware. This book does not use this device type.

Creating a Device

Direct3D devices are created by the **CreateDevice** method of **IDirect3D9**. This method requires several parameters, including the unique ID of the device to be created, the device type, a window handle, and some behavior flags specifying how the created device should operate. Once successful, this function returns a valid pointer to an **IDirect3DDevice9** interface. The **CreateDevice** syntax and parameters appear as follows:

```
HRESULT IDirect3D9::CreateDevice(  
    UINT Adapter,  
    D3DDEVTYPE DeviceType,  
    HWND hFocusWindow,  
    DWORD BehaviorFlags,  
    D3DPRESENT_PARAMETERS *pPresentationParameters,  
    IDirect3DDevice9** ppReturnedDeviceInterface  
);
```

UINT Adapter

[in] Ordinal number denoting the display adapter. This parameter represents the device to use. This book uses `D3DADAPTER_DEFAULT` since this is the primary display adapter.

D3DDEVTYPE DeviceType

[in] Member of the `D3DDEVTYPE` enumerated type that denotes the desired device type. If the desired device type is not available, the method will fail. This value will be `D3DDEVTYPE_HAL`.

HWND hFocusWindow

[in] Handle of the window to be associated with the created device. `IDirect3DDevice9` will use this window as a canvas for drawing upon. This value will be the window you created in Step 1 earlier.

DWORD BehaviorFlags

[in] One or more flags indicating how the device should behave. This book uses `D3DCREATE_SOFTWARE_VERTEX-PROCESSING`. For now, don't worry too much about the significance of this parameter.



TIP. Direct3D devices are not by default created to support multithreaded applications. However, by passing a value of `D3DCREATE_MULTITHREADED` in the `BehaviorFlags` parameter, you can support threading, although this will entail a performance hit.

D3DPRESENT_PARAMETERS *pPresentationParameters

[in] This is a pointer to a `D3DPRESENT_PARAMETERS` structure, which specifies how your created device should operate. Using this structure you can specify, among other things, screen resolution and whether the application runs full screen or in a window. The structure looks like this:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT BackBufferWidth, BackBufferHeight;
    D3DFORMAT BackBufferFormat;
    UINT BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD MultiSampleQuality;
    D3DSWAPEFFECT SwapEffect;
    HWND hDeviceWindow;
    BOOL Windowed;
    BOOL EnableAutoDepthStencil;
    D3DFORMAT AutoDepthStencilFormat;
    DWORD Flags;
    UINT FullScreen_RefreshRateInHz;
    UINT PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

Again, don't worry if you don't understand what each parameter does. Many can be left blank and most are explained as this book progresses. For reference, a brief description of this structure follows.

UINT *BackBufferWidth*, UINT *BackBufferHeight*

Specifies the width and height of the back buffer (explained later). If your application is running in full-screen mode, then these parameters must match the screen resolution. If your application is running in a window, then they may be any value. 0 indicates a back buffer whose dimensions match the window size.

D3DFORMAT *BackBufferFormat*

This is an enumeration of type D3DFORMAT. It specifies the color format of the back buffer and is something like 256-color, or 16-, 24-, or 32-bit color. It also defines how the RGB (red, green, blue) and alpha components are arranged. There are many possible values. You can also specify D3DFMT_UNKNOWN if the format is not known.

UINT *BackBufferCount*

The number of back buffers (explained later) you desire. This can be zero or more. Mostly you will only require one.

D3DMULTISAMPLE_TYPE *MultiSampleType*

Multisampling can achieve graphical effects like antialiasing, motion blurring, and more. Typically you won't need this, so you can pass D3DMULTISAMPLE_NONE.

DWORD *MultiSampleQuality*

Indicates the quality. This value can be left blank if MultiSampleType is D3DMULTISAMPLE_NONE.

D3DSWAPEFFECT *SwapEffect*

Describes how the back buffer behaves when “flipped” (explained later). You will nearly always specify D3DSWAPEFFECT_DISCARD.

HWND *hDeviceWindow*

This is a handle to the window where rendering will occur.

BOOL *Windowed*

Boolean value indicating whether the application will run full screen or in a window. True for a window and False for full screen.

BOOL *EnableAutoDepthStencil*

You'll usually set this to False. Passing True will allow DirectX to manage your depth buffers.

D3DFORMAT *AutoDepthStencilFormat*

If EnableAutoDepthStencil is True, you should specify the color format of the depth buffers, much like how you specified the color format of the BackBufferFormat parameter.

DWORD *Flags*

Typically you will pass D3DPRESENTFLAG_LOCKABLE_BACKBUFFER or you can leave this blank.

UINT FullScreen_RefreshRateInHz

Defines the screen refresh rate in Hz per second. So, 75 would mean 75 updates per second. For windowed mode this value must be 0.

UINT PresentationInterval

Specifies how fast the back buffer is presented. For windowed mode you must specify D3DPRESENT_INTERVAL_DEFAULT, and you'll likely always use this value for any application.

IDirect3DDevice9 ppReturnedDeviceInterface**

[out, retval] Address of a pointer to the returned IDirect3DDevice9 interface. If the function is successful, then a valid pointer to your created device will be returned here.

Some sample code to create a standard windowed Direct3D device might look like this:

```
//Pointer to a Direct3D device
IDirect3DDevice9 *g_pd3dDevice = NULL;

D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.Windowed = true;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;

// Create the D3DDevice
if(FAILED(g_pD3D->CreateDevice(D3DADAPTER_DEFAULT,
                              D3DDEVTYPE_HAL, hWnd,
                              D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                              &d3dpp, &g_pd3dDevice)))
{
    return E_FAIL;
}
```

*** NOTE.** Notice how we are automatically selecting the primary graphics card by passing `D3DADAPTER_DEFAULT` to `CreateDevice`.

Or, to create a standard **full-screen Direct3D device**, you might code something like the following. Remember, by creating a full-screen device, Direct3D will automatically enter full-screen mode and change the computer's resolution to the one you specified.

```
//Pointer to a Direct3D device
IDirect3DDevice9 *g_pd3dDevice = NULL;

D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.Windowed = false;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
d3dpp.BackBufferCount = 1;
d3dpp.BackBufferWidth = 1024;
d3dpp.BackBufferHeight = 768;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;

// Create the D3DDevice
if(FAILED(g_pD3D->CreateDevice(D3DADAPTER_DEFAULT,
                                D3DDEVTYPE_HAL, hWnd,
                                D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                &d3dpp, &g_pd3dDevice)))
{
    return E_FAIL;
}
```

*** NOTE.** Many games do not hard-code all parameters passed to `CreateDevice`; they often allow the user to select things like the resolution, etc., in an options screen.

More on Direct3D Devices

You have already seen how Direct3D devices represent graphical hardware and are very important for Direct3D applications. This section provides a brief overview of some things `IDirect3DDevice9` can do. Many of these points are explained in more detail later.

With a Direct3D device you can do all the following and more:

- **Render a scene to a window**

That's right; a Direct3D device will be used to draw (present) images in its associated window, on a frame-by-frame basis. Later in this chapter, the Direct3D device will be used to load an image from a file and draw it in the window. To do this, we'll use a combination of its methods. This is explained later.

- **Query device capabilities**

`IDirect3DDevice9` tells us much about the hardware it represents. Using its `GetDeviceCaps` method to populate a `D3DCAPS9` structure, we can check its properties to see what our hardware can and can't do. The `GetDeviceCaps` method looks like this:

```
HRESULT IDirect3DDevice9::GetDeviceCaps(  
    D3DCAPS9 *pCaps  
);
```

- **Set a cursor image**

`IDirect3DDevice9` can be used to show a cursor image and set its position on screen. This is shown later when we examine the `SetCursorPosition` and `SetCursorProperties` methods.

- **Create and manage graphical resources**

A Direct3D device can create graphical resources in memory, like loading images from files or loading 3D models.

- **Simulate 3D environments**

`IDirect3DDevice9` provides methods to situate 3D models in 3D space and then draw them correctly in the window. Objects will be drawn according to their distance from the camera. It will also ensure that objects closer to the camera will properly occlude those objects behind them.

Step 4 — Configure the Message Loop

A window was created in Step 1, then a Direct3D object (Step 2), and then a Direct3D device (Step 3). Right now, if you compiled and ran your application you'd immediately notice a problem. It doesn't run. Well, it does run but immediately exits so you never see anything. This is because — as you should already know — there is no message pump in action. Typically an application should enter a loop, called the message loop, and this loop continues until a user exits an application. During this loop the application receives and processes messages, which are generated as events occur, like when a mouse button is clicked or a key is pressed. When a user quits an application, a `WM_QUIT` message is generated, the loop exits, and the application ends. The following shows how this might look in code.

```
MSG msg;
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

This kind of loop is the standard kind you're probably used to by now. It's useful for many Windows applications because it means they consume little processor time, as they just sit around and only process messages as they occur. This means Windows applications don't generally interfere with each other; they only consume CPU power when instructed to and they don't get in the way too much. Games are different, however.

The Game Loop

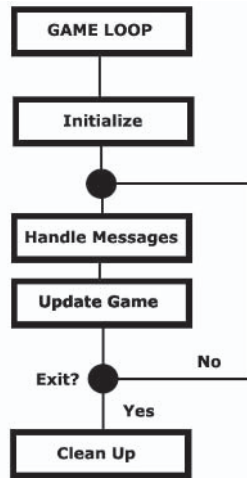


Figure 2.2: Game message loop

Games are greedy. They require a different kind of message loop entirely, a message loop unlike the standard one. Instead, games require a lot of power. This is for the following reasons:

■ Graphics

Games need to ensure their graphics are updated and redrawn many times per second so animations appear smoothly. The number of times it redraws per second is known as FPS (frames per second).

■ Input

Games need to have a very exact knowledge of which controls a user presses and when in order to ensure the game reacts appropriately.

■ Timing

Games require very precise control over timing. This is because many events, like lip syncing, need to be synchronized.

The standard window message loop can be amended to accommodate games easily, as shown in the following code fragment. Like before, this loop runs until the user exits with a `WM_QUIT`

message, but when no messages occur the application does not sit around; instead, we utilize all the time we have. When there is a message to process, we process it. When there are no messages to process, we continually process our game data, again and again.

```
MSG mssg;

PeekMessage(&mssg, NULL, 0, 0, PM_NOREMOVE);
    // run till completed
    while (mssg.message!=WM_QUIT)
    {
        // is there a message to process?
        if (PeekMessage(&mssg, NULL, 0, 0, PM_REMOVE))
        {
            // dispatch the message
            TranslateMessage(&mssg);
            DispatchMessage(&mssg);
        }
        else
        {
            //No message to process?
            // Then do your game stuff here
            Render();
        }
    }
}
```

This game loop continually calls my application's `Render` function, which I set aside for processing game data, and specifically for drawing images to the screen as we'll see later. You can call your function something different if you want. Next we examine how a `Direct3D` device presents a scene (draws data) during this loop.

Step 5 — Render and Display a Scene

If you compile and run your application using all the steps previously mentioned, you'll notice it runs and shows a window that remains until you exit; however, there is still a problem. The window will display all kinds of artifacts in its client area, and this gets

worse if you drag the window around the desktop or drag other windows in front of it. If your application is full screen, it may instead just look like a dull gray background. The problem is that nothing is being drawn on the window. It never gets updated, cleaned, or redrawn during the game loop. This is one of the duties of the Direct3D device (IDirect3DDevice9), and this section examines how to clear the window and draw an empty scene. It's empty right now because we have no 3D objects to draw or any images loaded from a file. Before we can draw this empty scene, however, you'll need to know a little about how Direct3D renders scenes.

Direct3D Surfaces — IDirect3DSurface9

Direct3D uses surfaces to perform its rendering. A surface is a flat rectangle of bytes in memory. Like a canvas, a *surface* stores an image; it could be a bitmap or just a blank image. If you've used GDI before, then a device context would be a similar analogy. Just think of a sheet of cardboard with a picture on it. Direct3D can create and store as many surfaces as memory will allow. You can even create surfaces to hold your own images. Surfaces can also be copied entirely or partially to and from one another. So, if you wanted to load two bitmaps into your application (a process shown later in this chapter), you would create two surfaces for each one and load the images from the files onto the surfaces.

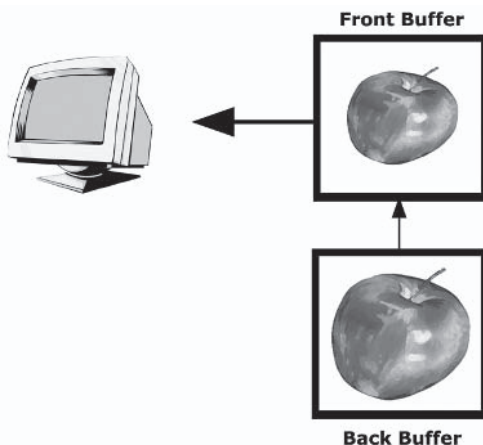


Figure 2.3: Monitor, front buffer, and back buffer

When Direct3D presents a scene to the window it actually, behind the scenes, uses a surface to do this. That's right; it presents the final scene on a surface. Seems sensible. In fact, Direct3D keeps track of two special surfaces exclusively for the rendering process. These surfaces are called the front buffer and the back buffer. The *front buffer* is the final surface you see on screen — that is, the final image presented in the window. The *back buffer* is a hidden surface — like a work in progress — being prepared for the next frame. The back buffer is actually where the 3D device will draw all your data first. Then, on the next frame, the two surfaces (the front buffer and the back buffer) are swapped (flipped). So the back buffer gets pushed to the front where it can now be seen — becoming the new front buffer — and the old front buffer gets moved to the back where it's cleared and becomes the new back buffer. And so the process goes on. Just remember that the front buffer is the surface the user will always see, while the back buffer is an area being prepared for the next frame. You never edit the front buffer, only the back buffer. Take a look at Figure 2.3 for an example of the front and back buffers and the flipping process.

Direct3D abstracts a surface with the `IDirect3DSurface9` interface — one interface per surface. Later sections explain how to create surfaces, load image data onto them, and copy this data between multiple surfaces. For now, it is sufficient to understand what surfaces are and how Direct3D uses them to render images.

Preparing to Present a Scene

A Direct3D device typically presents a scene on every frame. To begin presenting a scene you first clear the back buffer, making it ready for drawing to occur. This wipes the surface clean and fills it with a color we choose. To do this, we use the **Clear** method of `IDirect3DDevice9`. `Clear` takes several parameters and gives us the option to clear the entire back buffer, or we can pass a series of `RECT` structures to clear only specified areas. Its syntax and parameters follow.

```

HRESULT Clear(
    DWORD Count,
    const D3DRECT *pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,
    DWORD Stencil
);

```

DWORD Count

[in] The number of rectangles in pRects. If you want to clear the entire surface, as you most often will, specify 0 for this value.

const D3DRECT *pRects

[in] An array of RECT structures defining regions on the back buffer to clear. To clear the entire surface this can be NULL.

DWORD Flags

[in] This defines the surface you want to clear since there are other special buffers Direct3D maintains in addition to the back buffer. Most often you will pass D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER.

D3DCOLOR Color

[in] This specifies the color to use for clearing either the entire surface or the areas specified by pRects.

*** NOTE.** Color in Direct3D can be specified using the D3DXCOLOR_XRGB macro.

For example,

This is blue: D3DXCOLOR_XRGB(0,0,255).

This is red: D3DXCOLOR_XRGB(255,0,0).

This is green: D3DXCOLOR_XRGB(0,255,0).

RGB stands for red, green, and blue. By providing values from 0 to 255 in each of these components, you can generate a final color value.

float Z

[in] Clear the depth buffer to this new Z value, which ranges from 0 to 1. For the purposes of this book, pass 1.0f.

DWORD Stencil

[in] Clear the stencil buffer to this new value, which ranges from 0 to 2^n-1 (n is the bit depth of the stencil buffer). For the purposes of this book, pass 0.

Here's an example of how to use the Clear method. Notice that we want to clear the entire back buffer, so we pass NULL for those RECT structures.

```
g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
    D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0);
```

Beginning and Ending a Scene

Once the back buffer has been cleared using the Clear method, a new scene can begin. Beginning a scene occurs when we call the BeginScene method of IDirect3DDevice9. It requires no parameters. Calling this method is like telling the device, “Hey, get the back buffer ready for some drawing.” This method simply prepares the back buffer.

The EndScene method of IDirect3DDevice9 ends a scene. Again, it requires no parameters and it's like saying, “Well, we've finished drawing to the back buffer now.” So this method confirms the back buffer is ready to be flipped and become the new front buffer. Typically, applications will perform all their drawing between the BeginScene and EndScene calls. For now we have nothing to draw, so the scene will be empty.

Once EndScene has been called, the back buffer has not actually flipped and become the front buffer yet. To do this final step we call the **Present** method of IDirect3DDevice9. This method officially signifies the end of a frame. This means whatever we rendered to the back buffer between BeginScene and EndScene will now become visible. The old front buffer then becomes the new back buffer, ready for the next frame. The syntax and parameters for Present appear below.

```
HRESULT Present(
    CONST RECT *pSourceRect,
```

```

    CONST RECT *pDestRect,
    HWND hDestWindowOverride,
    CONST RGNDATA *pDirtyRegion
);

```

CONST RECT *pSourceRect

You can pass a RECT structure if you only want to show a specified rectangle from the back buffer. Normally, you'll want to flip the whole surface; to do this pass NULL.

CONST RECT *pDestRect

For this value you will normally pass NULL. However, you can specify a rectangle on the front buffer into which the contents of the back buffer will be pasted.

HWND hDestWindowOverride

If you want to present a scene inside a different window, you can pass its handle here. Normally you will pass NULL.

CONST RGNDATA *pDirtyRegion

The value must be NULL unless the swap chain was created with D3DSWAPEFFECT_COPY. For the purposes of this book, this value should be NULL.

The entire rendering loop can be coded as follows. This loop should be executed on each frame, during an application's message pump. This will refresh the window and present a scene.

```

// Clear the back buffer and the z buffer
g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
    D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0);

// Begin the scene
if(SUCCEEDED(g_pd3dDevice->BeginScene()))
{
    //Draw stuff here

    // End the scene
    g_pd3dDevice->EndScene();
}

```

```
// Present the back buffer contents to the display  
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
```



Figure 2.4: Blank scene window

Step 6 — Shut Down Direct3D

As your application ends it's your responsibility to free up all the DirectX interfaces you created. This stops memory leaks and keeps your application nice and neat. To delete a Direct3D interface you call its **Release** method. This is something many COM programmers will already be familiar with. The code to free up your `IDirect3D9` and `IDirect3DDevice9` pointers looks like this:

```
g_pD3D->Release();  
pd3dDevice->Release();
```

More on Surfaces

Now it's time to see how bitmaps can be loaded from a file and onto a surface, and then finally rendered to the window by `IDirect3DDevice9`. Surfaces are encapsulated by the `IDirect3DSurface9` interface. To recap, surfaces have the following properties:

- **Surfaces hold images**

A surface is a buffer, a rectangle of bytes in memory. Like a canvas, surfaces are used to store images, or pixel data.

- **Surfaces can be loaded from files or resources**

Surfaces are originally created blank. They just hold blank pixel data. Once a surface has been created, you can then load image data onto it. This can originate from a file on disk or in memory or a resource. DirectX provides functions to achieve these processes, as demonstrated shortly.

- **Surfaces can be copied to and from one another**

The pixel data on a surface can be copied between surfaces. This means you can load an image onto a surface, and then copy all of it onto another surface, or you can copy just a subset of it onto another surface.

Creating Surfaces

You can create surfaces to hold your own image data; however, you cannot create surfaces with image data preloaded onto them. You must first create a blank surface of a specified width and height. You then load image data onto it at a later stage. The created surface will not be rendered immediately; it will simply be used to hold image data in memory. Later, it will be drawn to the window by copying it to the back buffer surface. To create a surface you call `IDirect3DDevice9`'s method **CreateOffscreenPlainSurface**. This function requires you to specify the width and height of the surface to create. Once completed it returns a valid `IDirect3DSurface9`

pointer. The syntax and parameters for `CreateOffscreenPlainSurface` appear below.

```
HRESULT IDirect3DDevice9::CreateOffscreenPlainSurface(
    UINT Width,
    UINT Height,
    D3DFORMAT Format,
    DWORD Pool,
    IDirect3DSurface9** ppSurface,
    HANDLE* pSharedHandle
);
```

UINT *Width*

[in] Width in pixels of the surface to create.

UINT *Height*

[in] Height in pixels of the surface to create.

D3DFORMAT *Format*

[in] Color format of the surface to create.

DWORD *Pool*

[in] A constant indicating in which area of memory to create the surface. You can create the surface in system memory or even the memory on your system's graphics card. Most often this value will be `D3DPOOL_SYSTEMMEM`.

IDirect3DSurface9 *ppSurface***

[out, retval] Address to which a valid `IDirect3DSurface9` pointer is to be returned.

HANDLE* *pSharedHandle*

[in] This is a reserved value. Just pass `NULL`.



Tip. You may be wondering exactly what to pass for the surface width, height, and format values, and how you can expect to know this information in advance. You could potentially be creating all kinds of surfaces to hold different sized images. To solve this issue, DirectX provides a useful function called `D3DXGetImageInfoFromFile`. It accepts a filename and populates a `D3DXIMAGE_INFO` structure with image size and format information. You can then pass these values straight onto the `CreateOffscreenPlainSurface` function to size your surface appropriately. DirectX also provides similar functions to read information from images already in memory and from resources. The declarations for all these functions are as follows:

```
HRESULT WINAPI D3DXGetImageInfoFromFile
(
    LPCSTR pSrcFile,
    D3DXIMAGE_INFO *pSrcInfo
);

HRESULT WINAPI D3DXGetImageInfoFromFile-
    InMemory
(
    LPCVOID pSrcData,
    UINT SrcDataSize,
    D3DXIMAGE_INFO *pSrcInfo
);

HRESULT WINAPI D3DXGetImageInfoFromResource
(
    HMODULE hSrcModule,
    LPCTSTR pSrcFile,
    D3DXIMAGE_INFO *pSrcInfo
);
```

The code to create and size a surface might look like the following example. It creates a surface whose width, height, and format match the information read from an image file on disk. This way, we know in advance that our surface is configured correctly to hold this image. Remember, the image is not actually loaded onto the surface yet. The loading process is described in the next section.


```
//Creates a surface
IDirect3DSurface9 *Surface = NULL;
D3DXIMAGE_INFO Info
D3DXGetImageInfoFromFile(Path, &Info))
g_pd3dDevice->CreateOffscreenPlainSurface(Info.Width, Info.Height,
                                           Info.Format,
                                           D3DPPOOL_SYSTEMMEM,
                                           &Surface, NULL);
```

Loading Images onto Surfaces

Surfaces are created blank, with no images loaded onto them. To load image data onto a surface DirectX provides a number of functions. This book demonstrates how to load an image from a file, which is achieved by calling the **D3DXLoadSurfaceFromFile** function. It's not a method of any specific interface; it's a standalone helper function. Its syntax and parameters follow.

```
HRESULT WINAPI D3DXLoadSurfaceFromFile
(
    LPDIRECT3DSURFACE9 pDestSurface,
    CONST PALETTEENTRY* pDestPalette,
    CONST RECT* pDestRect,
    LPCTSTR pSrcFile,
    CONST RECT* pSrcRect,
    DWORD Filter,
    D3DCOLOR ColorKey,
    D3DXIMAGE_INFO* pSrcInfo
);
```

LPDIRECT3DSURFACE9 pDestSurface

[in] Specifies the Direct3D surface onto which the image will be loaded. Pass your surface here.

CONST PALETTEENTRY* pDestPalette

[in] Pointer to a PALETTEENTRY structure, the destination palette of 256 colors, or NULL. Usually this will be NULL.

CONST RECT* pDestRect

[in] This specifies a rectangle on the surface that receives the image. Your surface could be larger than the image or you may

only want a portion of the surface to be loaded. Pass NULL to use the entire surface.

LPCTSTR *pSrcFile*

[in] Specifies the filename of the image to be loaded. Images can be in the following formats: .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm, and .tga.

CONST RECT* *pSrcRect*

[in] You can pass a rectangle to select a portion of the source image to load. This is useful if you don't want to load the entire image. To load the entire image, pass NULL.

DWORD *Filter*

[in] Using this flag you can tweak how the surface will appear. Most often you will pass D3DX_FILTER_NONE.

D3DCOLOR *ColorKey*

[in] You can pass a hexadecimal color value generated from the D3DXCOLOR_ARGB macro. Using this you can manipulate the alpha channel to make the surface fully visible, fully transparent, or partially transparent. Alpha channels and transparency are covered in detail when we examine textures later in the book. To disable color keying, as we will do in this example, just pass 0. This will load the image normally from the file.

D3DXIMAGE_INFO* *pSrcInfo*

[in, out] You can pass a D3DXIMAGE_INFO structure in here to receive information about the loaded image. Usually you will pass NULL.

*** NOTE.** DirectX also provides other means to load images onto surfaces: D3DXLoadSurfaceFromFileInMemory, D3DXLoadSurfaceFromMemory, D3DXLoadSurfaceFromResource, and D3DXLoadSurfaceFromSurface.

The following code demonstrates how to load an image from a file onto a surface. Notice how simple a process it really is.

```
IDirect3DSurface9 *Surface;
//...
```

```
//Loads a surface from a file
D3DXLoadSurfaceFromFile(Surface, NULL, NULL, Path, NULL,
                        D3DX_FILTER_NONE, 0, NULL);
```

Copying Surfaces

Copying image data from one surface onto another is a simple process. It works much like the copy and paste routine in a normal photo editing program. The source surface is where data is taken from and the destination surface is where data is copied to. You can copy the whole source surface onto the destination surface or you can even select a rectangle on the source surface from which you want to copy. You can also choose the XY location on the destination surface where pasting begins. To copy surfaces you call the **UpdateSurface** method of IDirect3DDevice9. Its syntax and parameters appear below.

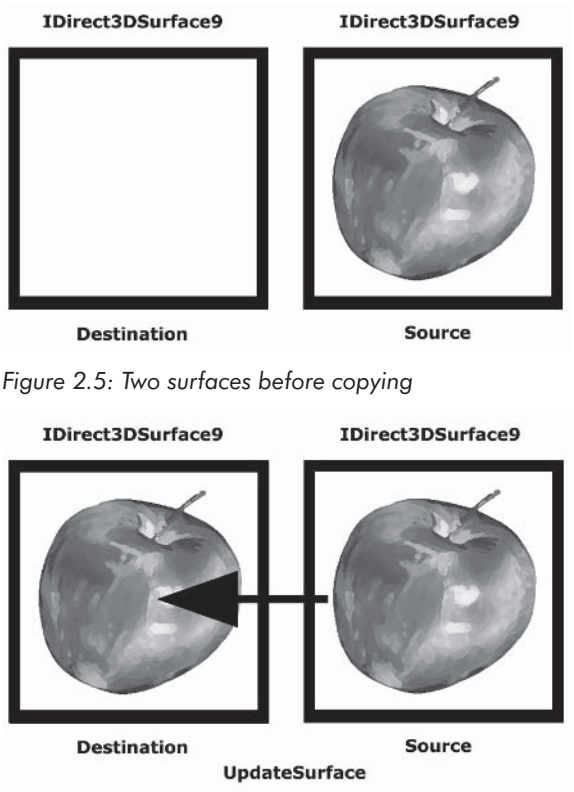


Figure 2.6: Two surfaces after copying

```

HRESULT IDirect3DDevice9::UpdateSurface
(
    IDirect3DSurface9* pSourceSurface,
    CONST RECT* pSourceRect,
    IDirect3DSurface9* pDestinationSurface,
    CONST POINT* pDestinationPoint
);

```

IDirect3DSurface9* pSourceSurface

[in] Pointer to the source surface from which data is to be copied.

CONST RECT* pSourceRect

[in] A rectangle structure selecting a region on the source surface from which data is to be copied. If you want to copy the whole surface, pass NULL.

IDirect3DSurface9* pDestinationSurface

[in] Pointer to the destination surface where data is to be copied.

CONST POINT* pDestinationPoint

[in] A point structure defining an XY location on the destination surface where pasting is to begin. To paste the source surface's pixel data to the top-left corner of the destination surface, just pass NULL.

Take a look at the following code. It shows how a source surface is copied to a destination surface.

```

IDirect3DSurface9 *SourceSurface;
IDirect3DSurface9 *DestSurface;
//...
//Copy a surface
g_pd3dDevice->UpdateSurface(SourceSurface, NULL, DestSurface, NULL);

```

***NOTE.** Here are some important rules to remember when surface copying:

- The source surface must have been created with `D3DPOOL_SYSTEMMEM`.
- The destination surface must have been created with `D3DPOOL_DEFAULT`.
- Neither surface can be locked or holding an outstanding device context.
- Neither surface can be created with multisampling. The only valid flag for both surfaces is `D3DMULTISAMPLE_NONE`.
- The surface format cannot be a depth stencil format.
- The source and dest rects must fit within the surface.
- No stretching or shrinking is allowed (the rects must be the same size).
- The source format must match the dest format.

Presenting Images with the Back Buffer

You now know how to create your own surfaces and load images onto them. That's good, but none of those surfaces are drawn to the window; you never see them on screen. Currently, they're just images held in memory. You've probably guessed already how you might show an image in the window. That's right; you need to copy your surface to the back buffer surface on each frame. Remember, the back buffer is a surface too. It's a working area where an image is composed and then pushed to the front (flipped) where it can be seen in the window. This occurs on every frame.

Before we can actually copy data to the back buffer we'll need its `IDirect3DSurface9` pointer. This is because we'll need to pass it to the `UpdateSurface` function as the destination surface for the copying operation. To retrieve the back buffer we call the **GetBackBuffer** method of `IDirect3DDevice9`. This returns an `IDirect3DSurface9` interface representing the back buffer. Its syntax and parameters follow.

```
HRESULT GetBackBuffer
(
    UINT iSwapChain,
    UINT BackBuffer,
    D3DBACKBUFFER_TYPE Type,
```

```

    IDirect3DSurface9 **ppBackBuffer
);

```

UINT iSwapChain

[in] An unsigned integer specifying the swap chain. For the purposes of this book, pass 0.

UINT BackBuffer

[in] Index of the back buffer object to return. Back buffers are numbered from 0 to the total number of back buffers minus one. A value of 0 returns the first back buffer. For the purposes of this book, pass 0.

D3DBACKBUFFER_TYPE Type

[in] Pass D3DBACKBUFFER_TYPE_MONO.

```

IDirect3DSurface9 **ppBackBuffer

```

[out, retval] Address of a surface to where the back buffer pointer is returned.

The code to get the back buffer and copy a surface onto it might look like the following. Be sure to copy your surface onto the back buffer between the BeginScene and EndScene statements of the rendering loop, which occurs on each frame. This will ensure your data gets copied onto the back buffer correctly and that it'll be visible as the frame is presented.

```

if(SUCCEEDED(g_pd3dDevice->BeginScene()))
{
    //Draw stuff here
    IDirect3DSurface9 *BackBuffer = NULL;
    g_pd3dDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, & BackBuffer);
    g_pd3dDevice->UpdateSurface(SourceSurface, NULL, BackBuffer, NULL);

    // End the scene
    g_pd3dDevice->EndScene();
}

// Present the back buffer contents to the display
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);

```

***NOTE.** You would not usually call `GetBackBuffer` in the rendering loop since you only need to get the back buffer pointer once throughout the application. You'd usually perform this step during application initialization.

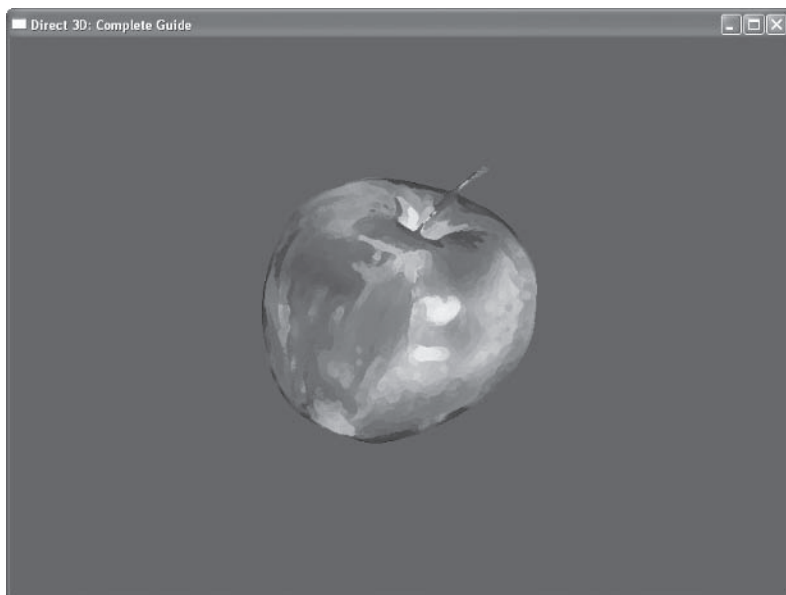


Figure 2.7: Image in window

Lost Devices

The last issue in this chapter to consider is a lost device. There comes a point in many Direct3D applications where things go wrong and you lose control of the Direct3D device and therefore lose control of the hardware. It just doesn't work. You can't present a scene, you can't render, and you can't really do anything. This can happen for many reasons. For example, the user may press the `Alt+Tab` key combination in a full-screen DirectX application. This switches focus from your program to another. This means your application falls into the background and you lose control of the display adapter. The resolution probably changes and other applications might access the 3D hardware. Thus, whatever surfaces or resources you created with the 3D device and stored on

the graphics card are considered invalid and overwritten. In short, the Direct3D device has been reset.

During this time most Direct3D functions will give a silent failure, meaning they will return successful even though they didn't actually do anything productive. The only thing your application can do is sit and wait until focus is restored and the device can be controlled again. As you regain control, you will need to set it up again. This means you'll need to free up whatever resources you created before — like surfaces and textures and 3D models — and then you'll need to create them again. To detect and handle lost devices you generally perform the following steps on every frame:

1. Test the cooperative level of the Direct3D device. This is basically a check to see if the device is with us or not. To check this status you can call the `TestCooperativeLevel` method of `IDirect3DDevice9`. It requires no parameters. If the function succeeds, then the device is operational and the application can continue. If it returns `D3DERR_DEVICELOST`, then the device has been lost and is still being used by another process. Finally, if it returns `D3DERR_DEVICENOTRESET`, this means the device was lost and is now ready to be reset and set up again by your application; in other words, your program has focus again.
2. If step 1 returns `D3DERR_DEVICELOST`, then do nothing; just sit and wait because the device is currently unavailable. If it returns `D3DERR_DEVICENOTRESET`, then go to step 3.
3. Here, control of the device has been regained and is ready to set up again. If you're running a windowed DirectX application, then you need to free up all resources — surfaces, textures, models, etc. — that were created in the graphics card's memory. If your application is full screen, you should free all resources you created. Go to step 4.
4. If you're running a windowed application, you then call the `Reset` method of `IDirect3DDevice9` to restore the device to an operational state. It requires one parameter, the `D3DPRESENT_PARAMETERS` structure you used to create

the device. If your application is full screen, you should release the device and create a new device. Move on to step 5.

5. Create your resources again.

Here is a sample render function that occurs on every frame. It tests and handles a lost device.

```
HRESULT Result = g_pd3dDevice->TestCooperativeLevel();

If(FAILED(Result))
{
    if(Result == D3DERR_DEVICELOST)
        return; //Do nothing

    if(Result == D3DERR_DEVICENOTRESET)
    {
        DeleteMyResources();
        g_pd3dDevice->Reset(&Params);
        InitMyResources();
    }
}

if(SUCCEEDED(g_pd3dDevice->BeginScene()))
{
    //Draw stuff here

    // End the scene
    g_pd3dDevice->EndScene();
}

// Present the back buffer contents to the display
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
```

Conclusion

Hopefully this chapter has set you well on your way and has given you a good idea about the general structure of a Direct3D application. It has demonstrated how to set up Direct3D and associate a device with a window. It has also shown how the game loop works and how this is used to preset a scene. Furthermore, it has shown how to draw images to the main window via surfaces and illustrated how you should handle lost devices.

Soon we examine how Direct3D can be used to render 3D geometry. I'm talking triangles, polygons, 3D meshes, and 3D worlds. First, however, you'll need to understand 3D mathematics and some general principles. The next chapter provides a crash course. Stick with me. It's not that difficult.

This page intentionally left blank.



Chapter 3

3D Mathematics

Ah, mathematics, the language of numbers — something people seem to either love or hate. That's what this chapter's all about. Don't worry though; it's not too difficult. The reason this chapter is important is because you'll need to understand 3D mathematics before you can go on to create fantastic games in 3D, or 2D for that matter. For example, you'll want to know how to position things in 3D space using coordinate systems and you'll probably want to rotate things by specified angles. No point in trying to do that if we don't know the theory behind it. This chapter will attempt to cover everything you'll need to know. Specifically, it aims to confront the following issues:

- Dimensions
- Coordinate systems
- Points
- Primitives
- Translation, rotation, and scaling
- Vectors
- Matrices
- Planes

*** NOTE.** Many of the theories and mathematical ideas I present are applied in the context of games. It may not be the most accurate definition mathematically, but it works for games. And, for this book and your work in Direct3D, that's what matters.

Coordinate Systems

The best way to start thinking about 3D mathematics is to use a practical example. Since probably the birth of mankind, humans have needed to measure distances to get an idea of how far something is from them or how far something has moved. We needed to know how far the nearest river was or how far the nearest fruit trees stood so we could estimate how much time it'd take to get there and back home again. So throughout the ages humans have charted the lands and developed various numerical systems to measure and understand distance. This means if I stumbled across a tribe living in the jungle and they educated me in the ways of their numerical system, I would mathematically be able to express a position on the land and they'd be able to find it.

This kind of system used to measure and define space is called a *coordinate system*. Now, it might sound rather obvious to say this, but all coordinate systems are *relative*. This means distances are all measured from a common reference point. For example, if you asked me where the bus stop was and I told you it was 50km north you'd probably give me a rather funny look. Did I mean 50km north from where you and I are standing? Or did I even mean 50km north from a crater on the moon? The problem is 50km north on its own has no real value because I've not specified a reference point from which to begin traveling. Thus, every coordinate system has a reference point — a center — from which distances are measured. This point is called the *origin*.

One-Dimensional (1D) Coordinate Systems

The best way to start learning about coordinate systems is with the simplest kind, a 1D system. If you could only walk forward and backward in a straight line, you'd be traveling in a 1D coordinate system. The point you're standing at in the beginning will be the origin. A good way to visualize a 1D system is to think of the standard number line (see Figure 3.1). The origin, as always, is at 0. Points to the right of the origin are positive integers (numbers with



Figure 3.1: A number line

no decimal place) such as 1, 2, 3, 4, 5, 6, etc. Numbers to the left of the origin are negative integers such as -1 , -2 , -3 , etc. So if I told you I was standing at point 5 you'd know exactly where I'd be.

Two-Dimensional (2D) Coordinate Systems

A standard XY graph is a classic example of a 2D coordinate system. Here, you can move up, down, left, and right. This kind of system is called a *Cartesian* coordinate system. It's basically two 1D number lines crossing one another. Each number line is called an *axis* (plural: *axes*). One of them is running horizontally, from left to right (like in a 1D system); this axis is called X. The other axis is running vertically from top to bottom and is called Y. The axes cross (intersect) in the middle, and the point where they intersect is the origin, 0. Take a look at Figure 3.2 for an example. To specify a point (also called a *vertex*, plural: *vertices*) on this coordinate system we need two pieces of information: an X value and a Y value. So if I were standing at $(5,3)$, meaning $X=5$ and $Y=3$, you'd be able to measure this from the origin and find exactly where I was. Simple.

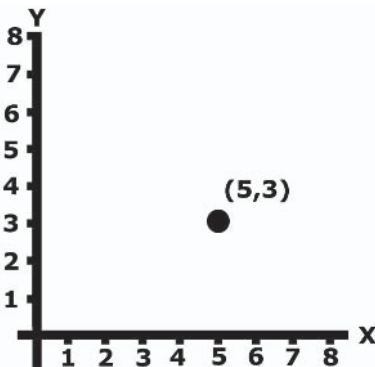


Figure 3.2: Cartesian coordinate system

***NOTE.** Direct3D provides a good structure to represent 2D vertices and vectors (explained later). This structure is **D3DXVECTOR2**. You can define a 2D vertex like this:

```
D3DXVECTOR2 Vertex;  
Vertex.x = 5.0f;  
Vertex.y = 3.0f;
```

Or, more simply:

```
D3DXVECTOR2 Vertex(5.0f,3.0f);
```

More on 2D Coordinate Systems

Before moving onto 3D, it's worth taking a look at some of the things we can already represent using a 2D coordinate system. We can express more than simply vertices. We can draw some pretty pictures too. We can represent lines, triangles, and many other shapes. Take a look.

■ Lines

Mathematically, lines are really just two vertices — a start vertex and an end vertex. The farther apart the vertices, the longer the line.

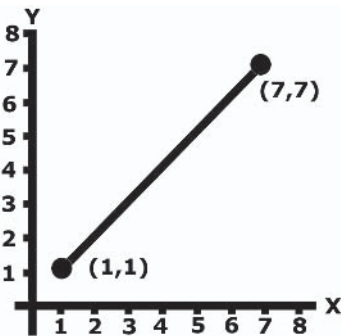


Figure 3.3: Line

■ Line strips

A line strip can be thought of as an array of joining lines; that is, a sequence of lines where the end vertex of one line becomes the starting vertex for the next line.

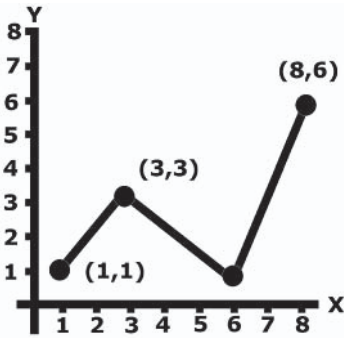


Figure 3.4: Line strip

■ Triangles

The triangle is a basic and very important primitive. Basically, it's three vertices — one vertex for each corner of the triangle. Using a combination of triangles it's possible to create more complex shapes.

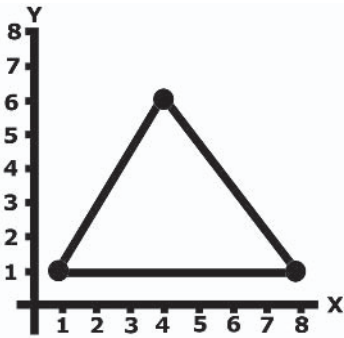


Figure 3.5: Triangle

■ Triangle strips

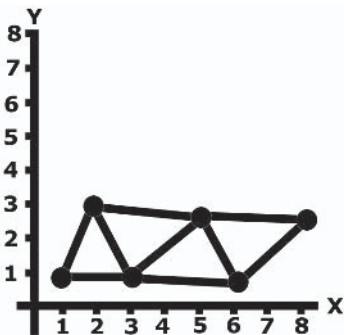


Figure 3.6: Triangle strip

Like line strips, triangle strips are sequences of connected triangles. The triangles are joined at the edges. From this you can form all kinds of shapes, like squares.

■ Triangle fans

Triangle fans are sequences of connected triangles, like triangle strips, except in triangle fans all the triangles share one corner — the same vertex.

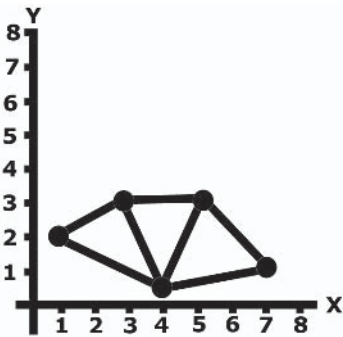


Figure 3.7: Triangle fan

Three-Dimensional (3D) Coordinate Systems

Once you've got the 2D coordinate system in your mind it's not too difficult to extend that to 3D. 3D is being able to move forward and backward, left and right, and up and down. 3D is composed of three axes, two of which come from the 2D system. You have X, which, like before, runs from left to right, and you still have Y, which runs up and down. You now have a third axis though, called Z. This runs into the distance, coming from behind you and disappearing ahead of you.

Actually there are two types of 3D coordinate systems, although we only need to worry about one of them. There are left-handed and right-handed coordinate systems. The only difference between them is the values of the Z axis. In a left-handed coordinate system (the one we will use), Z extends positively into the distance. In a right-handed system, the Z axis extends negatively into the distance. Take a look at Figures 3.8 and 3.9 to see what I'm talking about.

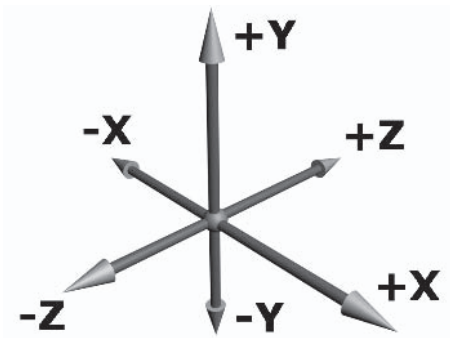


Figure 3.8: Left-handed system

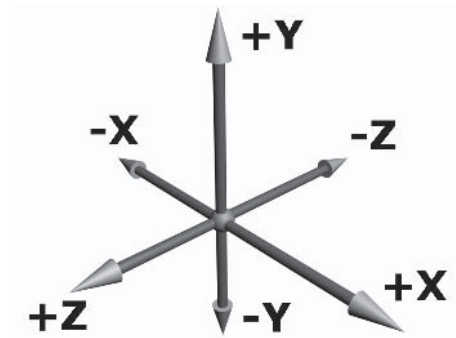


Figure 3.9: Right-handed system

Not surprisingly, vertices in a 3D coordinate system are specified like 2D systems, except for an extra Z component, like this: (5,3,7). You'll also notice that nearly everything that works in a 2D system will apply to a 3D one too. You can still represent lines, triangles, rectangles, etc.; you just need to include that extra Z component. Direct3D provides a structure to represent 3D vertices called `D3DXVECTOR3`. You can use it in just the same way as a 2D vertex; just be sure to specify Z too.

Geometric Transformations

In the previous sections we examined coordinate systems and how we can plot vertices. Furthermore, you saw how to use a collection of vertices to build shapes like triangles and squares, generally called *primitives*. That's fine; however, you'll still want to know how you can make those primitives do things like move around the screen. You don't want them to just sit there doing nothing. Specifically, you'll want to know how you can move them around, rotate them, and scale them.

Translation

The simplest form of transformation is a *translation*. This means you physically move your vertices from one place to another. Therefore, if I were standing at (5,5) and I moved to (10,7), I have translated my position by 5 units on the X axis and 2 units on the Y axis.

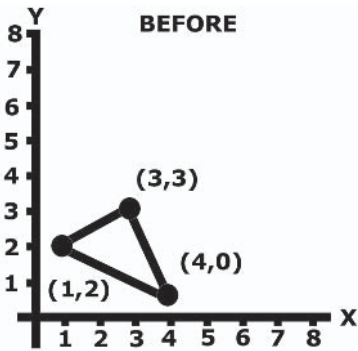


Figure 3.10: Before translation

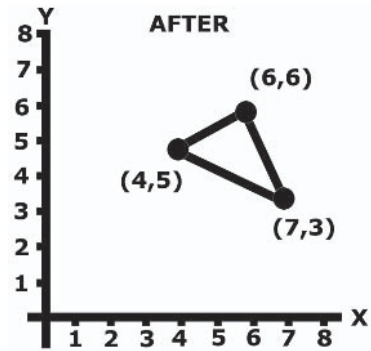


Figure 3.11: After translation

Calculating a translation mathematically is simple. Just add the amount you want to move onto your existing position, like this:

```
D3DXVECTOR2 Position(0, 0);
Position.x += AmountToMove.x;
Position.y += AmountToMove.y;
```

*** NOTE.** You can also use the Direct3D function `D3DXVec3Add` to add two vertices or vectors.

Rotation

Rotation is about turning something a specified number of degrees about the origin. Rotation always occurs around the origin. If you're rotating a triangle or some other shape, notice what actually happens: It's not the shape itself that is rotating, but the vertices that make up its corners. It's simply the vertices that move and, thus, the shape rotates. Take a look at Figure 3.12.

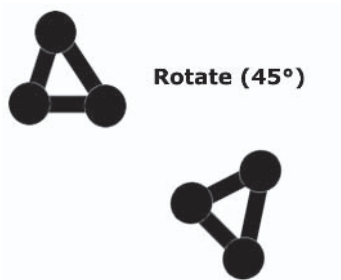


Figure 3.12: Rotation

Rotating vertices by a specified angle requires some knowledge of trigonometry, which is not covered in this book. Don't worry, though, I'll show you the formula and an example. Fortunately, there are a number of other ways to rotate objects in Direct3D that do not require you to learn the mathematics. These are covered later as we consider matrices. The formula to rotate a vertex about the origin is:

$$\begin{aligned}x &= x * \cos(\text{Angle}) - y * \sin(\text{Angle}); \\y &= x * \sin(\text{Angle}) + y * \cos(\text{Angle});\end{aligned}$$

For example, if I wanted to rotate a vertex at (5,5) by 45 degrees, the code to achieve this would be:

```
D3DXVECTOR2 Vertex(5.0f, 5.0f);
FLOAT Angle = 45.0f;

Vertex.x = Vertex.x * cos(Angle) - Vertex.y * sin(Angle);
Vertex.y = Vertex.x * sin(Angle) + Vertex.y * cos(Angle);
```

*** NOTE.** Generally in Direct3D, angles are specified in a measurement called *radians*, not degrees. This is explained later.

Scaling

Scaling means to make something smaller or larger. If I wanted to make a triangle half its current size, for example, then I would need to scale it. In reality, all that happens when you scale is that you move a primitive's vertices closer together for shrinking and farther apart for enlarging. Simple.

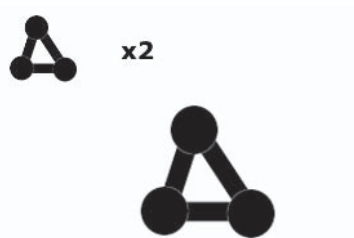


Figure 3.13: Scaling

To scale a vertex all you need to do is multiply it by a *scaling factor*. A scaling factor is a *scalar* number (an ordinary number) that tells us by how much you'd like to scale. This number can range from 0 to infinity. 1 means to keep the original scale, 0 means to shrink it so much that it becomes invisible, 0.5 shrinks it to half its size, 2 increases the size by double, and so on.

*** NOTE.** You can even check out this scaling idea with ordinary numbers. Let's take the number 2.

Now, if I want to scale 2 by half (in other words, make it half its size) I multiply it by 0.5. Like the following:

$$2 * 0.5 = 1. \text{ Correct.}$$

Let's try one more. If we wanted to make 2 larger by six times, then we'd scale it by a factor of 6.

$$2 * 6 = 12. \text{ Correct.}$$

The formula to scale a vertex looks like this:

$$x = x * \text{ScalingFactor}$$

$$y = y * \text{ScalingFactor}$$

Some sample code to do this would look like this:

```
D3DXVECTOR2 Vertex(5.0f, 5.0f);  
FLOAT ScalingFactor = 0.5;  
  
Vertex.x = Vertex.x * ScalingFactor;  
Vertex.y = Vertex.y * ScalingFactor;
```

*** NOTE.** Direct3D provides a function to scale vertices and vectors called `D3DXVEC2Scale`. There is also a version available for 3D vertices called `D3DXVEC3Scale`.

Vectors

Previous sections have explained 1D, 2D, and 3D coordinate systems. We also elaborated on how to plot vertices, create primitives, and apply basic transformations to them. These transformations are translation, rotation, and scaling. It's now time to consider the concept of direction and movement more closely.

We now know how to move a primitive to a specified point by translating it; for example, moving a triangle at (5,5) to (10,10). This is all well and good but it requires that we know in advance the location to which we wish to translate. Instead, what if we were moving a monster in a game? The monster may be traveling in some direction and it may take him five minutes to reach his destination. During that time he would be moving closer and closer to his destination, traveling at a certain speed. Currently, our mechanism of specifying absolute positions (points in our coordinate space measured from the origin) doesn't seem too convenient. It means we have to know every move he makes and keep track of every point he walks on. Instead, wouldn't it be nice if we could just keep track of the creature's direction and be able to move him along it according to his speed? Indeed. This section attempts to address this issue by using vectors.

Vectors look just like vertices. They can be 2D or 3D, just like vertices, and they have the standard XYZ structure too. The difference, though, is that vectors represent direction and tell you how far along a direction to travel. Vertices just specify a position and nothing more. For example, imagine I'm standing at (5,3) and was given a *vector* of (2,3). This vector is not telling me to move to the point (2,3) but it's telling me to move 2 units along the X axis and 3 along Y from where I'm currently standing. This will take me to vertex (7,6). Thus, vectors represent direction. Take a look at Figure 3.14 to see a vector.

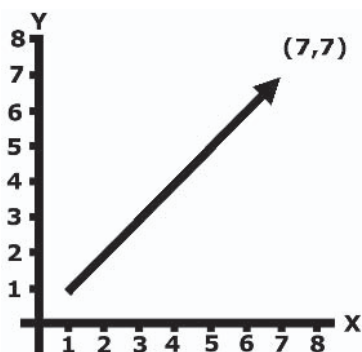


Figure 3.14: Vectors

*** NOTE.** Like vertices, vectors can be represented in the `D3DXVECTOR2` and `D3DXVECTOR3` structures.

*** NOTE.** Vectors can be negative, so $(-2, -3)$ means you'll move left by two and down by 3. The vector $(2, 3)$ means you'll move right by 2 and up by 3.

TIP. Vectors are very useful for representing paths your game characters must travel.

As you will see, game characters won't usually travel vectors the long way around. Instead, they'll walk the direct route: across the diagonal.

Length of the Diagonal (Magnitude)

The distance along a vector's diagonal (its length) is called a vector's *magnitude*. There's a special kind of notation mathematicians use to represent vector magnitude. It looks like two vertical bars placed on either side of the vector name. So if we have a vector called V , its magnitude can be expressed as $|V|$.

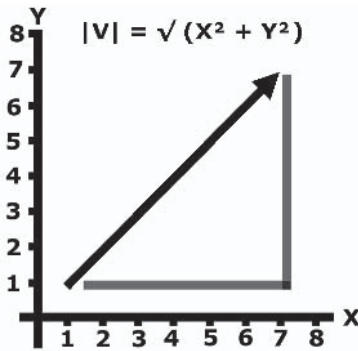


Figure 3.15: Magnitude

Working out the length of this diagonal might remind you of your days in school when you learned how to solve the diagonal in a right-angle triangle using the Pythagorean theorem. Don't worry if you can't remember it though; the formula is provided here and Direct3D provides a function to solve it for you too. To work out the length of the diagonal you simply add together the square of each component in the vector, XYZ, and then take the square root of the total. The formula follows.

$$| \text{Vector} | = \text{sqrt}((\text{Vector.x} * \text{Vector.x}) + (\text{Vector.y} * \text{Vector.y}))$$

Direct3D provides two functions to calculate vector length, one for 2D vectors and one for 3D. These are **D3DXVec2Length** and **D3DXVec3Length** respectively. The D3DXVec3Length function syntax and parameter are:

```
FLOAT D3DXVec3Length
(
    CONST D3DXVECTOR3 *pV
);
```

CONST D3DXVECTOR3 *pV

 [in] Pointer to the source D3DXVECTOR3 structure.

Vector Addition

If you've got one or more vectors that make a connected path, perhaps some vectors zigzagging this way and that, you can add them all up to make a shorter and more direct path between your source and final destination.

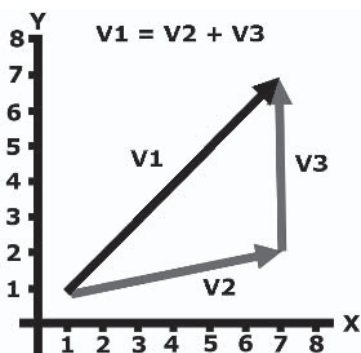


Figure 3.16: Addition

Adding two vectors is a simple process. You just go through each of their components, XYZ, and add each one to the corresponding component in the other vector. You can do this manually or you can use the Direct3D functions **D3DXVec2Add** and **D3DXVec3Add** to do it for you. The syntax and parameters appear below, along with some sample code to add two 3D vectors.

```
D3DXVECTOR3 *D3DXVec3Add
(
    D3DXVECTOR3 *pOut,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2
);
```

D3DXVECTOR3 *pOut

[in, out] Pointer to the D3DXVECTOR3 structure that is the result of the operation.

CONST D3DXVECTOR3 *pV1

[in] Pointer to a source D3DXVECTOR3 structure.

CONST D3DXVECTOR3 *pV2

[in] Pointer to a source D3DXVECTOR3 structure.

Code sample:

```
D3DXVECTOR3 Vec1(0,0,0);
D3DXVECTOR3 Vec2(5,5,5);
D3DXVECTOR3 Result(0,0,0);

//Result (5,5,5)
D3DXVec3Add(&Result, &Vec1, &Vec2);
```

Vector Subtraction

Simply put, subtracting two vectors gives you a vector between their end points. In other words, subtracting two vertices (point X and point Y) gives you the vector between them. This means you can add the result onto point X and the result will take you to point Y. So if I were standing at (5,5) and wanted to reach point (10,10), I could subtract my destination from (5,5) and it gives me the route I need to travel.

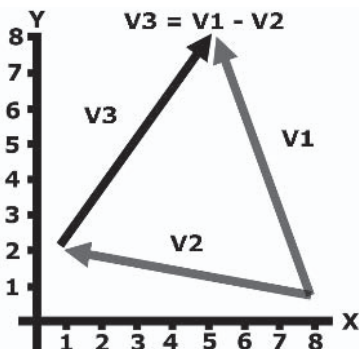


Figure 3.17: Subtraction

The order in which vectors are subtracted is important too. Just like in normal subtraction, subtracting a larger vector from a smaller one will result in a vector with a negative direction. Mathematically, subtracting vectors is a similar process to addition. You just work through all the destination vector's XYZ components and subtract them from the source vector's. Again, you can perform this process manually or you can use Direct3D's **D3DXVec2-Subtract** and **D3DXVec3Subtract** functions to do it for you. The code to subtract two vectors follows.

```
D3DXVECTOR3 Vec1(5,5,5);
D3DXVECTOR3 Vec2(1,3,5);
D3DXVECTOR3 Result(0,0,0);

//Result (4,2,0)
D3DXVec3Subtract(&Result, &Vec1, &Vec2);
```

Vector Multiplication

Multiplying a vector by a scalar means to scale the vector, just like multiplying a vertex or a number achieves scaling. By scaling a vector you can make it larger or smaller. To make a vector twice as long you multiply it by 2. To make a vector the same size (rather pointless) you multiply it by 1. To make a vector half the size you multiply it by 0.5. You can also negate a vector's direction too and make it point in the opposite direction. To do this, you multiply by a negative value. So, to make a vector twice as long and point in the opposite direction you multiply by -2 .

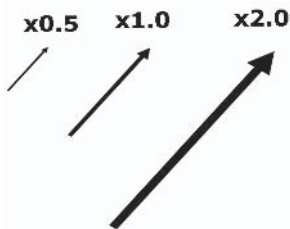


Figure 3.18: Multiplication

Vector Normalization

Mathematically, vector normalization means that a vector's magnitude is reduced to 1. It still points in the same direction, but the length of its diagonal is 1. A vector whose magnitude is 1 can be said to have been normalized; this vector is known as a *unit vector*. You may be wondering exactly why anyone would want to do this. First, there are a number of Direct3D functions that require normalized vectors as their arguments. But also, unit vectors are a smart way to simply represent direction, direction without distance. You can normalize a vector using the Direct3D function `D3DXVec3Normalize`.

Vector Dot Product

The dot product is also known as the scalar product. Essentially, the dot product is a nice little formula that takes two vectors and returns a scalar number. This number tells us some useful information about the angle between two vectors. It can also be used to actually find the angle between two vectors. The notation used to represent a dot product looks like a dot between two vectors. So if we have two vectors, A and B, the dot product looks like this: $(A \cdot B)$. Don't worry too much about the actual theory behind the dot product. The formula to compute it is shown below, and I also explain how you can use the resulting scalar to find out stuff about the angle. Direct3D also provides a function to work out the dot product of two vectors for you. The formula for the dot product is shown in the following code fragment:

```
D3DXVECTOR3 VectorA(1.0f, 1.0f, 1.0f);
D3DXVECTOR3 VectorB(2.0f, 5.0f, 7.0f);

FLOAT DotProduct = (VectorA.x * VectorB.x) + (VectorA.y * VectorB.y) +
                   (VectorA.z * VectorB.z)
```

Direct3D provides two functions to calculate the dot product, one for 2D vectors and one for 3D vectors. These are **D3DXVec2Dot**

and **D3DXVec3Dot** respectively. The syntax and parameters follow.

```
FLOAT D3DXVec3Dot
(
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2
);
```

```
CONST D3DXVECTOR3 *pV1
```

[in] Address of vector A.

```
CONST D3DXVECTOR3 *pV2
```

[in] Address of vector B.

The scalar returned by the dot product tells us about the angle between two vectors. Using this number we can tell the following:

- If the dot product is a negative value, the angle between the two vectors is *acute* (less than 90°).
- If the dot product is greater than 0, the angle between the two vectors is *obtuse* (greater than 90°).
- If the dot product is equal to 0, the angle between the two vectors is *perpendicular* (equal to 90°).

If you just want to find the angle between two vectors you can use the dot product like this:

```
FLOAT Angle = acos(D3DXVec3Dot(&Vec1, &Vec2) / (D3DXVec3Length(&Vec1) *
                                                    D3DXVec3Length(&Vec2)));
```

Vector Cross Product

The vector cross product is also known as the vector product. The cross product takes two vectors and returns a vector that is perpendicular. This means the angle between the new vector and the other two is 90°. Note the cross product is not commutative. This means it matters which way you cross the vectors. The cross

product of Vector1 and Vector2 does not give the same answer as the cross product of Vector2 and Vector1, just like $5 - 3$ is not the same as $3 - 5$. The notation for cross product looks like a giant X. So the cross product of vectors A and B is like this: $(A \times B)$. The formula for the cross product appears in the code fragment below.

```
D3DXVECTOR3 v;
D3DXVECTOR3 pV1;
D3DXVECTOR3 pV2;

v.x = pV1.y * pV2.z - pV1.z * pV2.y;
v.y = pV1.z * pV2.x - pV1.x * pV2.z;
v.z = pV1.x * pV2.y - pV1.y * pV2.x;
```

Direct3D provides a function to compute the cross product of two vectors called **D3DXVec3Cross**. Its syntax and parameters follow.

```
D3DXVECTOR3 *D3DXVec3Cross
(
    D3DXVECTOR3 *pOut,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2
);
```

D3DXVECTOR3 *pOut

[in, out] Pointer to the D3DXVECTOR3 structure that is the result of the operation.

CONST D3DXVECTOR3 *pV1

[in] Pointer to a source D3DXVECTOR3 structure.

CONST D3DXVECTOR3 *pV2

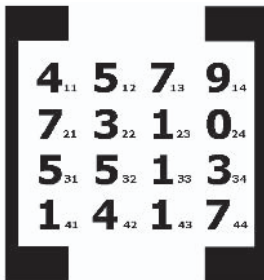
[in] Pointer to a source D3DXVECTOR3 structure.

Matrices

We've seen how vectors represent directions and how transformations like translation and rotation can be applied to vertices. It's time to move on from transformations and vectors now. This section examines a mathematical structure called a *matrix* (plural: *matrices*). By now you've probably guessed that in real games lots of transformations will occur as polygons move about. Some of those transformations will be translations, others will be rotations, and some may be scaling.

Take a space combat simulator where spaceships have a showdown to blast one another from the skies. Here, you must keep track of every spaceship, including its position, orientation, and size. If a craft needs to move, then you'll have to cycle through every vertex in the 3D model and update its position using the translation techniques previously shown. Then if it rotates you'll need to turn every vertex too. This can get tiresome as you can imagine. Now, wouldn't it be great if we could build a structure that encodes a complete combination of transformations — translation, rotation, and scaling — all in one? Then, we could say to all our vertices, "Hey, we have a structure here telling you every transformation to go through. Now go and do it." For the purposes of games, matrices can be used for exactly this purpose.

Simply put, a matrix is like a collection of vectors; a whole grid of numbers inside a box. This represents a set of instructions telling Direct3D how to do something or how to transform something. Matrices can be all kinds of dimensions, for example, 3x3, 4x4, etc. Here's what a matrix looks like:



4 ₁₁	5 ₁₂	7 ₁₃	9 ₁₄
7 ₂₁	3 ₂₂	1 ₂₃	0 ₂₄
5 ₃₁	5 ₃₂	1 ₃₃	3 ₃₄
1 ₄₁	4 ₄₂	1 ₄₃	7 ₄₄

Figure 3.19: Matrix

Direct3D represents a matrix using a D3DXMATRIX structure. For 3D transformations Direct3D will use a 4x4 matrix, and for 2D transformations it will use a 3x3 matrix. Before examining how matrices are applied in transformations, it's a good idea to see how they work mathematically.

***NOTE.** Matrices are very important for 3D calculations, and Direct3D makes heavy use of them.

Matrix Components

Direct3D represents a matrix by using the D3DXMATRIX structure. Each number in a matrix is referred to as a *component*. So, a 3x3 matrix has 9 components and a 4x4 matrix has 16. Since these components are arranged in columns and rows we can reference each component by its grid index; for example, (1,1) or (3,3). To access the components of a matrix structure in Direct3D you use the following member names:

```
_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44
```

The code to declare a matrix and set some of its components to arbitrary values could look like this:

```
D3DXMATRIX Matrix;
Matrix._11 = 5.0f;
Matrix._31 = 7.0f;
Matrix._43 = 9.0f;
```


Matrix Addition

Adding two matrices together is really simple. It's just like adding two vectors together. You simply go through each component in the source matrix and add it to the corresponding component in the destination matrix. It's important to remember you can only add two matrices of the same dimension. So you can add two 4x4 matrices, but not one 4x4 matrix and one 3x3 matrix. The code to add two matrices looks like this:

```
D3DXMATRIX Result;  
D3DXMATRIX Matrix1;  
D3DXMATRIX Matrix2;  
  
Result = Matrix1 + Matrix2;
```

Matrix Subtraction

Again, like vector subtraction, you simply subtract all the components of one matrix from the respective components of another. Both matrices must be the same dimension. Some code to do this appears below:

```
D3DXMATRIX Result;  
D3DXMATRIX Matrix1;  
D3DXMATRIX Matrix2;  
  
Result = Matrix1 - Matrix2;
```

Matrix Multiplication (Scalar)

A single matrix can be multiplied by a scalar. Like addition, you simply cycle through each component of the matrix and multiply it by the specified number. Some code to do this follows:

```
D3DXMATRIX Result;  
D3DXMATRIX Matrix1;  
FLOAT Scalar = 1.0f;  
  
Result = Matrix1 * Scalar;
```

Matrix by Matrix Multiplication

Matrix by matrix multiplication is often referred to as *matrix concatenation*, for reasons I explain later. A matrix can be multiplied by another matrix. The exact process to do this is a little lengthy, and considering Direct3D provides a function to do it for you, there's no real need to trouble ourselves with this here. The function to multiply two matrices together is **D3DXMatrixMultiply**. It requires three arguments: the output matrix where the result is returned and two matrices to multiply. Its syntax and parameters follow and then a code sample is shown.

```
D3DXMATRIX *WINAPI D3DXMatrixMultiply
(
    D3DXMATRIX *pOut,
    CONST D3DXMATRIX *pM1,
    CONST D3DXMATRIX *pM2
);
```

D3DXMATRIX *pOut

[in, out] Pointer to the D3DXMATRIX structure that is the result of the operation.

CONST D3DXMATRIX *pM1

[in] Pointer to a source D3DXMATRIX structure.

CONST D3DXMATRIX *pM2

[in] Pointer to a source D3DXMATRIX structure.

Code sample:

```
D3DXMATRIX Result;
D3DXMATRIX Matrix1;
D3DXMATRIX Matrix2;

D3DXMatrixMultiply(&Result, &Matrix1, &Matrix2);
```

Identity Matrix

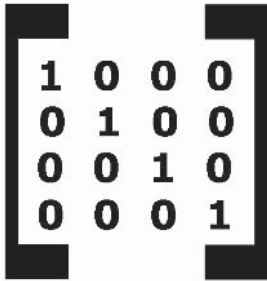


Figure 3.20: Identity matrix

The above matrix is called the *identity matrix*. This matrix is to matrices what the number 1 is to scalars. That's right; **in the world of matrices the identity matrix is like the number 1. It's like a matrix variable that has been initialized.** If you were to multiply another matrix by the identity matrix, you'd get the same matrix back with no changes. If you were to transform a bunch of polygons by the identity matrix, it'd be the same as not transforming the polygons at all; they'd just stay where they were. This doesn't seem very useful, but it's important to remember that when you create a matrix structure in Direct3D it's not initialized. Before you do any work with matrices, make sure they are initialized first as an identity matrix. The function to make a matrix an identity matrix is called **D3DXMatrixIdentity**. You can also check to see whether a matrix is an identity matrix by calling the **D3DXMatrixIsIdentity** function.

The syntax and parameter for **D3DXMatrixIdentity** follow.

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

D3DXMATRIX *pOut

[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Code sample:

```
D3DXMATRIX Result;
D3DXMatrixIdentity(&Result);

If(D3DXMatrixIsIdentity(&Result))
    MessageBox(NULL, "Is Identity Matrix", "", MB_OK);
```

Inverse Matrix

The inverse of a matrix is like the reverse of a matrix. It's like 1 and -1 . If you were to take the inverse of a matrix and multiply it by the original matrix, you'd end up with the identity matrix. Some matrices can be inverted and some can't. Don't worry too much about the inverse of a matrix for now; just be aware it exists.

Direct3D provides the **D3DXMatrixInverse** function to compute the inverse of a matrix. If the matrix cannot be inverted, then this function returns NULL.

The function and parameter table for **D3DXMatrixInverse** can be seen below. Some sample code follows.

```
D3DXMATRIX *WINAPI D3DXMatrixInverse
(
    D3DXMATRIX *pOut,
    FLOAT *pDeterminant,
    CONST D3DXMATRIX *pM
);
```

D3DXMATRIX *pOut

[in, out] Pointer to the D3DXMATRIX structure that is the result of the operation.

FLOAT *pDeterminant

[in, out] Set this parameter to NULL.

CONST D3DXMATRIX *pM

[in] Pointer to the source D3DXMATRIX structure.

Code sample:

```
D3DXMATRIX Result;
D3DXMATRIX Matrix1;

D3DXMatrixInverse(&Result, NULL, &Matrix1);
```

Matrices for Geometric Transformations

Let's put all the matrix mathematics stuff to the back of our minds for the moment. Earlier, we mentioned how a game uses matrices to apply transformations to its objects, such as spaceships, polygons, models, etc. We previously saw how to transform points, vertices, and vectors using standard mathematics. Now we'll examine how matrices can be used to achieve the same thing. Matrices also have the nice advantage of being able to accumulate a whole series of single transformations and group them all into one matrix.

Matrix Translation

You already know translation is about moving stuff from one place to another. Matrices can be used to encode a translation. For example, we can code a matrix to translate something by 5 units on the X axis, 3 units on the Y axis, and 2 units on the Z axis. Then we can pick one or more objects, say a bunch of vertices, and transform them by our matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ X & Y & Z & 1 \end{bmatrix}$$

Figure 3.21: Translation

A translation matrix looks like the one in Figure 3.21. Where it says X, Y, and Z, you can just fill those in with your own coordinates. To make things simpler, Direct3D provides a function to create a translation matrix for you. This function is called **D3DXMatrixTranslation**. It simply requires a pointer to a resultant matrix and the coordinates for the translation.

The syntax and parameters for D3DXMatrixTranslation can be seen below. An example follows.

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

D3DXMATRIX *pOut

[in, out] Pointer to the D3DXMATRIX structure that is the result of the operation.

FLOAT x

[in] X-coordinate offset.

FLOAT y

[in] Y-coordinate offset.

FLOAT z

[in] Z-coordinate offset.

Code sample:

```
D3DXMATRIX Result;
D3DXVECTOR3 Vector1(5.0f, 7.0f, 9.0f);

D3DXMatrixTranslation(&Result, Vector1.x, Vector1.y, Vector1.z);
```

*** NOTE.** Later in this book we shall see how matrix encoded transformations are actually applied to 3D geometry.

Matrix Rotation

Matrices can encode rotation. This means they can rotate your 3D game geometry. Rotation in 3D can occur about any of the three axes (X, Y, or Z). Turn your head left and right; that’s a rotation around the Y axis. Raise and lower your head up and down; that’s a rotation about the X axis. Finally, roll your head from side to side; that’s a rotation about the Z axis. The three separate rotation matrices for each axis appear below. Where I’ve written “Angle” it means you can just put your angle (in degrees) in there.

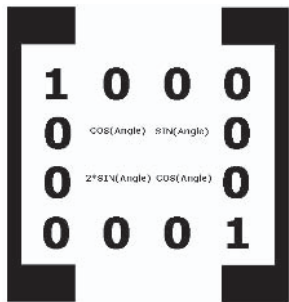


Figure 3.22: RotationX

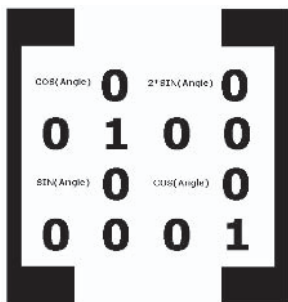


Figure 3.23: RotationY

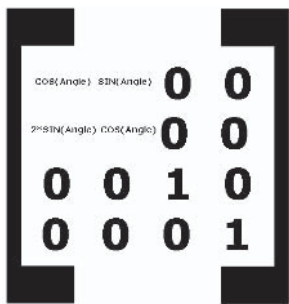


Figure 3.24: RotationZ

Direct3D also provides three separate functions to build the rotation matrices for you. First, however, it is important to understand that many of the Direct3D functions represent angles differently. It may come as a surprise to you that they don’t measure angles in degrees. Instead, angles are specified in radians. The degree to radian conversion diagram in Figure 3.25 demonstrates how to

convert between the systems. Direct3D also provides two macros to convert back and forth between radians and degrees.

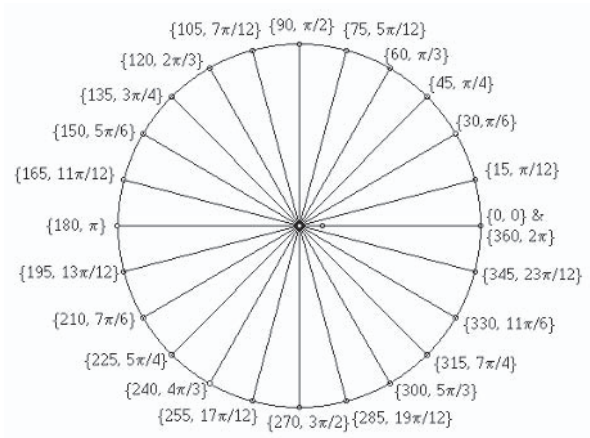


Figure 3.25: Degree to radians

***NOTE.** D3DX_PI is a Direct3D constant representing π .

■ Converts Degrees to Radians

```
#define D3DXToRadian(degree) ((degree) * (D3DX_PI / 180.0f))
```

Example:

```
Float AngleInRadians = D3DXToRadian(45);
```

■ Converts Radians to Degrees

```
#define D3DXToDegree(radian) ((radian) * (180.0f / D3DX_PI))
```

Example:

```
Float AngleInDegrees = D3DXToDegree(D3DX_PI/2);
```

The three functions Direct3D provides to build a rotation matrix are listed below. Each of them rotates about a specific axis and each simply requires the address of a matrix and an angle in radians. A code sample is also provided.


```

D3DXMATRIX *WINAPI D3DXMatrixRotationX
(
    D3DXMATRIX *pOut,
    FLOAT Angle
);

D3DXMATRIX *WINAPI D3DXMatrixRotationY
(
    D3DXMATRIX *pOut,
    FLOAT Angle
);

D3DXMATRIX *WINAPI D3DXMatrixRotationZ
(
    D3DXMATRIX *pOut,
    FLOAT Angle
);

```

Code sample:

```

D3DXMATRIX Result;
D3DXVECTOR3  FLOAT Angle = D3DXToRadian(45);

D3DXMatrixRotationX(&Result, Angle);

```

You can also rotate about a custom axis you define rather than the standard X, Y, or Z axes. To do so, Direct3D provides the function **D3DXMatrixRotationAxis**. By passing this function a vector structure, you can specify the direction and alignment of your own axis. This function can then rotate a specified number of radians around this axis. The syntax and parameters for this function follow.

```

D3DXMATRIX *WINAPI D3DXMatrixRotationAxis
(
    D3DXMATRIX *pOut,
    CONST D3DXVECTOR3 *pV,
    FLOAT Angle
);

```

D3DXMATRIX *pOut

[in, out] Pointer to the D3DXMATRIX structure that is the result of the operation.

CONST D3DXVECTOR3 *pV

[in] Pointer to the D3DXVECTOR3 structure that identifies the axis angle.

FLOAT Angle

[in] Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Matrix Scaling

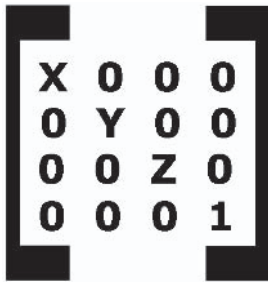


Figure 3.26: Scaling

Matrices can also encode scaling transformations. You can specify a scaling factor for each axis. This means all your geometry will be enlarged or shrunk on each axis by the scaling factor. You can manually build a scaling matrix using the template above. Just replace the X, Y, and Z with scaling factors. **Remember: 1 keeps the size the same, 0.5 halves the size, 2 makes it twice as large, etc.** Also, Direct3D provides a matrix scaling function to do this for you called **D3DXMatrixScaling**. It requires the address of a matrix for the result and three scaling factors, one for each axis.

The syntax and parameters for `D3DXMatrixScaling` appear below. Some sample code follows.

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

D3DXMATRIX *pOut

[in, out] Pointer to the `D3DXMATRIX` structure that is the result of the operation.

FLOAT sx

[in] Scaling factor that is applied along the X axis.

FLOAT sy

[in] Scaling factor that is applied along the Y axis.

FLOAT sz

[in] Scaling factor that is applied along the Z axis.

Code sample:

```
D3DXMATRIX Result;

D3DXMatrixScaling (&Result, 2.0f, 1.0f, 0.5f);
```

Combining Transformations

I said earlier that a matrix can also represent combined transformations. In other words, a single matrix can encode translation, rotation, and scaling all in one. You simply need to create a separate matrix for **each transformation, one for translation, one for scaling**, etc. Then you can join them together, resulting in a single matrix. The process of joining matrices together is called **matrix concatenation**, a term we heard earlier in this chapter. To do this, you just multiply the matrices together. That's right; it's that simple. To join two matrices you multiply them. Matrix multiplication can be

achieved with the `D3DXMatrixMultiply` function or you can use the multiplication operator. Some sample code to combine several matrices appears below.

```
D3DXMATRIX Translation;  
D3DXMATRIX Rotation;  
D3DXMATRIX Combined;  
  
Combined = Translation * Rotation;
```

Planes

So you've seen matrix mathematics and how matrices are applied to encode transformations. It's now time to move on. The final issue to consider in this chapter is planes. Imagine an ordinary, flat sheet of paper aligned in 3D space, perhaps aligned at an angle. Now imagine the sheet of paper is infinite in height and width but has no thickness. Simple; you've just imagined a plane.



Figure 3.27: Plane

A *plane* is a mathematical structure used to classify points in 3D space. Using a plane you can determine whether any point in 3D space is in front of the plane, behind the plane, or resting on the plane. This is useful for all kinds of things. For example, you can determine whether objects in 3D space are visible to the camera.

You can also use it for collision detection so your game's monsters and creatures don't walk through walls.

Direct3D represents planes using the **D3DXPLANE** structure. Don't worry too much about this structure's components. In short, this structure will allow us to represent a plane in 3D space mathematically. Direct3D provides us with some additional functions to create planes so we can choose how they are aligned.

The D3DXPLANE structure looks like this:

```
typedef struct D3DXPLANE {
    FLOAT a;
    FLOAT b;
    FLOAT c;
    FLOAT d;
} D3DXPLANE;
```

Creating Planes from Three Points

The simplest way to create a plane is with three points. Three points can define a triangle. In the context of planes, three points in 3D space exactly define a plane, assuming each of the points is resting on the plane. They show us how the plane is aligned. The function to create a plane from three points is called **D3DXPlaneFromPoints**. This function requires the address of a plane structure into which the created plane is returned, and it also requires the coordinates of the three points. Its syntax and parameters can be seen below, followed by an example.

```
D3DXPLANE WINAPI D3DXPlaneFromPoints
(
    D3DXPLANE *pOut,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2,
    CONST D3DXVECTOR3 *pV3
);
```

D3DXPLANE *pOut

[in, out] Pointer to the D3DXPLANE structure that is the result of the operation.

CONST D3DXVECTOR3 *pV1

[in] Pointer to a D3DXVECTOR3 structure, defining one of the points used to construct the plane.

CONST D3DXVECTOR3 *pV2

[in] Pointer to a D3DXVECTOR3 structure, defining one of the points used to construct the plane.

CONST D3DXVECTOR3 *pV3

[in] Pointer to a D3DXVECTOR3 structure, defining one of the points used to construct the plane.

Code sample:

```
D3DXPLANE PPlane;
D3DXVECTOR3 V1(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 V2(1.0f, 0.0f, 0.0f);
D3DXVECTOR3 V3(0.0f, 0.0f, 1.0f);
D3DXPlaneFromPoints(&PPlane, &V1, &V2, &V3);
```

Creating Planes from Point and Normal

If three points are not known and you only have one point, you can create a plane from a single point and a *normal* vector. A normal is a perpendicular arrow that sticks out from a point or polygon. In a way, it represents the direction a point or polygon is facing. A normal is a unit vector (magnitude of 1). By specifying a point and a normal you are defining the position and orientation of a plane in 3D space. Direct3D provides the **D3DXPlaneFromPointNormal** function to construct a plane from a point and a normal. This function requires an address of a resultant plane structure, a point, and a normal. The syntax and parameters follow, along with an example.

```
D3DXPLANE *WINAPI D3DXPlaneFromPointNormal
(
    D3DXPLANE *pOut,
    CONST D3DXVECTOR3 *pPoint,
    CONST D3DXVECTOR3 *pNormal
);
```

D3DXPLANE *pOut

[in, out] Pointer to the D3DXPLANE structure that is the result of the operation.

CONST D3DXVECTOR3 *pPoint

[in] Pointer to a D3DXVECTOR3 structure defining the point used to construct the plane.

CONST D3DXVECTOR3 *pNormal

[in] Pointer to a D3DXVECTOR3 structure defining the normal used to construct the plane.

Code sample:

```
D3DXPLANE Plane;
D3DXVECTOR3 Point(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 Normal(0.0f, 1.0f, 0.0f);

D3DXPlaneFromPointNormal(&Plane, &Point, &Normal);
```

Classifying Points in 3D Using Planes

Once a plane has been constructed, either from three points or from one point and a normal, you can then classify points in 3D space. This means you can test the plane to see whether a specific point is in front of the plane, behind the plane, or on the plane. Direct3D provides a function to classify a point in relation to a plane called **D3DXPlaneDotCoord**. This function requires the address of a resultant plane and a point, and returns a scalar. You should then check this value as follows:

- If the result is 0, the point and the plane are *coplanar*. This means the point is on the plane.
- If the result is greater than 0, the point is *in front of* the plane.
- If the result is less than 0, the point is *behind* the plane.

The syntax and parameters for `D3DXPlaneDotCoord` follow, and then some sample code.

```
FLOAT D3DXPlaneDotCoord
(
    CONST D3DXPLANE *pP,
    CONST D3DXVECTOR3 *pV
);
```

```
CONST D3DXPLANE *pP
```

[in] Pointer to a source `D3DXPLANE` structure.

```
CONST D3DXVECTOR3 *pV
```

[in] Pointer to a source `D3DXVECTOR3` structure.

Code sample:

```
D3DXVECTOR3 Point(0.0f, 0.0f, 0.0f);

FLOAT result = D3DXPlaneDotCoord(&Plane, &Point);
If(result == 0) //On the plane
If(result > 0) //In front of
If(result < 0) //Behind
```

Plane and Line Intersection

Another useful classification planes can provide is line intersection. Remember, a line is just two points, one for the start and one for the end of the line. By representing a line in this way we can test to see whether a line crosses through a plane and, if so, we can also determine the point at which the line crosses. Direct3D provides the `D3DXPlaneIntersectLine` function to calculate this. It requires the address of a plane structure for the result, the address of a vector structure where the intersection point will be returned if there's an intersection, and the start and end points of the line to test. Its syntax and parameters follow, and then an example.


```
D3DXVECTOR3 *WINAPI D3DXPlaneIntersectLine
(
    D3DXVECTOR3 *pOut,
    CONST D3DXPLANE *pP,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2
);
```

D3DXVECTOR3 *pOut

[in, out] Pointer to a D3DXVECTOR3 structure identifying the intersection between the specified plane and line.

CONST D3DXPLANE *pP

[in] Pointer to the source D3DXPLANE structure.

CONST D3DXVECTOR3 *pV1

[in] Pointer to a source D3DXVECTOR3 structure defining a line starting point.

CONST D3DXVECTOR3 *pV2

[in] Pointer to a source D3DXVECTOR3 structure defining a line ending point.

Code sample:

```
D3DXVECTOR3 StartPoint(-5.0f, -5.0f, -5.0f);
D3DXVECTOR3 EndPoint(5.0f, 5.0f, 5.0f);
D3DXVECTOR3 Result;

D3DXPlaneIntersectLine(&Result, &Plane, &StartPoint, &EndPoint);

if(Result)
{
    //Result now equals intersection point
}
```

***NOTE.** If the line is parallel to the plane, then NULL is returned.

Conclusion

This chapter has covered a broad area of mathematics in quite some detail. Hopefully, you now understand coordinate systems and how to represent vertices and vectors in them. Furthermore, you should have a sound grasp of transformations and how matrices can be efficiently deployed to represent them. Finally, you have seen how planes are an excellent structure for classifying points and primitives in 3D space.

The next chapter investigates how 3D environments are created and rendered in Direct3D. This is where the fun begins.

This page intentionally left blank.



Chapter 4

Direct3D for 3D Graphics

This chapter begins our voyage into the world of 3D. Much of the material we'll be covering here requires that you already know about the concepts and ideas presented in the previous two chapters. Specifically, you'll need to know how to create a Direct3D application. This involves creating a window, a Direct3D object, and a Direct3D device, and configuring a game loop to render data every frame. You'll also need to know about some of the mathematics presented in Chapter 3, like coordinate systems, vertices, and matrices. This chapter will show you how to set up a 3D environment and draw a triangle and some other primitives like triangle strips. Overall, this chapter demonstrates the following:

- How to create vertices in 3D space using vertex buffers
- FVF (flexible vertex formats)
- How to set streams
- How to draw primitives like triangles, triangle strips, and lines
- How to set up the view, projection, and transformation matrices

Getting Started

The best way to get started in the world of 3D graphics is to draw a triangle, which is simply three vertices in 3D space. For now we won't worry about lighting, textures, or other such issues; they're covered in detail later in this book. Here we'll just concentrate on rendering vertices. For a Direct3D application to render vertices and primitives it needs to perform several steps. These are listed below and the following sections examine each step in more detail.

1. **Create and define a vertex format**

In Direct3D vertices can express more than just an XYZ position in 3D space. They can also have color, store information about lighting calculations, express how textures will appear, etc. So you'll need to tell Direct3D what kind of vertices you're using. For now our vertices will just be storing position.

2. **Create a vertex buffer**

Once you've created all your vertices — three for a triangle, four for a rectangle, etc. — you load them all into a vertex buffer, which is an array of vertices.

3. **On every frame:**

■ **Set the view, projection, and transformation matrices**

This step tells Direct3D where your camera (viewpoint) is situated in 3D space. It will also tell Direct3D if your triangle needs to be transformed by a matrix.

■ **Set the stream source**

This tells Direct3D that you want to draw some vertices or some shapes. To do this you need to load your vertex buffer into the stream.

■ **Set the FVF**

During this step you tell Direct3D the format of your vertices. You defined this in step 1.

■ Draw one or more primitives

This is the actual drawing part. You have already set the stream and told Direct3D the format of your vertices. You now need to tell Direct3D what these vertices represent: a triangle, a rectangle, or something else.

4. Release the vertex buffer

When your application has finished drawing vertices, you'll need to release the created vertex buffer using the standard Release method.

Create and Define a Vertex Format

A *vertex* is a point in 3D space. Using them we can create shapes: two vertices make a line; three make a triangle, etc. These shapes are often called *primitives*. When rendering a primitive in Direct3D, the first thing you should do is create a structure to hold your vertices. You must also define what is known as an FVF (discussed below). In Direct3D vertices are expressed as data structures and each has an XYZ position. Additionally, they can contain other information such as color value or data for lighting calculations or texture information. The list of possibilities is huge. Some applications store vast amounts of data in their vertices while others may only store position data. Since Direct3D cannot predict what data you might be storing in your vertices, you'll need to tell Direct3D their exact format. For this chapter we'll be rendering a triangle and we'll only need position and color data. A structure to store this would look like the following:

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;    //Position
    DWORD color;      //Color
};
```

In order to express your vertex format, Direct3D requires you to fill in an FVF (flexible vertex format) descriptor. The *FVF* is simply

a DWORD value that holds a combination of tags describing which data your vertex stores. For the vertex structure above, the FVF descriptor would look like the following. Notice how it contains flags for XYZ position and color information.

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
```

***NOTE.** This FVF will be used later when passed to various Direct3D functions.

There are many other values your FVF can specify if you choose to store extra data. The following list contains some of these values and describes their purposes. Don't worry if you don't understand every value since much of this subject matter is included in later chapters.

D3DFVF_XYZ

Untransformed XYZ position for the vertex. In your data structure this will take the form of FLOAT X, Y, Z.

D3DFVF_NORMAL

Vertex normal. Remember, this is a perpendicular unit vector. In your data structure this will take the form of FLOAT X, Y, Z.

D3DFVF_DIFFUSE

This is the diffuse color of the vertex; the standard color. In your data structure this will take the form of FLOAT COLOR.

D3DFVF_SPECULAR

This is the color for highlights on a shiny surface. In your data structure this will take the form of FLOAT COLOR.

D3DFVF_TEX0 – D3DFVF_TEX7

Texture coordinates for this vertex. In your data structure this will take the form of FLOAT TEXTURE.

Using Vertices

Once you've created a vertex structure you can actually make some vertices. That's right; you can start declaring instances of your vertex structure and fill them with position and color information. For

a triangle this means we create three vertices, one for each corner. The sample code below demonstrates how this can be done using the vertex structure defined in the previous section. This code specifies the XYZ value first and then defines a vertex color. When the triangle is finally presented the colors will be blended over the triangle's surface, between the vertices.

```
CUSTOMVERTEX vertices[] =
{
    { 150.0f, 50.0f, 0.5f, D3DCOLOR_XRGB(255, 0, 255), },
    { 250.0f, 250.0f, 0.5f, D3DCOLOR_XRGB(0, 255,0), },
    { 50.0f, 250.0f, 0.5f, D3DCOLOR_XRGB(255, 255, 0), },
};
```

Create a Vertex Buffer

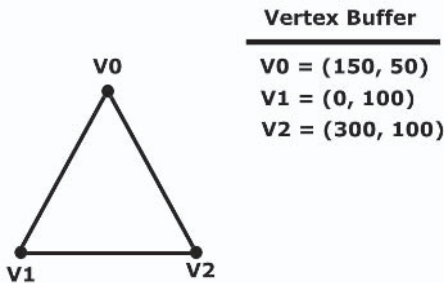


Figure 4.1

So now you've created three vertices defining the corners of a triangle. That was easy. Nothing is visible on your screen yet, though. The next step involves putting all those vertices into a buffer that Direct3D recognizes. This buffer is called a *vertex buffer*. Simply put, it represents an array of vertices. Direct3D allows you to create vertex buffers and use them through the `IDirect3DVertexBuffer9` interface. Vertex buffers are created with the **CreateVertexBuffer** method of `IDirect3DDevice9`. Remember, `IDirect3DDevice9` is the Direct3D device that represents your graphics hardware. The syntax and parameters for `CreateVertexBuffer` follow.

***NOTE.** Pointers to `IDirect3DVertexBuffer9` can either be written as `IDirect3DVertexBuffer9*`, or you can use the typecast `LPDIRECT3DVERTEXBUFFER9`.

```
HRESULT IDirect3DDevice9::CreateVertexBuffer
(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPOOL Pool,
    IDirect3DVertexBuffer9 **ppVertexBuffer,
    HANDLE *pSharedHandle
);
```

UINT Length

Size in bytes of the vertex buffer to create. This will be something like `sizeof(CUSTOMVERTEX)*3`.

DWORD Usage

Just pass 0.

DWORD FVF

The FVF of your vertex structure. This will tell Direct3D how your vertices are structured. For this value you can pass your `DWORD FVF` constant, as defined earlier.

D3DPOOL Pool

The memory in which the vertex buffer should be created. You can create this in system memory or on your graphics card. For this example we will use `D3DPOOL_SYSTEMMEM`. Possible values can be:

```
typedef enum _D3DPOOL {
    D3DPOOL_DEFAULT = 0,
    D3DPOOL_MANAGED = 1,
    D3DPOOL_SYSTEMMEM = 2,
    D3DPOOL_SCRATCH = 3,
    D3DPOOL_FORCE_DWORD = 0x7fffffff
} D3DPOOL;
```

IDirect3DVertexBuffer9 **ppVertexBuffer

Address where the created vertex buffer is returned.

HANDLE *pSharedHandle

This is a reserved value. Just pass NULL.

Notice this function does not ask for a pointer to your actual vertices. That is, it doesn't want to know where you created your vertices in memory. This is because a vertex buffer is created blank, just like surfaces are created blank. It initially contains no vertices; it just starts off as a bunch of nullified bytes. The size of the buffer matches the size you specified. In the next section we'll see how to put your vertices into the buffer. For now, take a look at the code below to see how a vertex buffer is created.

```
LPDIRECT3DVERTEXBUFFER9 g_pVB = NULL;

g_pD3DDevice->CreateVertexBuffer(sizeof(CUSTOMVERTEX)*3,
                                0, D3DFVF_CUSTOMVERTEX,
                                D3DPOOL_SYSTEMMEM,
                                &g_pVB, NULL);
```

Fill the Vertex Buffer

A vertex buffer is not created with vertices initially loaded into it. It's simply an area of memory reserved for vertex data to be loaded into it. To copy data into the vertex buffer you first need to lock it. This gives you a pointer to an area of memory where the vertices should be copied. You then copy your vertices here. The whole operation is then finished by unlocking the vertex buffer.

You can lock the vertex buffer by calling its **Lock** method. You can unlock the vertex buffer by calling **Unlock**. Both these methods are part of the **IDirect3DVertexBuffer9** interface. The syntax and parameters for **Lock** are shown below.

```
HRESULT Lock(
    UINT OffsetToLock
    UINT SizeToLock
    VOID **ppbData
    DWORD Flags
);
```

UINT *OffsetToLock*

Indicates the offset from the start of the buffer from which you want to lock. This is measured in bytes. In most cases you will want to lock the entire buffer for modification. To do this, just pass 0.

UINT *SizeToLock*

Indicates the size in bytes, from *OffsetToLock*, that should be locked for modification. To lock the entire buffer, pass 0.

VOID *ppbData***

The address of a pointer that is to receive the buffer pointer. After the function has been called you'll copy your vertex data here.

DWORD *Flags*

This value allows you to choose the behavior of the vertex buffer. For example, you can access the buffer in read-only mode, write mode, etc. Mostly you'll pass 0 for this value.

We use the **memcpy** function to copy data from one memory location to another. It requires source and destination pointers and the amount of data in bytes you want to copy. Its syntax and parameters are shown below.

```
void* memcpy
(
    void* Dest,
    const void* Source,
    DWORD Size
);
```

void* *Dest*

Destination to receive copied bytes. This is where you want to copy data.

const void* *Source*

Source of bytes. This is where data will be taken from.

DWORD Size

Number of bytes to copy.

***NOTE.** Be sure to lock the vertex buffer before copying and be sure to unlock the vertex buffer when completed. If you do not unlock the vertex buffer before presenting the frame, no vertices will be rendered.

Assuming we're using the vertices created earlier in the section called "Using Vertices," which define the corners of a triangle, we could write some code to fill the vertex buffer like below. Notice the use of Lock and Unlock and how vertices are copied into the buffer using the memcpy function.

```
VOID* pVertices = NULL;
if(FAILED(g_pVB->Lock(0, sizeof(vertices), (void**)&pVertices, 0)))
    return E_FAIL;

memcpy(pVertices, vertices, sizeof(vertices));
g_pVB->Unlock();
```

Rendering the Triangle

Now that you've created some vertices and you have a vertex buffer set up, you're almost ready to draw the triangle. However, there are just a few more steps to perform before we'll see anything. All of these steps will occur inside the rendering loop of our application. First, we'll need to set the view, transform, and projection matrices. Then we'll use the Direct3D device to set the stream source and FVF, and finally to draw our triangle. These steps are explained in more detail over the next few sections.

View, Projection, and Transform Matrices

Chapter 3 explained the concept of matrices. To recap, a *matrix* is a grid of numbers that acts like a set of instructions. Matrices are often used to express transformations and can tell Direct3D how to

move objects around in 3D space. To present any 3D scene in Direct3D you'll need to fill in three kinds of matrices. These are the view, projection, and transform matrices. Once filled in, you'll pass them to your Direct3D device so that it will know how to display a scene properly in your application window. Let's take a quick look at each type of matrix.

■ **Transform matrix**

The transform matrix tells Direct3D how to transform the geometry it's about to render. It instructs Direct3D **where to position your triangle in 3D space, as well as how to rotate it and how to scale it**. By setting the transform matrix you're saying, "Hey Direct3D, I've got a transformation matrix here and I want you to apply it to any geometry that is rendered."

■ **View matrix**

The view matrix tells Direct3D **where to position your camera in 3D space**. It orients it too, so your camera can be looking up, down, left, right, etc. When Direct3D comes to render your 3D world it will do so from the vantage point of your camera.

■ **Projection matrix**

The projection matrix tells Direct3D **how to project a 3D image from your camera onto the flat 2D surface of your screen**. In other words, this matrix makes sure you can see your 3D world accurately.

Transformation Matrix

The first matrix to set is the transformation matrix. This tells Direct3D how to position and orient vertices in 3D space. They can make your polygons dance. In other words, if you want to move your triangle about a 3D world, you'll need to use a transformation matrix to do it. In Chapter 3 you learned about transformation matrices. Specifically, you saw how transformations can be a combination of translation, rotation, and scaling. For this example, we'll just **move the triangle to the origin by translating it to (0, 0, 0)**. The code to build a transformation matrix that does this follows.

```
D3DXMATRIX g_Transform;
D3DXMatrixIdentity(&g_Transform);
D3DXMatrixTranslation(&g_Transform, 0.0f, 0.0f, 0.0f);
```

*** NOTE.** Remember, when you create a matrix, be sure to initialize it as an identity matrix. Like this:

```
D3DXMATRIX Matrix;
D3DXMatrixIdentity(&Matrix);
```

*** NOTE.** You can combine transformations by multiplying matrices.



TIP. If you adjust the transformation matrix over time, you can animate your triangle. For example, you can alter its rotation on each frame so the triangle spins.

Once you've created a transformation matrix you'll need to notify Direct3D. You do this by calling the **SetTransform** method of **IDirect3DDevice9**. You pass it your transformation matrix as an argument. This allows Direct3D to know which transformation to use when rendering vertices. The **SetTransform** method will also be used to set the view and projection matrices too, as shown later. The syntax and parameters for **SetTransform** follow.

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

D3DTRANSFORMSTATETYPE State

Represents the matrix type. This is where you tell Direct3D whether you're setting the transform, view, or projection matrix. The values can be:

```
D3DTS_WORLD //Transform
D3DTS_VIEW //View
D3DTS_PROJECTION //Projection
```

CONST D3DMATRIX *pMatrix

Pointer to your matrix. In other words, the address of the matrix you wish to use.

Some code to set the world transformation matrix looks like this:

```
g_pd3dDevice->SetTransform(D3DTS_WORLD, &g_Transform);
```

View Matrix

Just as the transformation matrix can position and orient polygons in a 3D world, the view matrix can position and orient your **camera**. That's right; the view matrix exclusively controls where your camera is situated in 3D space. This means when Direct3D presents a scene to the window it does so from the vantage point of your camera. This opens up all kinds of possibilities. For example, if you animate your view matrix over time you can create a first-person camera that moves around a 3D world. Chapter 8 shows you how to do this.

A view matrix can be built manually, or you can use one of Direct3D's provided functions. Specifically, you can use **D3DXMatrixLookAtLH**. This function builds a view matrix according to three arguments: the position of your camera, the point at which it's looking, and which way is up. The syntax and parameters follow.

```
D3DXMATRIX *WINAPI D3DXMatrixLookAtLH
(
    D3DXMATRIX *pOut,
    CONST D3DXVECTOR3 *pEye,
    CONST D3DXVECTOR3 *pAt,
    CONST D3DXVECTOR3 *pUp
);
```

D3DXMATRIX *pOut

Address to where a view matrix is returned.

CONST D3DXVECTOR3 *pEye

The camera's position. This might be something like (0, -15, 0).

CONST D3DXVECTOR3 *pAt

The point at which the camera is looking.

CONST D3DXVECTOR3 *pUp

A vector indicating which way is up. It might seem silly to pass this value, but to perform calculations correctly and to make sure your image is the right way up, Direct3D needs to know this. Usually, up will be (0, 1, 0). Upside down would be (0, -1, 0).

Some sample code to build a view matrix follows. It will also use SetTransform to set it as Direct3D's active view matrix. This works just like setting the transform matrix, except we pass D3DTS_VIEW for the matrix type parameter. Take a look at the following code.

```
D3DXMATRIX g_View;
D3DXVECTOR3 Eye (0.0f, -5.0f, 0.0f);
D3DXVECTOR3 LookAt(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 Up(0.0f, 1.0f, 0.0f);

D3DXMatrixLookAtLH(&g_View, &Eye, &LookAt, &Up);
g_pd3dDevice->SetTransform(D3DTS_VIEW, &g_View);
```

Projection Matrix

The projection matrix tells Direct3D how to **project your 3D world, as seen from your camera, onto the flat 2D surface of your application's window.** In other words, it makes sure you get to see your 3D world properly, with objects closer to the camera looking bigger than objects farther away. Mathematically, it's the most complicated matrix to compute. Don't worry if you don't understand the mathematics behind it; Direct3D provides a function to do all that hard work for you. This function is called D3DXMatrixPerspectiveFovLH, and it builds a projection matrix according to some arguments you provide. Its syntax and parameters follow.

```
D3DXMATRIX *WINAPI D3DXMatrixPerspectiveFovLH
(
    D3DXMATRIX *pOut,
    FLOAT fovy,
    FLOAT Aspect,
```



```

    FLOAT zn,
    FLOAT zf
);

```

D3DXMATRIX *pOut

Address of a matrix that is to become the final projection matrix. This is where you put your matrix.

FLOAT fovy

Aspect ratio, defined as view space width divided by height. This value will nearly always be D3DX_PI/4.

FLOAT Aspect

This value is window width/window height.

FLOAT zn

Z value of the near view plane. This value will usually be 1.0f.

FLOAT zf

This value indicates how far into the distance the camera can see. Objects beyond this distance will be considered beyond the horizon and will not be drawn. For a simple triangle sample, this value might be something like 500.0f.

The code below shows how you can build a projection matrix. It also uses SetTransform to set it as the current projection matrix. This time we pass it the D3DTS_PROJECTION flag. Take a look.

```

D3DXMATRIX Projection;
FLOAT FOV = D3DX_PI / 4;
FLOAT Aspect = WindowWidth/WindowHeight;

D3DXMatrixPerspectiveFovLH(&Projection, FOV, Aspect, 1.0f, 500.0f);
g_pd3dDevice->SetTransform(D3DTS_PROJECTION, &Projection);

```

Setting the Stream Source

Once the transform, view, and projection matrices are set it means your scene is almost ready to draw. This is because we've now told Direct3D how geometry is transformed, where the camera is positioned, and how the world is drawn in the window. All that remains

is to prepare Direct3D for actually rendering the triangle. This begins by setting the stream source. A Direct3D *stream* is a channel of information. Direct3D has several streams, which it numbers from 0 onward. You tell Direct3D which vertices to render by connecting the vertex buffer to a stream. This is like saying, “Hey Direct3D, I have some vertices to draw and you’ll find them waiting in Channel 1.” To connect a vertex buffer to a stream you call the **SetStreamSource** method of IDirect3DDevice9. The syntax and parameters for this function are listed below.

```
HRESULT SetStreamSource(
    UINT StreamNumber,
    IDirect3DVertexBuffer9 *pStreamData,
    UINT OffsetInBytes,
    UINT Stride
);
```

UINT StreamNumber

This is the number of the stream to which your vertex buffer should be connected. In this case we’ll pass 0, which means the first stream.

IDirect3DVertexBuffer9 *pStreamData

This is a pointer to the vertex buffer to be connected. In other words, you’ll pass your vertex buffer pointer here.

UINT OffsetInBytes

This is an offset from the beginning of the stream in bytes where vertex data begins. To set this to the beginning you should pass 0.

UINT Stride

This is the size in bytes of a single vertex. The value would be something like sizeof(CUSTOMVERTEX).

Here’s some code to set the stream source for the triangle. This won’t actually render data yet; there are still two more steps.

```
g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(CUSTOMVERTEX));
```

Setting the FVF

After setting the stream source we next set the FVF. Remember, FVF stands for flexible vertex format. It's a DWORD descriptor that tells Direct3D what data your vertices contain, such as position, color, texture information, etc. Before we can render the triangle we'll need to pass your FVF to Direct3D so it can understand the vertices loaded into the stream. This is really simple to do. To set the FVF you must call the SetFVF method of IDirect3DDevice9. It requires only one parameter, your FVF. The code to achieve this follows.

```
g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
```

Drawing a Primitive



Figure 4.2

Finally, we can draw a triangle. The vertex buffer has been created and filled. The transform, view, and projection matrices have been set. The stream source has been linked to a vertex buffer and now the FVF has been defined. This means we can go ahead and give

the command for Direct3D to draw something. To draw a triangle or any other primitive from vertices in a stream you call the **DrawPrimitive** method of `IDirect3DDevice9`. It requires you to indicate what type of primitive to draw (triangle list, triangle strip, etc.), which vertex in the stream is the first vertex, and finally how many primitives to draw. Its syntax and parameters follow.

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

D3DPRIMITIVETYPE PrimitiveType

This is the kind of primitive you want to draw. For a standard triangle this must be `D3DPT_TRIANGLELIST`. It can also be other values for triangle strips, line strips, triangle fans, etc. These types were covered in Chapter 3. Possible values can be:

```
typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST = 1,
    D3DPT_LINELIST = 2,
    D3DPT_LINESTRIP = 3,
    D3DPT_TRIANGLELIST = 4,
    D3DPT_TRIANGLESTRIP = 5,
    D3DPT_TRIANGLEFAN = 6,
    D3DPT_FORCE_DWORD = 0xffffffff
} D3DPRIMITIVETYPE;
```

UINT StartVertex

The first vertex in the stream to use for rendering. This value will usually be 0.

UINT PrimitiveCount

The number of primitives to draw using the vertices in the stream. For one triangle this value will be 1.

The following sample code represents the entire render procedure, including how to draw a triangle using `DrawPrimitive`. Notice the function calls to `Clear`, `BeginScene`, `EndScene`, and `Present`. These

functions were explained in Chapter 2. This procedure represents one frame and is called many times per second to draw a 3D scene.

```

VOID Render()
{
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET,
                      D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0);

    //Set the matrices
    D3DXMATRIX g_Transform;
    g_pd3dDevice->SetTransform(D3DTS_WORLD, &g_Transform);

    D3DXMATRIX g_View;
    D3DXVECTOR3 Eye(0.0f, -5.0f, 0.0f);
    D3DXVECTOR3 LookAt(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 Up(0.0f, 1.0f, 0.0f);

    D3DXMatrixLookAtLH(&g_View, &Eye, &LookAt, &Up);
    g_pd3dDevice->SetTransform(D3DTS_VIEW, &g_View);

    D3DXMATRIX Projection;
    FLOAT FOV = D3DX_PI / 4;
    FLOAT Aspect = WindowWidth/WindowHeight;

    D3DXMatrixPerspectiveFovLH(&Projection, FOV, Aspect, 1.0f, 500.0f);
    g_pd3dDevice->SetTransform(D3DTS_PROJECTION, &Projection);

    // Begin the scene
    if(SUCCEEDED(g_pd3dDevice->BeginScene()))
    {
        g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(CUSTOMVERTEX));
        g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
        g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

        // End the scene
        g_pd3dDevice->EndScene();
    }
}

```

```
// Present the back buffer contents to the display
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}
```

Animating the Triangle

Now that the triangle has been drawn and everything is looking good, let's try something a bit more adventurous. In this section we'll make the triangle rotate; that's right, make it spin around the Y axis. Rotation is really simple to do; it's all related to matrices. You simply change the transform matrix every frame. We simply build a rotation matrix using `D3DXMatrixRotationY` (we saw this in the previous chapter) and on each frame we adjust the angle of rotation a little. To make the triangle spin, simply adjust the transform matrix code as follows:

```
//Set the matrices
UINT iTime = timeGetTime() % 1000;
FLOAT fAngle = iTime * (2.0f * D3DX_PI) / 1000.0f;
D3DXMatrixRotationY(&g_Transform, fAngle);
g_pd3dDevice->SetTransform(D3DTS_WORLD, &g_Transform);
```

Drawing Other Primitives

In Direct3D you're not limited to just drawing a triangle. You can draw many triangles in all kinds of arrangements. You can even join them together to make cubes, spheres, boxes, and many more shapes. Most complex models are created in a 3D modeling package like 3D Studio MAX, Maya, or Softimage. Then they are exported from the package and imported into Direct3D as meshes. Meshes are explained in Chapter 7. This section shows you how to create a number of other basic primitive types, specifically, point lists, line lists, line strips, triangle strips, and triangle fans.

■ Point list — D3DPT_POINTLIST

You don't have to render your vertex buffers as triangles. You can also fill in your vertex buffer with lots of vertices and render them all as unconnected points — just a load of dots in 3D space. This is called a *point list*, which is a collection of one or more points. Point lists can be used to simulate star effects. DrawPrimitive can render a point list like this:

```
g_pd3dDevice->DrawPrimitive(D3DPT_POINTLIST, 0, NumPoints);
```

***NOTE.** In the above code sample, NumPoints is the number of points to render. You should have at least that many vertices in the vertex buffer.

■ Line list — D3DPT_LINELIST

Similar to a point list, a *line list* is a collection of one or more lines. A line is formed by two vertices: a start point and an end point. So for every line, you need two vertices in the vertex buffer. DrawPrimitive can render a line list like this:

```
g_pd3dDevice->DrawPrimitive(D3DPT_LINELIST, 0, NumLines);
```

■ Line strip — D3DPT_LINESTRIP

A *line strip* is a collection of one or more connected lines. If there is more than one line, then the next line's start point joins onto the end point of the previous line. DrawPrimitive can render a line strip like this:

```
g_pd3dDevice->DrawPrimitive(D3DPT_LINESTRIP, 0, NumLines);
```

■ Triangle strip — D3DPT_TRIANGLESTRIP

In a similar fashion to a line strip, a *triangle strip* is a collection of one or more connected triangles. Each triangle joins onto the edge of the previous triangle. DrawPrimitive can render a triangle strip like this:

```
g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, NumTriangles);
```

■ Triangle fan — D3DPT_TRIANGLEFAN

Triangle fans are collections of one or more triangles where every triangle shares a common vertex. This is a bit like a triangle strip except the triangles will form a fan shape.

DrawPrimitive can render them like this:

```
g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, NumTriangles);
```

Indexed Primitives

Indexed primitives are an optimization on the standard primitives we've seen so far. In other words, indexed primitives are generally more efficient than standard primitives. To illustrate, let's consider a rectangle. In Direct3D, a rectangle is made from two right-angle triangles aligned with diagonals touching. In Direct3D, a triangle is often called a face or a polygon. A rectangle made from two triangles is shown in Figure 4.3.

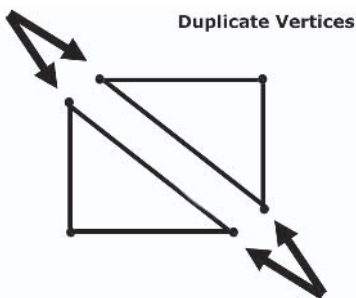


Figure 4.3

This means to represent this rectangle we need to store six vertices, one for each corner of a triangle. There are three vertices per triangle and there are two triangles. Notice the problem here? Those two triangles have vertices in the same place: two of the corner points. There's an overlap. Wouldn't it be nice if we could share those two end vertices between both triangles? This means we could store fewer vertices in our vertex buffer and Direct3D would process fewer vertexes. This is the issue indexed primitives

attempt to resolve. You'll find them particularly useful when representing complex geometry, that is, objects with lots of vertices.

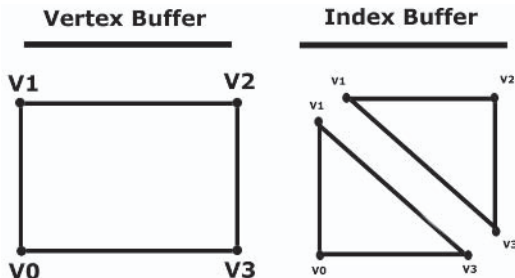


Figure 4.4

Indexed primitives work very much like standard primitives. We'll still be using FVFs and vertex buffers; however, this time around, vertex buffers will be used as simply a storage place for vertices. They will store all the vertices of our primitive, but the vertices will not be arranged in any specific order. They will not relate to the order in which Direct3D draws geometry. Also, the buffer will not store any duplicate vertices. Every vertex in the buffer will be unique. To define the order and arrangement of vertices we'll need a separate buffer, called an index buffer. This is represented by the `IDirect3DIndexBuffer9` interface. Take a look at Figure 4.4 to see how an index buffer works.

Every element in the index buffer is an offset into the vertex buffer. In other words, every element points to a vertex. The index buffer refers to vertices in the vertex buffer by their offset. Let's take our rectangle for example. Previously we would have stored six vertices, and some of them would be duplicates where the two triangles overlap. Now, this time, we only store four vertices in the vertex buffer: the corner points of the rectangle. Remember, no duplicates are allowed. Then using the index buffer we define our two triangles by referencing the vertices in the vertex buffer. So our first three entries in the index buffer would be 0, 1, and 2. This tells Direct3D that our first triangle is made up from the first three points in the vertex buffer. Then our next three entries in the index buffer would be 1, 2, and 3. This defines our second triangle. Notice how we share points 1 and 2 between both triangles. Simple.

Setting Up Index Buffers

Setting up your application to use index buffers requires only a few amendments. You still create your vertices, define an FVF, and create a vertex buffer. Once this is created, you will create an index buffer with the **CreateIndexBuffer** method of `IDirect3DDevice9`. Its syntax and parameters are listed below, followed by some sample code to populate a vertex buffer with four vertices, which specify a rectangle's corners. Notice how, like vertex buffers, the index buffer is locked while data is copied into it. This data will not be vertices. Instead, you copy over numbers that represent offsets into the vertex buffer. The next section shows how you can draw primitives with index buffers.

```
HRESULT CreateIndexBuffer(
    UINT Length,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DIndexBuffer9 **ppIndexBuffer,
    HANDLE *pSharedHandle
);
```

UINT Length

Size of the index buffer in bytes. For a rectangle you'll need to hold six indices, so this would be `sizeof(short)*6`.

DWORD Usage

Just pass 0.

D3DFORMAT Format

This can be either `D3DFMT_INDEX16` or `D3DFMT_INDEX32`. Most of the time you'll pass `D3DFMT_INDEX16`.

D3DPOOL Pool

The area of memory in which to create the index buffer. On most occasions you'll pass `D3DPOOL_DEFAULT`.

IDirect3DIndexBuffer9 **ppIndexBuffer

Address to where a created index buffer is to be returned.

HANDLE *pSharedHandle

This is a reserved value. Just pass NULL.

Sample code:

```
//Create vertices for rectangle
CUSTOMVERTEX vertices[] =
{
    { 50.0f, 50.0f, 0.5f, 0xffff0000, }, // x, y, z
    { 250.0f, 250.0f, 0.5f, 0xff00ff00, },
    { 50.0f, 250.0f, 0.5f, 0xff00ffff, },
    { 250.0f, 50.0f, 0.5f, 0xff00ffff, },
};

//Create vertex buffer
if(FAILED(g_pd3dDevice->CreateVertexBuffer(4*sizeof(CUSTOMVERTEX),
                                          0, D3DFVF_CUSTOMVERTEX,
                                          D3DPPOOL_DEFAULT, &g_pVB, NULL)))
{
    return E_FAIL;
}

//Fill vertex buffer
VOID* pVertices;
if(FAILED(g_pVB->Lock(0, sizeof(vertices), (void**)&pVertices, 0)))
    return E_FAIL;
memcpy(pVertices, vertices, sizeof(vertices));
g_pVB->Unlock();

//Create index buffer
if(FAILED(g_pd3dDevice->CreateIndexBuffer(sizeof(short)*6,
                                          0, D3DFMT_INDEX16,
                                          cc, &g_pIndexBuffer, NULL)))
{
    return E_FAIL;
}
```

```
//Fill index buffer
short Indices[6] =
{
    0, 1, 2,
    3, 1, 0
};

VOID* IndexData = NULL;
if(SUCCEEDED(g_pIndexBuffer->Lock(0, 0, &IndexData, 0)))
{
    memcpy(IndexData, (void*) &Indices, sizeof(Indices));

    g_pIndexBuffer->Unlock();
}
```

Drawing Indexed Primitives

Once you've created a vertex and index buffer you can draw your indexed primitive. This works very much like drawing a normal primitive. It still occurs during the rendering loop and you still need to call `SetStreamSource` and set the FVF; however, you'll also need



Figure 4.5

to call the **SetIndices** method of `IDirect3DDevice9`. This tells Direct3D which index buffer to use. Additionally, you no longer call `DrawPrimitive` to render your vertices; instead you'll call **DrawIndexedPrimitive**. This does the same thing as `DrawPrimitive` except it's used for indexed primitives. The syntax and parameters for both `SetIndices` and `DrawIndexedPrimitive` follow. Then we'll see some sample code to render a rectangle (two diagonally aligned triangles).

```
HRESULT IDirect3DDevice9::SetIndices
(
    IDirect3DIndexBuffer9 *pIndexData
);
```

IDirect3DIndexBuffer9 *pIndexData

Pointer to the index buffer used for rendering.

```
HRESULT IDirect3DDevice9::DrawIndexedPrimitive
(
    D3DPRIMITIVETYPE Type,
    INT BaseVertexIndex,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);
```

D3DPRIMITIVETYPE Type

The type of primitive you're drawing. Possible values are the same as for `DrawPrimitive`. In the case of a rectangle we'll choose triangle list.

INT BaseVertexIndex

The vertex from which rendering should begin. In most cases this will be 0.

UINT MinIndex

This is the minimum index for vertices that you'll be using. Usually this will be 0.

UINT NumVertices

This is the maximum number of vertices you'll be using. For a rectangle this will be four — one for each corner.

UINT StartIndex

The position in the index buffer to start reading vertices. Normally this will be 0.

UINT PrimitiveCount

This is the number of primitives you'll be rendering. For a rectangle this will be two because there are two triangles.

Sample code:

```
// Clear the back buffer to a blue color

g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET,
                  D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0);

// Begin the scene

if(SUCCEEDED(g_pd3dDevice->BeginScene()))
{
    g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(CUSTOMVERTEX));
    g_pd3dDevice->SetIndices(g_pIndexBuffer);
    g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
    g_pd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 4, 0, 2);

    // End the scene
    g_pd3dDevice->EndScene();
}

// Present the back buffer contents to the display
g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
```

Conclusion

This chapter has covered a lot of new and important ground. Having come this far you should understand how Direct3D renders geometry in 3D space. Specifically, you have seen how this is accomplished largely with the help of vertex buffers and index buffers.

The next chapter moves on to make your 3D environments look more realistic. It'll show you how to make your geometry look as though it's made from real-life materials — like walls made from brick, trees looking like trees, and all kinds of stuff. Furthermore, it'll examine how we can light your scene or make some areas brighter than others. In short, the next chapter examines materials, textures, and lights.



Chapter 5

Materials, Lights, and Textures

The previous chapter introduced the basic concepts governing the architecture of a Direct3D application, including the rendering loop and how a Direct3D device renders graphical data to the screen. Additionally, it explained how to create and render vertices for standard 3D primitives like triangles, rectangles, and cubes. This chapter goes on to explain how the realism of 3D primitives and a 3D world can be enhanced using materials, lights, and textures. Specifically, this chapter discusses the following points:

- Lighting: Flat shading and Gouraud shading
- Materials: Diffuse, ambient, and specular color
- Light types: Spotlights, point lights, and directional lights
- Textures
- Texture coordinates and texture addressing
- ID3DXSprite interface

Lighting

Until now your Direct3D applications have either been very, very dark because the lighting level was low, or they looked perfectly normal because Direct3D lighting was disabled. Regardless, it's pretty obvious by now that Direct3D has a lighting system, and programmers use this to illuminate their 3D world, much like we use lights to illuminate our own world. However, before we consider Direct3D lighting further, let's examine light in the real world.

As you may remember from high school science, light bounces off everything. Some objects absorb light and other objects, like glass or metal, scatter light in various directions. The net result means that the appearance of every object we see — its coloration and hue — is affected by light. For example, the color of a white sheet of paper will change when placed beneath green, red, or some other colored light. Nowadays, computers are powerful enough to simulate this kind of lighting, and various applications are available to render 3D graphics with advanced lighting. However, such accurate simulation of light severely reduces the speed at which 3D scenes are calculated, and therefore it becomes unsuitable for the fast needs of real-time graphics like Direct3D. So Direct3D makes compromises and uses a less accurate lighting system that still produces results close enough to the real thing to make our 3D worlds believable.

So what does Direct3D do? Simply put, Direct3D uses a vertex-based lighting system. This means Direct3D gives an individual weight to each vertex in the scene. Then, it checks to see where each light is located, how bright each light is, and in which direction it's facing, which it then compares to each vertex weight and blends the light appropriately across the polygon's surface, between the vertices. Don't worry too much if this doesn't make much sense. Direct3D hides a lot of implementation from us and makes lighting really simple to use. Let's take a look at how lighting is used in Direct3D.

Turning the Lights On and Off

A good place to start learning about lights is how to enable and disable lighting in Direct3D. This is really simple to do. Remember, this process doesn't actually put any lights into the scene; it simply tells Direct3D whether or not to enable lighting. To enable or disable lighting you call the **SetRenderState** method of **IDirect3DDevice9**. This function can be used to set a whole lot of Direct3D states. Its syntax and parameters follow.

```
HRESULT SetRenderState(  
    D3DRENDERSTATETYPE State,  
    DWORD Value  
);
```

D3DRENDERSTATETYPE State

This parameter can be one of many different values. To enable or disable lighting, this value should be **D3DRS_LIGHTING**.

DWORD Value

The value of this parameter depends on the first parameter, *State*. For lighting, this value will be either **True** or **False** to indicate whether lighting is to be enabled or disabled respectively.

Below is some code to enable lighting in Direct3D:

```
g_pDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
```

*** NOTE.** If lighting is enabled and there are no lights in the scene, then everything will appear dark. On the other hand, if lighting is disabled, the scene will *not* appear dark. Instead, Direct3D will use its default lighting configuration.

More on Lights

Now that you know how to enable and disable lighting in Direct3D, let's look at lighting modes. In Direct3D there are two modes of lighting, both of which are independent of the other: ambient lighting and direct lighting.

■ Ambient lighting

Think of this as general lighting — like brightness and contrast. It's rather like a general light that pervades the scene and distributes its brightness and color equally and infinitely. There is only one ambient light, and by adjusting its properties the overall lighting of Direct3D is changed.

■ Direct lighting

Direct lighting is generally what we think of as lighting. It's the system that allows developers to place lights in a scene. These lights will have a specific position, brightness, and direction.

Consequently, depending on their brightness, they may not illuminate the entire scene.

Setting the Ambient Lighting

As explained above, ambient lighting controls the overall lighting of a scene. Setting the ambient light in Direct3D is a simple process: You just call the `SetRenderState` function — the same function to enable or disable lighting — and you pass `D3DRS_AMBIENT` for the first parameter. For the second parameter you pass a `D3DCOLOR` structure specifying the color and alpha for the light in the form of RGBA (red, green, blue, and alpha). As you learned from Chapter 2, these color components can be any value from 0 to 255. Here's an example to set the ambient lighting to red:

```
g_pDevice->SetRenderState(D3DRS_AMBIENT, D3DCOLOR_RGBA(255,0,0,22));
```

Getting Started with Direct Lights

Direct lights are the primary part of Direct3D's lighting system. They typically have a position, brightness, and direction. They usually won't illuminate the entire scene and they can be moved around in real time.

Before we examine direct lights practically, we'll need to learn more about how Direct3D simulates lighting using a shading mode, and we'll also need to understand how polygons use materials to instruct Direct3D how lighting is to interact with their surfaces (faces).

Direct3D Shading Modes

As mentioned earlier, Direct3D does not accurately simulate lighting as it behaves in the real world. Instead, it expedites lighting calculations by approximating light so it looks real enough to the average eye. To light a polygon's surface Direct3D uses one of two shading modes: flat shading or Gouraud shading. Depending on which shading mode you use, Direct3D will light a polygon's surface differently.

■ Flat shading

This shading mode calculates quickly but isn't very accurate. As a result, this speeds things up nicely but doesn't look particularly realistic. Nowadays, hardly any game uses flat shading.

■ Gouraud shading

This is the most realistic shading mode. In this mode the light is blended smoothly across the polygon's surface, between each vertex.

To set Direct3D's shading mode you call the `SetRenderState` method of `IDirect3DDevice9`. By default it will be set to Gouraud shading. For the first parameter you should pass `D3DRS_SHADEMODE`. The second parameter is then used to set the shading mode.

D3DSHADE_FLAT

Flat shading

D3DSHADE_GOURAUDGouraud shading

Here's an example of setting Direct3D's shading mode to Gouraud shading:

```
g_pDevice->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
```

Materials

In Direct3D materials are simple structures that define how lights affect a polygon's surface. Direct3D assigns a material to each vertex in a polygon and each material defines how light will interact with the surface of a polygon. For example, light acts differently when striking a shiny surface as opposed to a soft cushion. So materials can make the surfaces of your polygons look shiny, dull, smooth, jagged, etc. Take a look at the following material structure.

```
typedef struct _D3DMATERIAL9 {  
    D3DCOLORVALUE Diffuse;  
    D3DCOLORVALUE Ambient;  
    D3DCOLORVALUE Specular;  
    D3DCOLORVALUE Emissive;  
    float Power;  
} D3DMATERIAL9;
```

As you can see, this structure contains various color values and a power factor. Together, these properties define how a vertex will react when hit by light. The following list describes the various members.

■ **Diffuse color**

Diffuse color is probably the most significant color for your vertices; it defines how vertices reflect light. Essentially, when your vertices are hit by light, the diffuse color defines what color your vertices will be.

■ **Ambient color**

The opposite of diffuse color, ambient color defines how vertices appear when not in direct light. Typically, this value will be the same as for diffuse color.

■ **Emissive color**

Emissive color creates the effect of fluorescence when the vertices are hit by light. In other words, by specifying the emissive color you can make your vertices glow.

■ **Specular color**

Specular color makes objects shine a specific color. This value is combined with power. Materials with 0 power don't shine, while materials with higher values have a greater shine.

***NOTE.** You can set each color component by using the standard `D3DCOLOR_RGBA` macro.

Once you've filled in a material structure you're ready to apply it to vertices. Of course, it won't have much of an effect if there are no lights in the scene. We'll see how to add lights later in this chapter. To apply a material to one or more vertices you call the **SetMaterial** method of `IDirect3DDevice9` as you draw vertices during the rendering loop, like when you call `DrawPrimitive` or `DrawIndexedPrimitive`. The syntax and parameter for `SetMaterial` are as follows.

```
HRESULT SetMaterial(
    CONST D3DMATERIAL9 *pMaterial
);
```

```
CONST D3DMATERIAL9 *pMaterial
```

Pointer to a material structure to set as the active material.

Take a look at the code below. It demonstrates how to fill in a material structure and use `SetMaterial` to apply it to the vertices of a primitive.

```
D3DMATERIAL9 g_Material;  
ZeroMemory(&g_Material, sizeof(D3DMATERIAL9));  
g_Material.Diffuse = D3DCOLOR_RGBA(1.0, 1.0, 1.0, 0.0);  
g_Material.Ambient = D3DCOLOR_RGBA(0.0, 0.0, 0.0, 0.0);  
g_Material.Specular = D3DCOLOR_RGBA(0.0, 0.0, 0.0, 0.0);  
g_Material.Emissive = D3DCOLOR_RGBA(0.0, 0.0, 0.0, 0.0);  
g_Material.Power = 0;  
g_pDevice->SetMaterial(&g_Material);
```

Direct Lighting Types

The previous section discussed a material structure and how it defines the way a surface should react when it's hit by light. Additionally, it explained how a material is assigned to vertices as they're rendered to the display. However, materials on their own simply don't cut it. You still need to add lights to your scene before any lighting will take effect. This section discusses direct lights and the various types available to you. Then we'll see how to use them practically.

■ Point lights

Point lights have a position in space and radiate light equally in all directions.

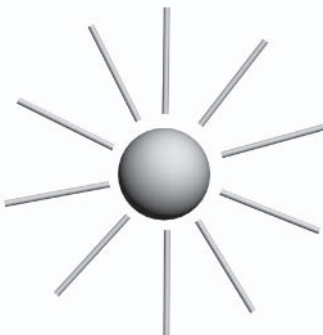


Figure 5.1: Point light

■ Spotlights

Spotlights have a position in space. Additionally, they have a direction in which they cast light. Furthermore, they have two invisible cones that determine the brightness and stretch of the light they cast: an inner cone and outer cone.

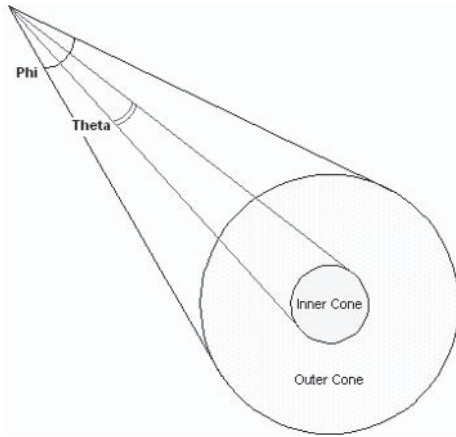


Figure 5.2: Spotlight

■ Directional lights

Directional lights have no physical position in space but cast light in a direction. You can think of a directional light as the sun.



Figure 5.3: Directional light

Creating a light in Direct3D is simple. You just need to fill in a `D3DLIGHT9` structure specifying various properties about the light, such as color, position, type, and direction. Here's what a

D3DLIGHT9 structure looks like. The parameters are described below.

```
typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE Type;
    D3DCOLORVALUE Diffuse;
    D3DCOLORVALUE Specular;
    D3DCOLORVALUE Ambient;
    D3DVECTOR Position;
    D3DVECTOR Direction;
    float Range;
    float Falloff;
    float Attenuation0;
    float Attenuation1;
    float Attenuation2;
    float Theta;
    float Phi;
} D3DLIGHT9;
```

D3DLIGHTTYPE *Type*

Defines the light type. This can be any value from the following enumeration D3DLIGHTTYPE:

```
D3DLIGHT_POINT
Point light
D3DLIGHT_SPOT
Spotlight
D3DLIGHT_DIRECTIONAL
Directional light
```

D3DCOLORVALUE *Diffuse*

Specifies the color of the diffuse light. This value can be generated by the D3DCOLOR_RGBA macro.

D3DCOLORVALUE *Specular*

Specifies the color of the specular light. This value can be generated by the D3DCOLOR_RGBA macro.

D3DCOLORVALUE *Ambient*

Specifies the color of the ambient light. This value can be generated by the `D3DCOLOR_RGBA` macro.

D3DVECTOR *Position*

Specifies the XYZ position of the light. This value is a `D3DXVECTOR3` structure, which is three floats. Remember, Direct3D will ignore this value for directional lights since they have no position.

D3DVECTOR *Direction*

A normalized vector specifying the direction of the light. This value is only meaningful for directional lights and spotlights. A value of (0, 0, 1) points into the monitor and (0, 0, -1) points outward from the monitor.

float *Range*

Specifies the maximum distance at which the light can shine. In other words, the greatest distance at which an object can still be hit by the light.

float *Falloff*

Falloff is the fade out between the beginning and end of the light cone.

float *Attenuation0***float** *Attenuation1***float** *Attenuation2*

Controls how the light fades over distance.

float *Theta*

Specifies the angle in radians of the spotlight's inner cone.

float *Phi*

Specifies the angle in radians of the spotlight's outer cone.

Once you've created a light, you simply perform two steps, as follows:

1. Direct3D provides several slots to store lights that are to be used in a scene. To tell Direct3D you wish to place a light in the scene, you must call the **SetLight** method of IDirect3DDevice9. This function requires two arguments: the slot number to store the light and a pointer to the light structure itself. You call this function like this:

```
g_pd3dDevice->SetLight(0, &Light);
```

2. Once the light has been placed into a slot, you must then turn the light on. You do this by calling the **LightEnable** method of IDirect3DDevice9, like this:

```
g_pd3dDevice->LightEnable(0, TRUE);
```

Here's an example of how to use a Direct3D light. Take a look at the code below and see how lights work with all the bells and whistles. Next, we'll take a look at how texturing works in Direct3D.

```
D3DXVECTOR3 vecDir;
D3DLIGHT9 light;
ZeroMemory(&light, sizeof(light));
light.Type = D3DLIGHT_DIRECTIONAL;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
vecDir = D3DXVECTOR3(cosf(timeGetTime()/360.0f),
                    0.0f,
                    sinf(timeGetTime()/360.0f));
D3DXVec3Normalize((D3DXVECTOR3*)&light.Direction, &vecDir);
light.Range = 1000.0f;
g_pd3dDevice->SetLight(0, &light);
g_pd3dDevice->LightEnable(0, TRUE);
```

Textures

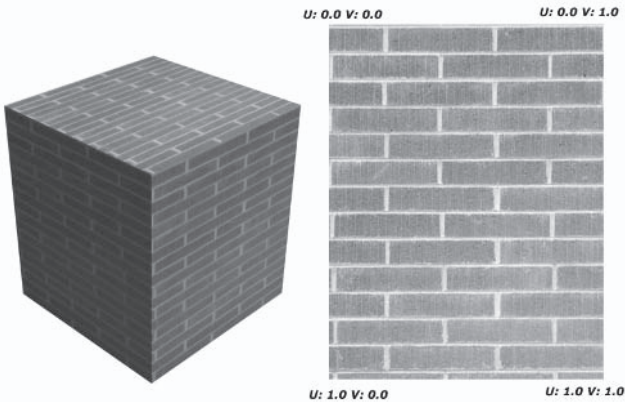


Figure 5.4: Texture mapped cube

Textures are one of the most important features for enhancing realism in a 3D application. In short, *textures* are flat, bitmap images painted onto the surface of one or more polygons. Textures can make your polygons appear to be manufactured from real-life materials such as bricks, metal, wood, etc. Furthermore, textures can be employed in more advanced techniques. For example, textures can be mixed and combined with one another, or they can be used for effects like bump mapping. (A *bump map* use textures combined with other textures and light calculations to simulate bumps and indentations on the surface of polygons, like the bumps on an orange skin.)

Direct3D encapsulates textures using the `IDirect3DTexture9` interface. In reality, Direct3D treats textures very much like surfaces (explained in Chapter 2). In fact, textures are like advanced surfaces and more or less everything you can do with surfaces you can also do with textures. You can load textures from image files like bitmaps or JPEGs, copy pixel data between textures, and copy surfaces and textures to and from one another.

* **NOTE.** Remember, image size is an important factor for textures. Textures should be sized to a power of 2, such as 2x2, 4x4, 8x8, 16x16, etc. Combinations are also allowed, like 16x64. If your textures are not sized properly, they may appear stretched or squeezed in your applications.

Creating Textures

There are many ways to create textures in Direct3D. You can create blank textures of a specified size or you can create textures with images already loaded onto them. Furthermore, as we shall see, Direct3D allows you to choose in which memory the created texture will reside. Textures can be created in standard system memory (RAM), or you can exploit the onboard memory of the system's graphics card.

Let's take a look at two ways you can create textures. Remember, when a texture is created, it starts off as an image held in memory. It's not drawn to the screen and it's not applied to the surface of any polygon straight away. We'll see how to apply a texture to a polygon later in this chapter.

Creating Blank Textures

Creating a blank texture is very much like creating a surface with no image loaded onto it. Effectively, a blank texture is a rectangle of memory filled in with white pixels. Don't worry though; a blank texture doesn't need to stay blank. You can copy image data to it later. To create a blank texture you call the **CreateTexture** method of **IDirect3DDevice9**. This method requires you to pass width and height parameters for the size of the rectangle in pixels. It also requires the format of the texture (like the surface format) and it requires the memory pool in which the texture will be created. The memory pool means the area of memory where the texture will be held, like RAM or on the graphics card. The syntax and parameters for **CreateTexture** are:

```

HRESULT CreateTexture
(
    UINT Width,
    UINT Height,
    UINT Levels,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DTexture9 **ppTexture,
    HANDLE *pSharedHandle
);

```

UINT Width

Width in pixels of the texture.

UINT Height

Height in pixels of the texture.

UINT Levels

Number of levels in the texture. If this is 0, Microsoft Direct3D will generate all texture sublevels down to 1x1 pixels for hardware that supports mipmapped textures.

DWORD Usage

Typically you will pass 0. You can pass one or more of the following values:

```

D3DUSAGE_AUTOGENMIPMAP
D3DUSAGE_DEPTHSTENCIL
D3DUSAGE_DMAP
D3DUSAGE_DONOTCLIP
D3DUSAGE_DYNAMIC
D3DUSAGE_NPATCHES
D3DUSAGE_POINTS
D3DUSAGE_RENDERTARGET
D3DUSAGE_RTPATCHES
D3DUSAGE_SOFTWAREPROCESSING
D3DUSAGE_WRITEONLY

```

D3DFORMAT Format

Image format of the texture. This can be many values, but a typical value is D3DFMT_A8R8G8B8.

D3DPOOL Pool

Specifies the memory pool in which the created texture will be stored. Typically you will pass `D3DPOOL_DEFAULT`. This parameter can be one or more of the following values:

```
D3DPOOL_DEFAULT
D3DPOOL_MANAGED
D3DPOOL_SCRATCH
D3DPOOL_SYSTEMMEM
```

IDirect3DTexture9 **ppTexture

Address to receive a valid texture interface pointer, which represents the created texture.

HANDLE *pSharedHandle

This is a reserved value. Just pass `NULL`.

Creating Textures from Image Files

One of the most common ways to create a texture is from an image file. The image file can be a simple bitmap or JPEG, or you can create textures from more complex formats that contain alpha channels. To create a texture and load an image file onto it you call the **D3DXCreateTextureFromFile** function. This function is simple to use and requires only three parameters: a pointer to the Direct3D device, a valid path to an image file, and an address to receive a pointer to the created texture interface. Let's take a look at this function's syntax.

```
HRESULT WINAPI D3DXCreateTextureFromFile(
    LPDIRECT3DDEVICE9 pDevice,
    LPCTSTR pSrcFile,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```

*** NOTE.** Direct3D provides other functions to load textures that are not covered in this book. These are: `D3DXCreateTextureFromFileEx`, `D3DXCreateTextureFromFileInMemory`, `D3DXCreateTextureFromFileInMemoryEx`, `D3DXCreateTextureFromResource`, and `D3DXCreateTextureFromResourceEx`.

Here's a sample code snippet provided as an example.

```
hr = g_pd3dDevice->CreateTexture(uintWidth, uintHeight, 1,  
                                D3DUSAGE_DYNAMIC,  
                                D3DFMT_X8R8G8B8, D3DPPOOL_DEFAULT,  
                                &m_Texture, NULL);
```

Texture Mapping

The previous sections explained how textures can be created either blank or from image files on disk. Now it's time to see how **textures are applied to the surface of 3D polygons**. This can really bring your 3D world to life. Using textures you can make your polygons appear as though they're made from real materials. For example, a polygon could appear to be brick, grass, or even water. Sounds good? Well, before we examine how textures work practically we'll need to examine the theory behind how textures work in Direct3D.

In Direct3D, textures are applied to polygons through a process called texture mapping. Let's consider the example rectangle in the figure below and imagine this to be a 3D primitive formed by an index buffer with **two triangles aligned side by side**. Thus, this rectangle is defined by four vertices, with a vertex on each corner.

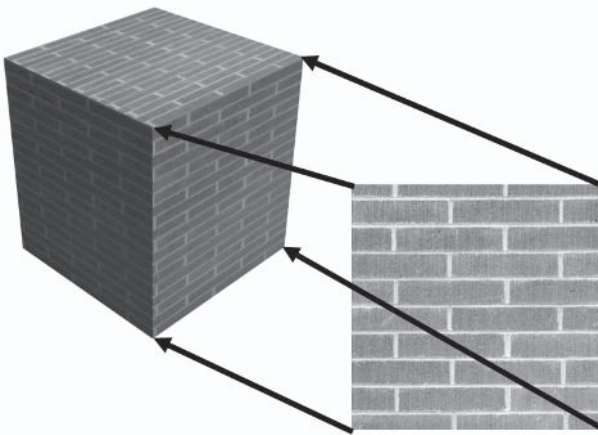


Figure 5.5

Notice that each vertex is encoded with additional information other than just its physical position in 3D space. The additional information is a set of coordinates; for example (0.0, 1.0). These are called *texture coordinates* and are used to tell Direct3D how a texture should be aligned across the surface of a polygon. What does this mean exactly? Well, as a texture is applied to a polygon, Direct3D ties the texture to specific vertices. Take a look at Figure 5.6 to see what's happening between the polygon and the texture.

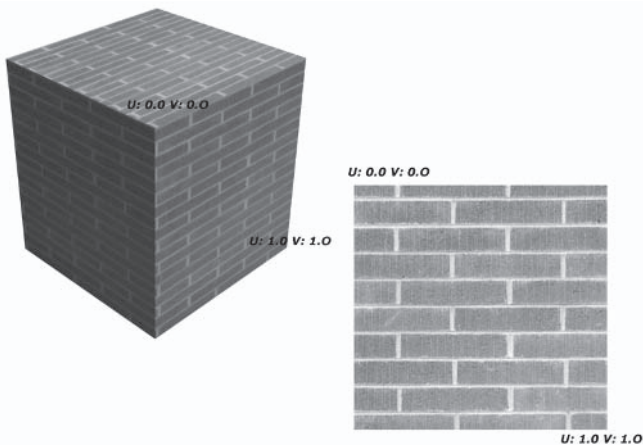


Figure 5.6

As you can see, *texture coordinates* (also called *texels*) take the form of U and V. U specifies the width of a texture and V specifies the height of a texture. You'll probably also notice that texels usually range from 0.0 to 1.0. These values correspond to the proportional dimensions of the texture. In Figure 5.6, (U:0.0 V:0.0) corresponds to the top-left corner of the texture while (U:1.0 V:1.0) represents the bottom-right corner of the texture. Therefore, each vertex specifies which portion of the texture should be overlaid on that part of the polygon's surface.

Let's see how to do this practically in Direct3D. As you may have surmised, texture coordinates are specified in your vertex structure, which was explained earlier. That's right, texture coordinates are added to your vertex structure, which until now has been used to encoded only position and color. The following code shows

a vertex structure containing texture information. Texture coordinates are specified by two floats.

```
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position;    // The position
    D3DCOLOR    color;      // The color
    FLOAT       tu, tv;     // The texture coordinates
};
```

The FVF descriptor for this structure would look something like the following. Notice the inclusion of the additional D3DFVF_TEX1 flag.

```
D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1)
```

Some code to create a rectangle and include texture coordinates might look like this:

```
//Create vertices for rectangle
CUSTOMVERTEX vertices[] =
{
    {50.0f, 50.0f, 0.5f, 0xffff0000, 0.0, 0.0}, // x, y, z
    {250.0f, 250.0f, 0.5f, 0xff00ff00, 0.0, 1.0},
    {50.0f, 250.0f, 0.5f, 0xff00ffff, 1.0, 0.0},
    {250.0f, 50.0f, 0.5f, 0xff00ffff, 1.0, 1.0},
};
```

Setting the Active Texture

The last foundational step you need to know before you can use textures in your applications is how to actually apply them to polygons. You do this very much like setting the active material using `SetMaterial`, as we saw earlier. During the rendering loop you call the **SetTexture** method of `IDirect3DDevice9` to tell Direct3D which texture should be the active texture. This texture will then be applied to whatever vertices are subsequently rendered. The syntax and parameters for this function are as follows.

```
HRESULT SetTexture
(
    DWORD Sampler,
    IDirect3DBaseTexture9 *pTexture
);
```

DWORD Sampler

For the purposes of this book, this value should be 0. By specifying other numbers you can perform various texture effects. However, multitexturing is beyond the scope of this book. Please see the DirectX documentation for more details.

IDirect3DBaseTexture9 *pTexture

Pointer to the texture to be used as the active texture.

Quite simple, really. The code below demonstrates a practical example of how to set an active texture. The following sections explore textures further.

```
g_pd3dDevice->SetTexture(0, g_pTexture);
```

Texture Filtering

Think about this for a moment: You have just created a nice brick texture sized at 256x256 pixels, and now you intend to map it onto a rectangular polygon in order to make it look like a wall. OK, so you've created the vertices and you've set up the texture coordinates so the image will map across the polygon's surface correctly. However, **there's a problem: The rectangle is larger than the brick texture, which means the texture is stretched to fit the rectangle. This makes the texture look blurred and horrible.** Furthermore, when the wall is viewed from an angle, the texture looks grainy and pixelated. This can be quite problematic. **The solution? Texture filtering.**

Texture filtering is the process Direct3D adopts for deciding how textures should be processed when mapped across a polygon's surface. There are several texture filtering modes that Direct3D can use, including nearest-point sampling, linear texture filtering, anisotropic texture filtering, and mipmap filtering.

■ **Nearest-point sampling**

This is the quickest of all filtering modes and should only be used when you know your texture matches the size of the polygon onto which it's mapped, without stretching or shrinking. This is because as your texture is resized or viewed from varying angles, it will appear blocky and blurred due to the quick and messy calculations Direct3D performs.

■ **Linear texture filtering**

Direct3D uses a technique called bilinear filtering for this mode, which offers improved filtering over nearest-point sampling but textures can still suffer from a blurred or blocky appearance.

■ **Anisotropic texture filtering**

The technicalities behind this filtering are beyond the scope of this book. In short, this filtering mode offers some advanced features and good performance. This is a good method for filtering your textures.

■ **Mipmap filtering**

Mipmap filtering works by applying a chain of textures to the same polygon. Each texture is of the same image at a progressively lower resolution than the last. This means Direct3D can juggle between textures as the surface is seen from different angles and distances. This method can be very effective but is computationally expensive.

By default Direct3D uses nearest-point sampling. However, you can set the texture filtering mode by calling the **SetSamplerState** method of **IDirect3DDevice9**. The syntax and parameters for this method are as follows.

```
HRESULT SetSamplerState
(
    DWORD Sampler,
    D3DSAMPLERSTATETYPE Type,
    DWORD Value
);
```

DWORD Sampler

The sampler stage index. For the purpose of this book, you can pass 0.

D3DSAMPLERSTATETYPE Type

This parameter can be many different values. For texture filtering you'll want to pass one of the following:

D3DSAMP_MAGFILTER

Controls the filtering to use for enlarging the texture.

D3DSAMP_MINFILTER

Controls the filtering to use for shrinking the texture.

D3DSAMP_MIPFILTER

Controls the filtering to use for mipmap filtering.

DWORD Value

This value selects the type of filtering to use. This can be one of the following:

D3DTEXF_NONE

No filtering

D3DTEXF_POINT

Nearest-point sampling

D3DTEXF_LINEAR

Linear texture filtering

D3DTEXF_ANISOTROPIC

Anisotropic texture filtering

As you can probably tell, this parameter controls many features of Direct3D, one of which we'll see shortly. First, however, let's see an example of how to set the texture filtering mode. It's very simple to do. Take a look at the code below.

```
hr = g_pd3dDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
```

Texture Addressing Modes

As explained previously, texture coordinates (UV) typically fall within the range from 0.0 to 1.0, which corresponds to the proportional extents of a texture. This influences how the surface of a polygon is mapped by a texture. However, texture coordinates need not be just values from 0.0 to 1.0; they could, for example, be higher than 1.0. What happens if a texture coordinate is 2.0? Well, a number of different things could happen, depending on which texture addressing mode Direct3D is using.

Direct3D uses a texture addressing mode to handle textures when vertices have texture coordinates outside the typical range. Using texture addressing you can perform various effects with textures, like mirroring and tiling. The various texture addressing modes that Direct3D offers are: wrap texture address mode, mirror texture address mode, clamp texture address mode, and border color texture address mode.

■ Wrap texture address mode

This texture mode tiles the texture when it encounters texture coordinates outside the range of 0.0 to 1.0. For example, if a rectangle has the texture coordinate (U:2.0 V:0.0) for its rightmost vertex, then the texture will be tiled twice across the width of the polygon.

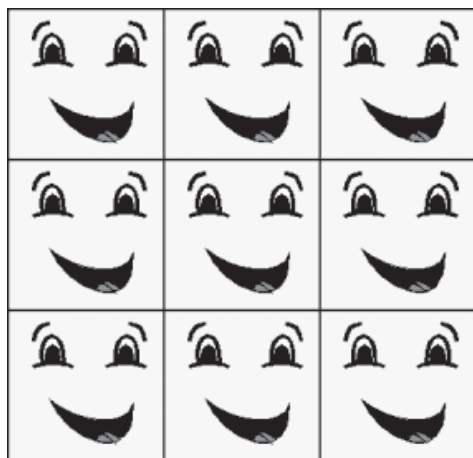


Figure 5.7: Wrap texture address mode

■ Mirror texture address mode

Like wrap texture address mode, mirror texture address mode will tile the texture across the polygon, except it will mirror the texture on each consecutive tile. See Figure 5.8.

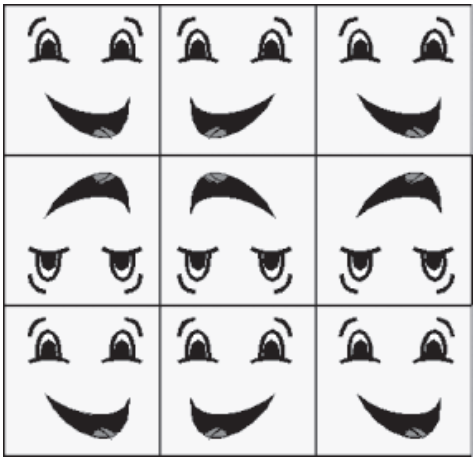


Figure 5.8: Mirror texture address mode

■ Clamp texture address mode

With clamp texture address mode, Direct3D takes the last column and row of pixels in the texture and drags them across the width and height of the polygon. Take a look at Figure 5.9.

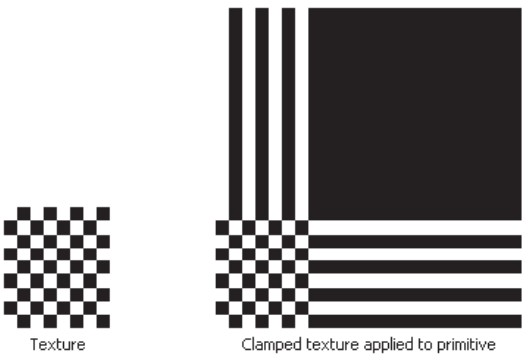


Figure 5.9: Clamp texture address mode

To set the texture addressing mode in Direct3D you call the **SetSamplerState** method of the **IDirectDevice9** interface. This method was introduced in the previous section and will be used

again here to set the texture addressing mode. In this case, the syntax and parameters for `SetSamplerState` follow.

```
HRESULT SetSamplerState
(
    DWORD Sampler,
    D3DSAMPLERSTATETYPE Type,
    DWORD Value
);
```

DWORD Sampler

The sampler stage index. For the purpose of this book, you can pass 0.

D3DSAMPLERSTATETYPE Type

This parameter can be many different values. For texture addressing you'll want to pass one of the following:

D3DSAMP_MAGFILTER

Controls the filtering to use for enlarging the texture.

D3DSAMP_MINFILTER

Controls the filtering to use for shrinking the texture.

D3DSAMP_MIPFILTER

Controls the filtering to use for mipmap filtering.

DWORD Value

This value selects the type of filtering to use. This can be one of the following:

D3DTEXF_NONE

No filtering

D3DTEXF_POINT

Nearest-point sampling

D3DTEXF_LINEAR

Linear texture filtering

D3DTEXF_ANISOTROPIC

Anisotropic texture filtering

Here's an example of how to set the texture addressing mode to wrap:

```
g_pDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_ANISOTROPIC);
g_pDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_ANISOTROPIC);
```

Texture Alpha Blending

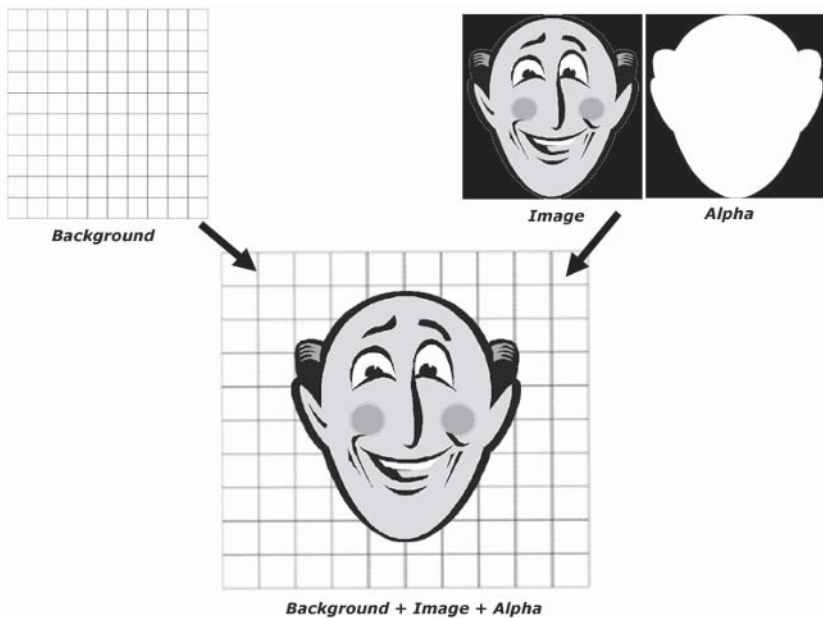


Figure 5.10

It's certainly worth mentioning that various image formats, like Targa and PNG, support alpha channels to define transparent, semi-transparent, or opaque regions in an image. **An alpha channel is a separate grayscale image that is attached to a normal image of the same size and dimensions.** Each pixel in the alpha channel corresponds to a pixel in the normal image and defines whether the pixel should be transparent, semi-transparent, or opaque. **Black pixels are transparent, white pixels are opaque, and gradations in between represent degrees of visibility** that become less visible the closer it comes to black. Thus, alpha channels can be used to create transparent effects with textures since Direct3D supports alpha

blending. *Alpha blending* is the process whereby Direct3D takes the alpha channel of an image and applies the transparency information to the final texture. Thankfully, Direct3D does all the hard work for us. All we need to do is enable alpha blending. To do this we call the `SetRenderState` function, as explained earlier. The code to enable and disable alpha blending appears as follows.

```
m_pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
m_pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
m_pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

* **NOTE.** When alpha blending isn't needed, be sure to disable it for an overall performance gain.

2D Texturing

So far this chapter has spoken about applying textures to polygons in 3D space. However, what if we wanted to create a user interface for a game, draw energy panels on the screen, or render a gauge that displays a player's remaining ammo? These kinds of graphics don't exist in 3D space. Instead, they're little widgets that display information in 2D screen space. Thus, it'll be useful to know how we can render textures directly to the screen in 2D. Furthermore, we'll be able to retain the benefits of textures.

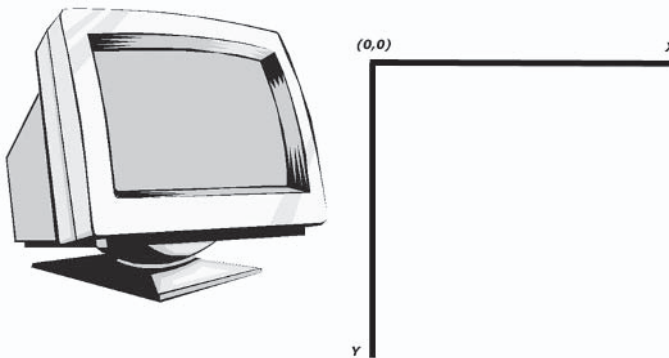


Figure 5.11

First, let's recap screen coordinates, as shown in Figure 5.11. Screen space has two axes: X and Y. These are measured in pixels. The origin for these axes is in the top-left corner of the screen, which is (0, 0).

Drawing a texture in screen space is simple and can be achieved in Direct3D using various methods. This book concentrates on using the ID3DXSprite interface to draw textures in 2D screen space. This interface acts very much like a GDI pen, or any normal pen for that matter. Before we examine how this interface is used, let's see how it's created.

To create a valid instance of ID3DXSprite you call the **D3DXCreateSprite** function. It requires two parameters: a pointer to a Direct3D device and an address to receive a valid ID3DXSprite interface. The syntax and parameters follow. Remember, this function simply creates a sprite interface in memory. The process of drawing textures is performed in the next section.

```
HRESULT WINAPI D3DXCreateSprite
(
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXSPRITE *ppSprite
);
```

LPDIRECT3DDEVICE9 pDevice

Pointer to a Direct3D device.

LPD3DXSPRITE *ppSprite

Address to receive an ID3DXSprite interface.

Here's an example of how to create a sprite interface:

```
LPD3DXSPRITE Sprite = NULL;
D3DXCreateSprite(&Sprite);
```

ID3DXSprite — Drawing Textures in 2D

As the previous section explained, **ID3DXSprite is like a pen. You simply pass it a texture and it will draw it in 2D screen space.** This is useful for the various kinds of interface widgets your game might have. To draw a texture in 2D screen space you call a number of ID3DXSprite methods during your application's rendering loop. Here's how the process works:

1. You start by calling the **Begin** method of ID3DXSprite. This prepares the interface for drawing a texture to the screen. This function requires only one parameter. Typically you will pass 0; however, for more advanced features, you can pass other values. The syntax and parameters for this function follow.

```
HRESULT Begin
(
    DWORD Flags
);
```

DWORD Flags

Can be 0 or a combination of one or more of the following flags. Please see the DirectX SDK documentation for more information.

```
D3DXSPRITE_ALPHABLEND
D3DXSPRITE_BILLBOARD
D3DXSPRITE_DONOTMODIFY_RENDERSTATE
D3DXSPRITE_DONOTSAVESTATE
D3DXSPRITE_OBJECTSPACE
D3DXSPRITE_SORT_DEPTH_BACKTOFRONT
D3DXSPRITE_SORT_DEPTH_FRONTTOBACK
D3DXSPRITE_SORT_TEXTURE
```

2. After Begin is called you'll want to call the SetTransform method of ID3DXSprite. This method accepts a D3DXMATRIX. This matrix should represent a 2D transformation that is to be applied to the drawn texture. In other words, this matrix

will tell ID3DXSprite how to draw a specified texture. We might want to translate it (5,5) from the screen origin, or we might want the texture rotated or scaled. Until now we have only learned how to build matrices that encode 3D transformations in Direct3D. However, Direct3D provides a function to assemble a 2D matrix that we can later pass to the SetTransform method. This function is called **D3DXMatrixTransformation2D**. Its syntax and parameters follow.

```
D3DXMATRIX *WINAPI D3DXMatrixTransformation2D
(
    D3DXMATRIX *pOut,
    CONST D3DXVECTOR2 *pScalingCenter,
    FLOAT *pScalingRotation,
    CONST D3DXVECTOR2 *pScaling,
    CONST D3DXVECTOR2 *pRotationCenter,
    FLOAT Rotation,
    CONST D3DXVECTOR2 *pTranslation
);
```

D3DXMATRIX *pOut

Address to receive a resultant 2D matrix.

CONST D3DXVECTOR2 *pScalingCenter

A D3DXVECTOR2 structure representing the center point at which scaling occurs. If scaling is not required, this parameter can be NULL.

FLOAT *pScalingRotation

Pointer to the scaling rotation factor. If scaling is not required, this parameter can be NULL.

CONST D3DXVECTOR2 *pScaling

A D3DXVECTOR2 structure defining the scaling factor. If scaling is not required, this parameter can be NULL.

CONST D3DXVECTOR2 *pRotationCenter

A D3DXVECTOR2 structure specifying the center of rotation. If rotation is not required, this value can be NULL.

FLOAT *Rotation*

Angle in radians to rotate. If you don't want to perform rotation, just pass 0.

CONST D3DXVECTOR2 **pTranslation*

A D3DXVECTOR2 structure representing the number of pixels to translate from the origin. If translation is not required, this parameter can be NULL.

Once you have built a 2D transformation matrix you can then pass this onto SetTransform. Like this:

```
D3DXMatrixTransformation2D(&Mat, NULL, 0, NULL, NULL, 0, NULL);
Sprite->SetTransform(&Mat);
```

- Now that you've prepared the interface for drawing and specified a transformation for the texture, you can call the **Draw** method of the ID3DXSprite interface to actually draw the texture to the screen. You can pass translation and rotation values to this function too, but since we provided these via a transformation matrix in step 2, we can set these values to NULL. The syntax and parameters for Draw follow.

HRESULT Draw

```
(
    LPDIRECT3DTEXTURE9 pTexture,
    CONST RECT *pSrcRect,
    CONST D3DXVECTOR3 *pCenter,
    CONST D3DXVECTOR3 *pPosition,
    D3DCOLOR Color
);
```

LPDIRECT3DTEXTURE9 *pTexture*

Pointer to a texture to draw.

CONST RECT **pSrcRect*

Pointer to a rectangle on the texture to draw. This can be NULL.

CONST D3DXVECTOR3 **pCenter*

Pointer to a vector marking the center of the sprite. This can be NULL.

CONST D3DXVECTOR3 *pPosition

Pointer to a vector representing the translation. This can be NULL.

D3DCOLOR Color

Using this value you can tint the texture a specific color or you can change its alpha value.

4. Finally, you call the End method of ID3DXSprite to complete the texture drawing process. This method requires no arguments. The entire drawing process can be summarized in the following code.

```
D3DXMATRIX Mat;
D3DXMatrixTransformation2D(&Mat, NULL, 0, NULL, NULL, 0, NULL);
Sprite->Begin(0);
Sprite->SetTransform(&Mat);
Sprite->Draw(Texture, NULL, NULL, NULL, 0xFFFFFFFF);
Sprite->End();
```

And that, ladies and gentlemen, is texture drawing. Simple, I'm sure you'll agree.

Conclusion

This chapter has presented an introduction to lighting, materials, and textures in Direct3D. Hopefully, having understood the content presented here, you'll no doubt realize the awesome potential at your fingertips. At this point I recommend going back through the chapter and playing with some code. Try tweaking some of the arguments you pass to the functions and see what happens. Then try moving lights in real time and see what happens.

The next chapter will take you a step further on our journey of Direct3D as we consider data storage and specifically X files.



Chapter 6

X Files — Loading and Saving Data

As any good programmer is likely to know, there are many ways to store persistent data. Games store their levels and meshes in files that are read in at run time. A graphic artist models 3D characters and saves them in a file, and then this data is loaded into DirectX by the programmer. Now, you could reinvent the wheel and design your own proprietary file format for all your application's data, or you could use X files, which is a file storage system provided with the DirectX SDK. X files allow you to store and load any kind of data of any size. Using them, or at least understanding how they work, makes your life easier because many of the DirectX mesh saving plug-ins (for exporting models from 3D packages) will save the mesh data into an X file. Beyond this, X files offer many benefits for storing your own game data. This chapter explains how to use X files. Specifically it explains how to:

- Load data from X files
- Enumerate data in X files
- Save data to X files

Introduction to X Files

Imagine for a moment you're making a first-person shooter game where players run around an environment and shoot baddies. Typically, you'd store each level in a data file that your game will read at run time. There is a lot of data to store. Usually, the levels themselves — walls, doors, floors, etc. — will be stored as a mesh. Remember, a mesh is just a complex collection of polygons. And the baddies themselves would also be meshes. Additionally, you'll want to store miscellaneous information, like the XYZ positions of power-ups, medi-kits, and ammo. You'll also want to know which parts of the floor are walkable and which are traps, and it'll be useful to know the locations to which teleporters will take you when activated. The list goes on. You could design your own file format to store all this data, or you could use a mechanism already made. Enter X files...

X files are so named because of their `.x` file extension. It's an open-ended file format, which means X files can be of any length and can store any kind of data. Furthermore, your data can be stored in one of two ways: binary mode or text mode. So X files can either be a bunch of unreadable bytes or they can be textual data, the kind you can open up and edit in a text editor.

To save and load information to and from an X file, you use a number of DirectX interfaces. Using these, every item of data you write to an X file — like your meshes, power-ups, starting health, etc. — will be written as a self-contained object, and these objects are contained inside the X file in a tree structure, much like an XML file or the files and folders on your hard disk. This kind of structure is called a *hierarchy*. In other words, any object inside an X file can have none, one, or more than one child objects. Don't worry though; DirectX provides all the methods and interfaces you'll need to enumerate and cycle through objects in an X file. Let's begin by taking a look at what an X file (text version) actually looks like:

```

xof 0302txt 0032

//Declaration

template MY_GAME_INFO{
    <AA1308FD-FF98-4f6e-9A55-CD083178672F>
    STRING GameName;
    STRING GameMaker;
    DWORD Version;
}

template MY_LEVEL{
    <0FC92315-6897-4f03-B2BD-A6CE20065861>
    STRING LevelName;
    [...]
}

template MY_MEDI_KIT{
    <DACCED4A-433E-4fa3-91A6-2A8EA6B6D090>
    DWORD XPos;
    DWORD YPos;
    DWORD ZPos;
}

//Definition

MY_GAME_INFO Game01
{
    "My Test Game";
    "Alan Thorn";
    1;
}

MY_LEVEL Level01
{
    "Level01";
    MY_MEDI_KIT
    {
        5;
        7;
    }
}

```

```
    13;  
}  
MY_MEDI_KIT  
{  
    435;  
    757;  
    139;  
}  
}
```

Structural Overview

The X file example in the previous section let you see what an X file looks like inside. In many ways it follows the typical C style syntax; however, there are differences. The structure of an X file is explained in the following subsections.

Header

You'll notice that each X file begins with a header. This is a line stating the version number and file mode (text or binary). It looks like the following. When creating your own X files you'll simply need to copy and paste this line at the top of each file.

```
xof 0302txt 0032
```

Templates

The next section of an X file is the declarations section. This is where you define all the data types that will appear in your file. Each definition is called a template. It's like writing ordinary classes. The template defines how your data is structured. So in this example our file will be storing data about a number of things; specifically, game info, level info, and medi-kits. The templates for these tell DirectX what kind of data our objects are made of. The game info template is defined like this:

```
template MY_GAME_INFO{
    <AA1308FD-FF98-4f6e-9A55-CD083178672F>
    STRING GameName;
    STRING GameMaker;
    DWORD Version;
}
```

***NOTE.** You'll also notice a line preceded by `///
comment, like in C++, and is ignored by DirectX. It's simply there for reference. You can put comments anywhere you want in the file. Comments can also be written by using # instead of ///.`

To define a template, you start by using the keyword “template,” followed by the name of your template. In this example, the name is `MY_GAME_INFO`. This is the name you will use later in the file when creating instances of your template. Then there's an opening brace that is partnered by a closing brace. These mark the beginning and end of your template declaration. Between them you declare whatever data types form your template. The first line is different, however. The first line is a GUID (globally unique identifier), which is a long bunch of numbers and letters that allows DirectX to distinguish one template from another. The GUID will also be used to identify your templates when coding DirectX applications. More on how to generate these numbers later.

After the GUID, you add whatever members you need to your template, just like class members. The kinds of data types available are very similar to C++. You'll notice our example `MY_GAME_INFO` holds three members: two strings, which hold the game name and the creator's name, and one integer to keep track of the current game version. The list below shows the various data types and reserved words available for use in X files.

ARRAY	STRING
BINARY	SWORD
BINARY_RESOURCE	TEMPLATE
CHAR	UCHAR
CSTRING	ULONGLONG
DOUBLE	UNICODE
DWORD	WORD
SDWORD	

In addition to the standard data types you'll also notice the [...] symbol in the template declaration for medi-kits. This is like a placeholder character. It details whether any objects can be embedded inside this object. By specifying [...] you're effectively saying, "Hey, instances of this template are going to have child objects. I don't know how many or what data type; it could be any number and any data type." Templates that can contain child objects are called *open templates* and templates that cannot are called *closed templates*. You'll get a feel for exactly how this works when you look further down in the file and see the templates in action. This is explained in more detail in the next section.

Data Objects

Data objects make up the rest of an X file and are the main body. This is where you actually store your own data, using objects based upon your templates. The definition for an object based upon the example MY_GAME_INFO template looks like this:

```
MY_GAME_INFO Game01
{
    "My Test Game";
    "Alan Thorn";
    1;
}
```

Simple. To create an object based upon a template you first write the template name, in this case MY_GAME_INFO. You can then give the object a name by which it can be identified throughout the file. The name is optional though. In this case I have used the name Game01. You then use the opening and closing braces to mark the beginning and end of the definition. Between them you define the object's properties line by line, each value corresponding to the order you defined in the template. So the top member is game name, which for object Game01 is "My Test Game." Remember to use the " " symbols for strings; numbers and other objects do not require these symbols. Also, be sure to end each line using a semicolon, like in C++.

Parent and Child Objects

I mentioned earlier that X file objects can be stored in a hierarchy, just like XML files or the files and folders on your hard disk. This means that objects can contain other objects. We've already seen that by using the [...] symbol we can create a template for objects to own none, one, or more children. The following extract from the example X file shows how a level object contains several medi-kits. These medi-kits are separate objects stored as child objects of the level. You'll notice it's exactly the same as creating any other object, except child objects are declared between the opening and closing braces of the parent object.

```
MY_LEVEL Level01
{
    "Level01";
    MY_MEDI_KIT
    {
        5;
        7;
        13;
    }
    MY_MEDI_KIT
    {
        435;
        757;
        139;
    }
}
```

Data Objects and References

The final object type to consider is the reference. This is like a pointer to an object elsewhere in the file. A parent can contain pointers to objects as its children, but it does not necessarily own the object that is being referenced. The reference simply refers to any object somewhere in the file. The examples we have been working with thus far do not contain an example of references, but they can be defined like this:

```
MY_GAME_INFO Game01
{
    "My Test Game";
    "Alan Thorn";
    1;
}

MY_LEVEL Level01
{
    "Level01";
    MY_MEDI_KIT
    {
        5;
        7;
        13;
    }
    MY_MEDI_KIT
    {
        435;
        757;
        139;
    }
    [Game01]
}
```

The reference [Game01] is a child of a level object but it references the MY_GAME_INFO object further up in the file. As you can see, it references an object by name. Later, when we cycle through all the objects in an X file using the DirectX SDK, we'll need to check each object to assess whether it is a standard object or a reference to another object.

Standard Templates

Now that you've got an idea how the X file template system works, you'll be interested to know what kind of premade templates DirectX provides for you. The standard templates and examples of their syntax are listed below. For now we won't be examining how to use the standard templates specifically, although some are covered later. However, it's good to know them and will provide further insight into how you can define your own templates, which is something we'll do later in this chapter.

Animation	IndexedColor
AnimationKey	Material
AnimationOptions	MaterialWrap
AnimationSet	Matrix4x4
AnimTicksPerSecond	Mesh
Boolean	MeshFace
Boolean2d	MeshFaceWraps
ColorRGB	MeshMaterialList
ColorRGBA	MeshNormals
Coords2d	MeshTextureCoords
DeclData	MeshVertexColors
EffectDWord	Patch
EffectFloats	PatchMesh
EffectInstance	PatchMesh9
EffectParamDWord	PMAAttributeRange
EffectParamFloats	PMInfo
EffectParamString	PMVSplitRecord
EffectString	SkinWeights
FaceAdjacency	TextureFilename
FloatKeys	TimedFloatKeys
Frame	Vector
FrameTransformMatrix	VertexDuplicationIndices
FVFData	VertexElement
Guid	XSkinMeshHeader


```

template Animation
{
    < 3D82AB4F-62DA-11cf-AB39-0020AF71E433 >
    [...]
}

template AnimationKey
{
    < 10DD46A8-775B-11CF-8F52-0040333594A3 >
    DWORD keyType;
    DWORD nKeys;
    array TimedFloatKeys keys[nKeys];
}

template AnimationOptions
{
    < E2BF56C0-840F-11cf-8F52-0040333594A3 >
    DWORD openclosed;
    DWORD positionquality;
}

template AnimationSet
{
    < 3D82AB50-62DA-11cf-AB39-0020AF71E433 >
    [ Animation < 3D82AB4F-62DA-11cf-AB39-0020AF71E433 > ]
}

template AnimTicksPerSecond
{
    < 9E415A43-7BA6-4a73-8743-B73D47E88476 >
    DWORD AnimTicksPerSecond;
}

template Boolean
{
    < 537da6a0-ca37-11d0-941c-0080c80cfa7b >
    DWORD truefalse;
}

template Boolean2d

```

```

    {
        < 4885AE63-78E8-11cf-8F52-0040333594A3 >
        Boolean u;
        Boolean v;
    }

template ColorRGB
{
    < D3E16E81-7835-11cf-8F52-0040333594A3 >
    float red;
    float green;
    float blue;
}

template ColorRGBA
{
    < 35FF44E0-6C7C-11cf-8F52-0040333594A3 >
    float red;
    float green;
    float blue;
    float alpha;
}

template Coords2d
{
    < F6F23F44-7686-11cf-8F52-0040333594A3 >
    float u;
    float v;
}

template DeclData
{
    < BF22E553-292C-4781-9FEA-62BD554BDD93 >
    DWORD nElements;
    array VertexElement Elements[nElements];
    DWORD nDWords;
    array DWORD data[nDWords];
}

template EffectDWord

```

```

    {
        < 622C0ED0-956E-4da9-908A-2AF94F3CE716 >
        DWORD Value;
    }

template EffectFloats
{
    < F1CFE2B3-ODE3-4e28-AFA1-155A750A282D >
    DWORD nFloats;
    array float Floats[nFloats];
}

template EffectInstance
{
    < E331F7E4-0559-4cc2-8E99-1CEC1657928F >
    STRING EffectFilename;
    [ ... ]
}

template EffectParamDWord
{
    < E13963BC-AE51-4c5d-B00F-CFA3A9D97CE5 >
    STRING ParamName;
    DWORD Value;
}

template EffectParamFloats
{
    < 3014B9A0-62F5-478c-9B86-E4AC9F4E418B >
    STRING ParamName;
    DWORD nFloats;
    array float Floats[nFloats];
}

template EffectParamString
{
    < 1DBC4C88-94C1-46ee-9076-2C28818C9481 >
    STRING ParamName;
    STRING Value;
}

```

```

template EffectString
{
    < D55B097E-BDB6-4c52-B03D-6051C89D0E42 >
    STRING Value;
}

template FaceAdjacency
{
    < A64C844A-E282-4756-8B80-250CDE04398C >
    DWORD nIndices;
    array DWORD indices[nIndices];
}

template FloatKeys
{
    < 10DD46A9-775B-11cf-8F52-0040333594A3 >
    DWORD nValues;
    array float values[nValues];
}

template Frame
{
    < 3D82AB46-62DA-11CF-AB39-0020AF71E433 >
    [...]
}

template FrameTransformMatrix
{
    < F6F23F41-7686-11cf-8F52-0040333594A3 >
    Matrix4x4 frameMatrix;
}

template FVFData
{
    < B6E70A0E-8EF9-4e83-94AD-ECC8B0C04897 >
    DWORD dwFVF;
    DWORD nDWords;
    array DWORD data[nDWords];
}

```

```

template Guid
{
    < a42790e0-7810-11cf-8f52-0040333594a3 >
    DWORD data1;
    WORD data2;
    WORD data3;
    array UCHAR data4[8];
}

template IndexedColor
{
    < 1630B820-7842-11cf-8F52-0040333594A3 >
    DWORD index;
    ColorRGBA indexColor;
}

template Material
{
    < 3D82AB4D-62DA-11CF-AB39-0020AF71E433 >
    ColorRGBA faceColor;
    FLOAT power;
    ColorRGB specularColor;
    ColorRGB emissiveColor;
    [...]
}

template MaterialWrap
{
    < 4885ae60-78e8-11cf-8f52-0040333594a3 >
    Boolean u;
    Boolean v;
}

template Matrix4x4
{
    < F6F23F45-7686-11cf-8F52-0040333594A3 >
    array float matrix[16];
}

```

```

template Mesh
{
    < 3D82AB44-62DA-11CF-AB39-0020AF71E433 >
    DWORD nVertices;
    array Vector vertices[nVertices];
    DWORD nFaces;
    array MeshFace faces[nFaces];
    [...]
}

template MeshFace
{
    < 3D82AB5F-62DA-11cf-AB39-0020AF71E433 >
    DWORD nFaceVertexIndices;
    array DWORD faceVertexIndices[nFaceVertexIndices];
}

template MeshFaceWraps
{
    < ED1EC5C0-C0A8-11D0-941C-0080C80CFA7B >
    DWORD nFaceWrapValues;
    array Boolean2d faceWrapValues[nFaceWrapValues];
}

template MeshMaterialList
{
    < F6F23F42-7686-11CF-8F52-0040333594A3 >
    DWORD nMaterials;
    DWORD nFaceIndexes;
    array DWORD faceIndexes[nFaceIndexes];
    [Material <3D82AB4D-62DA-11CF-AB39-0020AF71E433>]
}

template MeshNormals
{
    < F6F23F43-7686-11cf-8F52-0040333594A3 >
    DWORD nNormals;
    array Vector normals[nNormals];
    DWORD nFaceNormals;
    array MeshFace meshFaces[nFaceNormals];
}

```

```

    }

template MeshTextureCoords
{
    < F6F23F40-7686-11cf-8F52-0040333594A3 >
    DWORD nTextureCoords;
    array Coords2d textureCoords[nTextureCoords];
}

template MeshVertexColors
{
    < 1630B821-7842-11cf-8F52-0040333594A3 >
    DWORD nVertexColors;
    array IndexColor vertexColors[nVertexColors];
}

template Patch
{
    < A3EB5D44-FC22-429D-9AFB-3221CB9719A6 >
    DWORD nControlIndices;
    array DWORD controlIndices[nControlIndices];
}

template PatchMesh
{
    < D02C95CC-EDBA-4305-9B5D-1820D7704BBF >
    DWORD nVertices;
    array Vector vertices[nVertices];
    DWORD nPatches;
    array Patch patches[nPatches];
    [ ... ]
}

template PatchMesh9
{
    < B9EC94E1-B9A6-4251-BA18-94893F02C0EA >
    DWORD Type;
    DWORD Degree;
    DWORD Basis;
    DWORD nVertices;

```

```

        array Vector vertices[nVertices];
        DWORD nPatches;
        array Patch patches[nPatches];
        [ ... ]
    }

template PMAttributeRange
{
    < 917E0427-C61E-4a14-9C64-AFE65F9E9844 >
    DWORD iFaceOffset;
    DWORD nFacesMin;
    DWORD nFacesMax;
    DWORD iVertexOffset;
    DWORD nVerticesMin;
    DWORD nVerticesMax;
}

template PMInfo
{
    < B6C3E656-EC8B-4b92-9B62-681659522947 >
    DWORD nAttributes;
    array PMAttributeRange attributeRanges[nAttributes];
    DWORD nMaxValence;
    DWORD nMinLogicalVertices;
    DWORD nMaxLogicalVertices;
    DWORD nVSplits;
    array PMVSplitRecord splitRecords[nVSplits];
    DWORD nAttributeMispredicts;
    array DWORD attributeMispredicts[nAttributeMispredicts];
}

template PMVSplitRecord
{
    < 574CCC14-F0B3-4333-822D-93E8A8A08E4C >
    DWORD iFaceCLW;
    DWORD iV1rOffset;
    DWORD iCode;
}

template SkinWeights

```



```

    {
        < 6F0D123B-BAD2-4167-A0D0-80224F25FABB >
        STRING transformNodeName;
        DWORD nWeights;
        array DWORD vertexIndices[nWeights];
        array float weights[nWeights];
        Matrix4x4 matrixOffset;
    }

template TextureFilename
{
    < A42790E1-7810-11cf-8F52-0040333594A3 >
    string filename;
}

template TimedFloatKeys
{
    < F406B180-7B3B-11cf-8F52-0040333594A3 >
    DWORD time;
    FloatKeys tfkeys;
}

template Vector
{
    < 3D82AB5E-62DA-11cf-AB39-0020AF71E433 >
    float x;
    float y;
    float z;
}

template VertexDuplicationIndices
{
    < B8D65549-D7C9-4995-89CF-53A9A8B031E3 >
    DWORD nIndices;
    DWORD nOriginalVertices;
    array DWORD indices[nIndices];
}

template VertexElement
{

```

```

    < F752461C-1E23-48f6-B9F8-8350850F336F >
    DWORD Type;
    DWORD Method;
    DWORD Usage;
    DWORD UsageIndex;
}

template XSkinMeshHeader
{
    < 3CF169CE-FF7C-44ab-93C0-F78F62D172E2 >
    WORD nMaxSkinWeightsPerVertex;
    WORD nMaxSkinWeightsPerFace;
    WORD nBones;
}

```

Custom Templates

Probably one of the main reasons you'll use X files is to store your own data. To do this you'll need to write your own templates, which is a really simple thing to do now that you know how they work. However, you may be wondering how to generate a valid unique number, the GUID, for your templates. There are many applications around that can generate this unique number for you at the touch of a button. One of them is GUIDgen (guidgen.exe), a small utility that comes with Microsoft Visual Studio. It can be found in the common\tools folder where you installed Visual Studio. Give it a

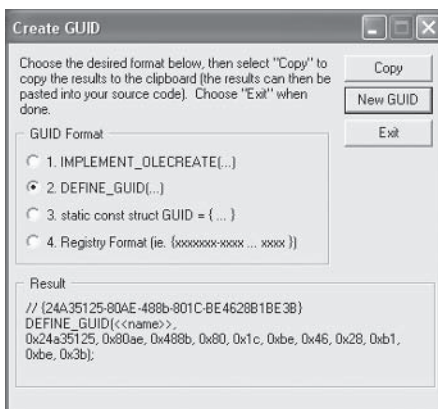


Figure 6.1

run, click the option for `DEFINE_GUID`, and then click the New GUID button. Finally, click the Copy button and you will have copied a new unique ID to the clipboard. All you need to do now is paste that ID into your template declaration and that's it; simple. Take a look at Figure 6.1.

Reading X Files Using DirectX

The best way to start using X files is to take an existing X file and have your application read data from it. Conveniently, DirectX provides a number of interfaces to read X file data for you. However, before you can compile your apps to use them, you'll need to add the following headers and libs to your C++ project.

Libs

`d3dxof.lib`
`dxguid.lib`

Includes

`dxfile.h`
`initguid.h`
`rmxftmpl.h`
`rmxfguid.h`

Preparing

To start working with X files — both reading and writing — you need to create an `ID3DXFile` interface. This interface will act like a manager that oversees the whole reading process. To create an instance of this interface you'll need to call the `D3DXFileCreate` function. You don't specify any actual filenames at this point; that will occur later. The syntax and parameter for `D3DXFileCreate` follow, and then an example.

```
STDAPI D3DXFileCreate
(
    LPD3DXFILE *lpIplDirectXFile
);
```

LPD3DXFILE *lpD3DXFile

Address to where an ID3DXFile pointer, representing a valid object, is to be returned.

Sample code:

```
LPDIRECTXFILE File = NULL;
if(FAILED(D3DXFileCreate(&File)))
    return;
```

Registering Templates

Once an X file object has been created you *may* need to tell DirectX which templates your files will be using. Although I didn't mention it earlier, it's possible to have an X file without any template declarations at the top. This is because you could have many different X files using the same templates, which would mean you'd have many files with duplicate template declarations. To avoid this waste of space, DirectX gives you the choice to omit template declarations from your X files. However, this means as you load those files in your app later, you'll need to manually tell DirectX which templates you use. On the other hand, if you include template declarations at the tops of all your X files, as shown in the previous sections, you won't need to register templates since DirectX does this automatically. If this is the case, you can skip this section.

To register one or more templates you use the **RegisterTemplates** method of ID3DXFile. Its syntax and parameters are shown below, then an example follows.

```
HRESULT RegisterTemplates
(
    LPVOID pvData,
    DWORD cbSize
);
```

LPVOID *pvData*

Pointer to a string that contains the X file templates to register. This string will be exactly how the templates appear in the X file.

DWORD *cbSize*

Size of string in bytes.

An example to register all the templates in our sample file shown earlier would look as follows:

```
char *szTemplates = "xof 0302txt 0032\
template MY_GAME_INFO{\
    <AA1308FD-FF98-4f6e-9A55-CD083178672F>\
    STRING GameName;\
    STRING GameMaker;\
    DWORD Version;\
}\
template MY_LEVEL{\
    <0FC92315-6897-4f03-B2BD-A6CE20065861>\
    STRING LevelName;\
    [...] \
}\
template MY_MEDI_KIT{\
    <DACCED4A-433E-4fa3-91A6-2A8EA6B6D090>\
    DWORD XPos;\
    DWORD YPos;\
    DWORD ZPos;\
}";
```

File -> **RegisterTemplates**(szTemplates, strlen(szTemplates));

*** NOTE.** DirectX provides a premade buffer to register the standard templates shown in an earlier section. To register these templates you can call RegisterTemplates with the following params:

```
File ->RegisterTemplates((LPVOID)D3DRM_XTEMPLATES,
                        D3DRM_XTEMPLATE_BYTES);
```

Opening a File

Once an ID3DXFile object has been created and after any templates have been registered, you can then open an X file from disk to start reading. To do this, you call the **CreateEnumObject** method of ID3DXFile. This function returns a pointer to an ID3DXFile-EnumObject interface, which represents the data in your file. The next section will show you how to cycle through this data and read from it. For now, though, let's take a look at the syntax and parameters for CreateEnumObject and then see an example of how to use it.

```
HRESULT CreateEnumObject
(
    LPVOID pvSource,
    DXFILELOADOPTIONS dwLoadOptions,
    ID3DXFileEnumObject **ppEnumObj
);
```

LPVOID pvSource

The source for the file. This can be a filename, memory address, or something else. In most cases it will be a filename.

DXFILELOADOPTIONS dwLoadOptions

Specifies where the file is loaded from. This value can be one of the following:

```
DXFILELOAD_FROMFILE
DXFILELOAD_FROMMEMORY
DXFILELOAD_FROMRESOURCE
DXFILELOAD_FROMSTREAM
DXFILELOAD_FROMURL
```

ID3DXFileEnumObject *ppEnumObj

Address of an enum pointer to where a valid object is returned. This function can return one of the following return codes:

```
DXFILE_OK
DXFILEERR_BADALLOC
DXFILEERR_BADFILEFLOATSIZE
DXFILEERR_BADFILETYPE
```

```

DXFILEERR_BADFILEVERSION
DXFILEERR_BADRESOURCE
DXFILEERR_BADVALUE
DXFILEERR_FILENOTFOUND
DXFILEERR_RESOURCENOTFOUND
DXFILEERR_URLNOTFOUND

```

The following code shows how to use the `CreateEnumObject` function to load an X file from a file on disk.

```

LPD3DXFILEENUMOBJECT EnumObject = NULL;
File->CreateEnumObject("MyXFile.X", DXFILELOAD_FROMFILE, &EnumObject);

```

Enumerating Top Objects

Once an enumeration object has been created you can then cycle through X file data, object by object, and pick out what you need. The enumeration object can be thought of as a linear list of all top-level objects in the X file. By top level, I mean all objects that may have children and do not have parents. Essentially, a single object in the file is represented by an `ID3DXFileData` object, and the enumeration object provides you with a list of these, one by one. To enumerate through all top-level objects you must first obtain the number of top-level objects, which can be retrieved by calling the **GetChildren** method of `ID3DXFileEnumObject`. Once you have retrieved the number of objects, you then call the **GetChild** method of `ID3DXFileEnumObject`. This function requires one parameter: the index of the child to retrieve. The syntax and parameters for both `GetChildren` and `GetChild` appear below, and then some code shows you how to enumerate through all top-level objects.

```

HRESULT GetChildren
(
    SIZE_T *puiChildren
);

```

SIZE_T *puiChildren

Address to receive the number of top-level objects.

HRESULT GetChild

```
(
    SIZE_T id,
    ID3DXFileData **ppObj
);
```

SIZE_T id

Index of the object to retrieve.

ID3DXFileData **ppObj

Address to where the data object is returned.

Below is some code that uses the enumerator to cycle through every top-level object in an X file.

```
SIZE_T Size = 0;

Enum->GetChildren(&Size);

for(SIZE_T Counter = 0; Counter < Size; Counter++)
{
    LPD3DXFILEDATA DataObject = NULL;
    Enum->GetChild(Counter, &DataObject);
    //Do stuff here
    DataObject->Release();
}

Enum->Release();
```

Enumerating Child Objects

You'll remember that objects in an X file can have child objects. Until now you've only seen how to enumerate top-level objects using the X file enumerator. Now we will look at enumerating child objects. Doing this is quite simple; in fact, it's exactly the same as

using the enumerator object to enumerate top-level objects. You just call the `GetChildren` and `GetChild` methods of the `ID3DXFileData` object. Remember, `ID3DXFileData` represents any single data object in an X file. This means you'll need to call `GetChildren` on each `ID3DXFileData` object to check whether it has children.

```
SIZE_T Size = 0;

Data->GetChildren(&Size);

for(SIZE_T Counter = 0; Counter < Size; Counter++)
{
    LPD3DXFILEDATA DataObject = NULL;
    Data->GetChild(Counter, &DataObject);
    //Do stuff here
    DataObject->Release();
}

Data->Release();
```

Processing Child Objects

As you cycle through child objects, you'll notice they can be one of two types: instanced or referenced. In other words, the object can be either a normal data object or it can be a *pointer* to another object elsewhere in the file. To check whether the object is an actual object or just a reference, you can call the **IsReference** method of `ID3DXFileData`.

```
bool IsReference = Data->IsReference();
```

Enumeration Overview

This section shows how the enumeration process can be divided into two small functions. These two functions will allow you to enumerate through all the objects in an X file. The next sections demonstrate how to actually read data from your objects.

```

VOID OpenXFile(CString FileName)
{
    LPD3DXFILE File = NULL;

    if(FAILED(D3DXFileCreate(&File)))
        return;

    File->RegisterTemplates((LPVOID)D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES);

    LPD3DXFILEENUMOBJECT EnumObject = NULL;

    If(FAILED(File->CreateEnumObject(FileName, DXFILELOAD_FROMFILE,
        &EnumObject)))
        return;

    SIZE_T Size = 0;

    EnumObject ->GetChildren(&Size);

    for(SIZE_T Counter = 0; Counter < Size; Counter++)
    {
        LPD3DXFILEDATA DataObject = NULL;
        EnumObject ->GetChild(Counter, &DataObject);
        //Do stuff here
        DataObject->Release();
    }

    EnumObject->Release();
    File->Release();
}

VOID ProcessChildObjects(LPD3DXFILEDATA Data)
{
    if(!Data)
        return;

    SIZE_T Size = 0;

    Data ->GetChildren(&Size);

```

```

for(SIZE_T Counter = 0; Counter < Size; Counter++)
{
    LPD3DXFILEDATA DataObject = NULL;
    Data ->GetChild(Counter, &DataObject);
    //Do stuff here
    DataObject->Release();
}
}

```

Getting Object Data

As you enumerate through objects in an X file you'll want to read data from them. Not surprisingly, you use the `ID3DXFileData` interface to do this. The following sections explain how to extract various pieces of information.

Object Names

The object name is the name you gave to an object in the X file; for example, `GAME01` or `MYOBJECT`. An object name is optional, so some objects may have names and others may not. To get the name of an `ID3DXFileData` object, you call its **GetName** method.

```

HRESULT GetName
(
    LPSTR szName,
    SIZE_T *puiSize
);

```

LPSTR szName

Pointer to an appropriately sized string to receive the object name. This can be `NULL` if you just want to retrieve the size of the string. This is useful to allow you to dynamically size your string accordingly.

SIZE_T *puiSize

Address of a `SIZE_T` structure, which is the size of the string in bytes.

Sample code:

```
SIZE_T Size = 0;
Data->GetName(NULL, & Size);
char* String = new char[Size];
Data->GetName(String, & Size);
```

Object Types

You'll also want to know whether an object is of a specific type. For example, you may have an X file filled with objects of different types — medi-kits, level objects, etc. So, it'll be useful to take an object and query it to see whether it's a medi-kit or whatever. To check an object type, you call the **GetType** method of `ID3DXFileData`.

```
HRESULT GetType
(
    const GUID *pType
);
```

```
const GUID *pType
```

Address to where the object's GUID is returned.

This function returns a GUID (globally unique identifier) that corresponds to the template declaration in the X file. This allows you to determine the data type of the object. Before we can check to see whether an object is of a specific type, we'll need to find a way to hold our object's GUIDs in code. This is so we'll be able to compare our returned GUID with something.

Let's take our sample `MY_GAME_INFO` template, whose GUID was generated by `guidgen.exe` as shown earlier in the chapter.

```
template MY_GAME_INFO{
    <CE1D5996-CA1D-4832-A6D9-C923EC39FC0F>
    STRING GameName;
    STRING GameMaker;
    DWORD Version;
}
```

When we copied this over from guidgen.exe, the text looked something like this:

```
// {CE1D5996-CA1D-4832-A6D9-C923EC39FC0F}
DEFINE_GUID(<<name>>,
0xce1d5996, 0xca1d, 0x4832, 0xa6, 0xd9, 0xc9, 0x23, 0xec, 0x39, 0xfc, 0xf);
```

To represent this GUID in our code we can copy some of this text and paste it to the top of our source file. We'll also need to fill in the object name where it says <<name>>. Like this:

```
DEFINE_GUID(GUID_MYGAMEINFO, 0xce1d5996, 0xca1d, 0x4832, 0xa6, 0xd9,
0xc9, 0x23, 0xec, 0x39, 0xfc, 0xf);
```

***NOTE.** DEFINE_GUID is macro that allows us to define GUIDs.

So now we can check whether an object is of type MYGAMEINFO, like this:

```
GUID ObjectGUID;

Data->GetType(&ObjectGUID);

If(ObjectGUID == GUID_MYGAMEINFO)
{
    //This object is of type MY_GAME_INFO
}
```

The standard GUIDs for normal mesh objects appear in the following list.

TID_D3DRMAnimation	TID_D3DRMCoords2d
TID_D3DRMAnimationKey	TID_D3DRMExternalVisual
TID_D3DRMAnimationOptions	TID_D3DRMFloatKeys
TID_D3DRMAnimationSet	TID_D3DRMFrame
TID_D3DRMAppData	TID_D3DRMFramePosition
TID_D3DRMBoolean	TID_D3DRMFrameRotation
TID_D3DRMBoolean2d	TID_D3DRMFrameTransformMatrix
TID_D3DRMCamera	TID_D3DRMFrameVelocity
TID_D3DRMColorRGB	TID_D3DRMGuid
TID_D3DRMColorRGBA	TID_D3DRMIndexedColor

TID_D3DRMInfo	TID_D3DRMMaterialWrap
TID_D3DRMInlineData	TID_D3DRMMatrix4x4
TID_D3DRMLight	TID_D3DRMMesh
TID_D3DRMLightAttenuation	TID_D3DRMMeshFace
TID_D3DRMLightPenumbra	TID_D3DRMMeshFaceWraps
TID_D3DRMLightRange	TID_D3DRMMeshMaterialList
TID_D3DRMLightUmbra	TID_D3DRMMeshNormals
TID_D3DRMMaterial	TID_D3DRMMeshTextureCoords
TID_D3DRMMaterialAmbient- Color	TID_D3DRMMeshVertexColors
TID_D3DRMMaterialArray	TID_D3DRMProgressiveMesh
TID_D3DRMMaterialDiffuse- Color	TID_D3DRMPropertyBag
TID_D3DRMMaterialEmissive- Color	TID_D3DRMRightHanded
TID_D3DRMMaterialPower	TID_D3DRMStringProperty
TID_D3DRMMaterialSpecular- Color	TID_D3DRMTextureFilename
	TID_D3DRMTextureReference
	TID_D3DRMTimedFloatKeys
	TID_D3DRMUrl
	TID_D3DRMVector

Object Data

Other than object type and name, the most important thing you're going to want to know is what data an object actually contains. This is really simple to do. To get the object data you must call the **Lock** method of `ID3DXFileData`. This locks the object's data buffer and returns a pointer to the data. Once you've finished accessing the data you must call the matching `Unlock` method. The syntax and parameters for `Lock` appear below, and then some sample code demonstrates how to access an object's data.

```
HRESULT Lock
(
    SIZE_T *pSize,
    const VOID **ppData
);
```

SIZE_T *pSize

Address to receive the size of the locked data buffer.

const VOID **ppData

Address to receive a pointer to the data.

Below is some code to get an object's data. Once the Lock function has succeeded it'll give you a buffer where all your data is stored. To access this data you can manually increment through the bytes and read in the data according to the order of your template, or you can define a structure that matches your template and typecast the buffer. The code below shows how to do this using the sample MY_GAME_INFO template.

```
//Structure for MY_GAME_INFO matches the X file template
struct MY_GAME_INFO{
    char* GameName;
    char* GameMaker;
    DWORD Version;
}

SIZE_T Size = 0;
MY_GAME_INFO* MyData = NULL;
Object->Lock((const void*)&MyData, &Size);
//Access data here
Object->Unlock();
```

Saving Data to X Files — Save Object

The last thing to cover in this chapter is how to save information to an X file. This would be a useful way to store your saved games. The first step is to create an ID3DXFile object, as shown earlier. Once completed, you must create an ID3DXFileSaveObject interface, which will be used to add data to your file. To do this, you call the **CreateSaveObject** method of ID3DXFile. This function is simple to use; you just pass it a filename to where your data should

be saved, the format of the file (binary or text), and an address to where the ID3DXFileSaveObject interface is to be returned.

```
HRESULT CreateSaveObject(
    LPCVOID pData,
    D3DXF_FILESAVEOPTIONS flags,
    D3DXF_FILEFORMAT dwFileFormat,
    ID3DXFileSaveObject **ppSaveObj
);
```

LPCVOID *pData*

Name of the file to create to which data is to be saved.

D3DXF_FILESAVEOPTIONS *flags*

Specifies the string format of the filename, which can be one of the following parameters:

```
D3DXF_FILESAVE_TOFILE
D3DXF_FILESAVE_TOWFILE
```

D3DXF_FILEFORMAT *dwFileFormat*

Format of the X file, which can be any of the following parameters:

```
D3DXF_FILEFORMAT_BINARY
D3DXF_FILEFORMAT_COMPRESSED
D3DXF_FILEFORMAT_TEXT
```

ID3DXFileSaveObject *ppSaveObj***

Address to where a file save object is returned.

```
LPD3DXFILESAVEOBJECT Save = NULL;
pDXFile->CreateSaveObject("test.x", DXFILEFORMAT_TEXT, &Save);
```

Preparing

Before data can be saved, you must register any templates your file will use. To register templates, you call the RegisterTemplates function, which was shown earlier.

Saving Data

Once you've registered your templates to the X file, you can start putting in your data. You do this by creating `ID3DXFileSaveData` objects. You fill them with your data and then link them together to make a tree. You then save all the top-level objects. Don't worry; DirectX will cycle through every object, their children, and their children, etc., and it'll save those for you.

To create an `ID3DXFileSaveData` object you call the **CreateDataObject** method of `ID3DXFileSaveObject`. This function allows you to give the object a name, tell it which template to use, and assign it some data. On completion it returns a valid `ID3DXFileSaveData` interface.

```
HRESULT CreateDataObject
(
    REFGUID rguidTemplate,
    LPCSTR szName,
    const GUID *pguid,
    DWORD cbSize,
    LPVOID pvData,
    ID3DXFileSaveData **ppObj
);
```

REFGUID rguidTemplate

GUID representing the ID of the template to use.

LPCSTR szName

Name of the object to create, or NULL if the object doesn't have a name.

const GUID *pguid

Most of the time you can pass NULL for this value.

DWORD cbSize

Size of the data object in bytes.

LPVOID pvData

Pointer to the buffer that actually contains the data.

ID3DXFileSaveData **ppObj

Address to receive a valid pointer to a save data object.

Sample code:

```
MYGAMEINFO Info;
ZeroMemory(&Info, sizeof(MYGAMEINFO));
Info.Name = "Test";

Save->CreateDataObject(GUID_MYGAMEINFO, "Test", NULL, sizeof(Test),
                       &T, &Data);
```

Building the Tree

As you already know, objects in an X file are stored in a structured hierarchy. That is, objects can have children, and they in turn may contain other children. The way you build this hierarchy is remarkably simple. You just call the `AddDataObject` method of `ID3DXFileSaveData`. You call this on the parent object and you pass, as an argument, a pointer to the object that is to become one of its children. Its syntax and some sample code are shown below. The parameters are the same as those for the `AddDataObject` function seen in the previous section.

```
HRESULT AddDataObject(
    REFGUID rguidTemplate,
    LPCSTR szName,
    const GUID *pId,
    SIZE_T cbSize,
    LPCVOID pvData,
    ID3DXFileSaveData **ppObj
);
```

Sample code:

```
//Adds a new child object to a data object
Data-> AddDataObject(ChildData);
```

Committing the Data

The final stage in saving X files is to actually flush the data to a file. This sends your data to the actual X file on disk. To do this you call the Save method of ID3DXFileSaveObject. You don't need to save all objects manually. This function will automatically cycle through child objects and save them to the file for you. It requires no arguments. You can save an object and all its children to file like this:

```
pSave->Save();
```

Conclusion

Hopefully this chapter has provided some excellent insight into how X files work and will serve as an excellent model for storing your own data as well as meshes. Before reading further in this book I recommend taking some time to play around and explore X files generally. A good way to practice would be to take an existing X file, like Tiny.x provided with the Microsoft DirectX SDK, and cycle through the objects.



Chapter 7

Meshes

You may have guessed by now that once you move beyond rendering polygons, cubes, and other basic primitives, it becomes considerably harder to represent complex shapes. For example, creating a 3D world with a house, some trees, and perhaps some hills requires a lot of polygons, and I mean a *lot*. Creating these sorts of shapes in code would just be too much hard work. The solution is to use a 3D package to visually design the models and let it do the hard work for you. Once the model is created, you can then export each and every polygon, including its position and orientation, into an X file. DirectX can then scan this file and load the model for you. Another name for a 3D model is a mesh, and meshes form the subject matter for this chapter. This chapter explains how to:

- Create meshes using 3ds max
- Export meshes to an X file
- Load meshes into Direct3D
- Render meshes
- Scan through a mesh's vertices
- Set a mesh's FVF
- Interpolate between two keyframed meshes

What Are Meshes?

A *mesh* is a 3D model. So, if you wanted to have a dragon in your game, or a spaceship, or even a car, each would be a separate mesh. Nowadays, meshes are made from a complex arrangement of polygons, which are faces defined by three vertices. A mesh is full of these very small polygons juxtaposed together, this way and that, to make shapes and models. Meshes will also have textures to make them look more real, like green, slimy skin for a dragon or sexy chrome bodywork for a futuristic car.

You can make a mesh in many ways. You could, if you really, really wanted, manually type the coordinates for each and every vertex in your mesh. Or, to make life simpler, you can use a 3D package to design a mesh in a fraction of the time, using just your mouse and some artistic ability. If you chose the former, I wish you the very best of luck. If you chose the latter, however, the next section explains some of the packages available to make these 3D meshes.

How to Make Meshes

There are an enormous number of 3D packages out there to make meshes for your games. Some of them are free, some of them are cheap, and some are very expensive. Furthermore, some are very simple to use while others are extremely complex. It is beyond the scope of this book to teach you how to use these programs, but I will point you in the right direction. This chapter briefly covers only one of these 3D package's methods for exporting your model to an X file. Of the many programs you can find out there, I shall mention a few (in no particular order):

- MilkShape 3D
- LightWave 3D
- 3D Studio MAX
- Maya

- Softimage
- Cinema 4D

***NOTE.** To get more information about the 3D packages listed here, see the recommended reading section at the back of this book.

This chapter looks more closely at 3ds max and exporting meshes to the X file format using a freely available third-party plug-in called Panda Exporter.

How to Export Meshes

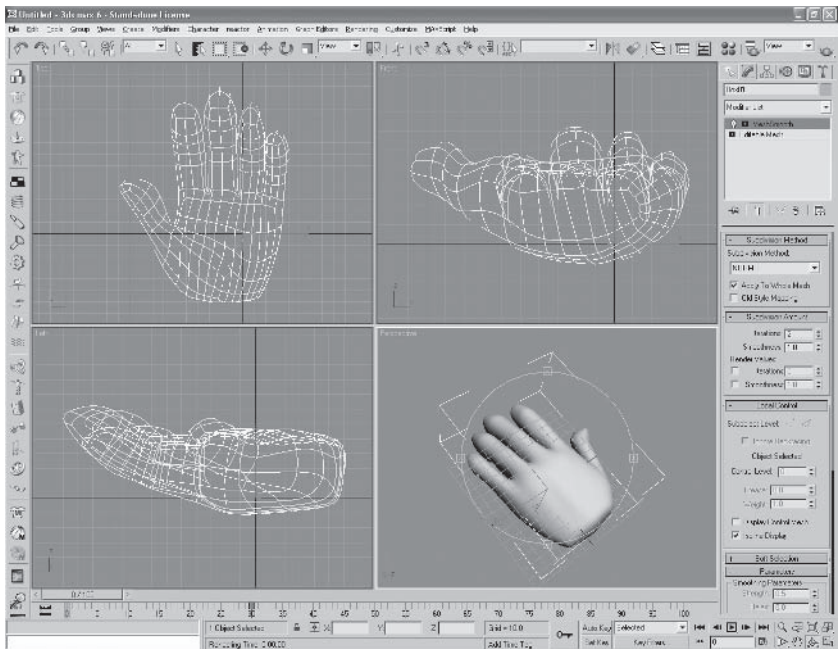


Figure 7.1

I used 3ds max to produce the sample mesh in Figure 7.1. Using MAX you can make the mesh and save it for use at a later time. However, 3ds max 6 — the latest version at the time of publication — does not come with any support to export meshes to the DirectX X file format. To do this, you must either develop your own

plug-in (yuck!) or download one somebody else has already made. In this chapter I use the Panda DirectX X file exporter from Pandasoft. This plug-in can be downloaded from <http://www.andy-tather.co.uk/Panda/directxmax.htm>. It is really simple to use and provides a lot of flexibility over how you can export your meshes. Once this file has been downloaded you simply copy it to the 3ds max plug-in directory and then load up MAX. To use this plug-in, you select a mesh to export in your scene and click the File | Export option. Select .X as the format to save and you'll be presented with a screen very much like Figure 7.2.

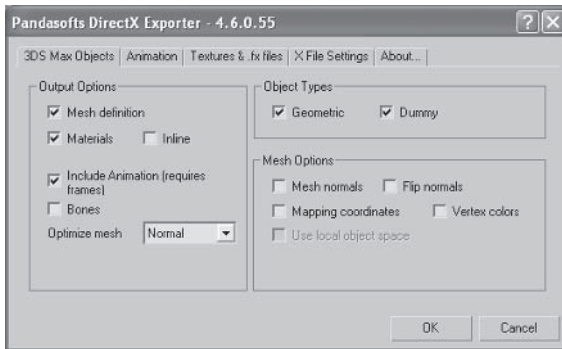


Figure 7.2

*** NOTE.** Don't worry about bones, frames, etc., at this point; this chapter only deals with static meshes, that is, meshes that are not animated. Animation is covered later in this book.

Testing Your Mesh

Before loading an exported mesh into your application using Direct3D you can use the DirectX Mesh Viewer application to view the mesh. This program allows you to see your mesh and rotate a camera around to see it from any angle. This is useful for checking whether the mesh exported properly from your 3D package.

*** NOTE.** Sometimes Mesh Viewer does not work properly when viewing animated meshes.

Meshes in Direct3D

Direct3D represents a single, static mesh using the `ID3DXMesh` interface. This interface keeps track of all the mesh's vertices, faces, and adjacency data. There are two common ways you can load a mesh into Direct3D. The next two sections explain each method.

***NOTE.** To see more on loading meshes, take a look at the SDK examples (Tutorial 6).

Loading Meshes from X Files

The simplest way to load a mesh is straight from an X file. This method works fine if there's only one mesh in your X file or you want DirectX to load in all the mesh data from the file and treat it as one single mesh. The DirectX SDK comes with some good examples of single meshes in files, such as the `tiger.x` file. To load a mesh from a file, you call the **`D3DXLoadMeshFromX`** function. This function accepts a filename for the X file and returns a valid `ID3DXMesh` interface representing the loaded data. It also returns a number of other buffers containing material and texture information. Don't worry about these for now; they will be processed later. After the mesh has been loaded, you'll get an `ID3DXMesh` interface. The syntax and parameters for `D3DXLoadMeshFromX` follow, and then I'll show you some sample code to load a mesh.

```
HRESULT WINAPI D3DXLoadMeshFromX
(
    LPCTSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE9 pD3DDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER *ppEffectInstances,
    DWORD *pNumMaterials,
```



```
LPD3DMESH *ppMesh
);
```

LPCTSTR *pFilename*

Filename of the X file to open.

DWORD *Options*

This can be one or more of the following options:

```
D3DMESH_32BIT
D3DMESH_DONOTCLIP
D3DMESH_DYNAMIC
D3DMESH_IB_DYNAMIC
D3DMESH_IB_MANAGED
D3DMESH_IB_SOFTWAREPROCESSING
D3DMESH_IB_SYSTEMMEM
D3DMESH_IB_WRITEONLY
D3DMESH_MANAGED
D3DMESH_NPATCHES
D3DMESH_POINTS
D3DMESH_RTPATCHES
D3DMESH_SOFTWAREPROCESSING
D3DMESH_SYSTEMMEM
D3DMESH_USEHWONLY
D3DMESH_VB_DYNAMIC
D3DMESH_VB_MANAGED
D3DMESH_VB_SHARE
D3DMESH_VB_SOFTWAREPROCESSING
D3DMESH_VB_SYSTEMMEM
D3DMESH_VB_WRITEONLY
D3DMESH_WRITEONLY
```

This parameter is used to specify various creation options for the mesh. Typically you will pass `D3DMESH_SYSTEMMEM`.

LPDIRECT3DDEVICE9 *pD3DDevice*

Pointer to the Direct3D device used to create the mesh.

LPD3DXBUFFER **ppAdjacency*

Address to receive a buffer containing polygonal adjacency information.

LPD3DXBUFFER *ppMaterials

Address to receive material information.

LPD3DXBUFFER *ppEffectInstances

Address to receive a buffer of effect instances. This subject is not covered in this book. Just pass NULL.

DWORD *pNumMaterials

Address to receive the number of materials in the buffer ppMaterials.

LPD3DXMESH *ppMesh

Address to receive a valid ID3DXMesh interface.

Sample code:

```
LPD3DXBUFFER pD3DXMtrlBuffer = NULL;    //Array of materials
DWORD dwNumMaterials = 0;                //Number of materials
LPD3DXMESH pMesh = NULL;                 //Mesh

D3DXLoadMeshFromX("Mesh.x", D3DXMESH_SYSTEMMEM,
    g_pd3dDevice, NULL,
    &pD3DXMtrlBuffer, NULL, &dwNumMaterials,
    &pMesh);
```

Loading Meshes from X File Data Objects

The second method of loading a mesh is preferred if you may have more than one mesh in a single X file or your file contains more than just mesh data. For example, you could specify that your game files store data about the levels and position of medi-kits, and also contain each mesh that'll appear in every level. If so, it's likely you'll want to:

- Cycle through the objects in your X file (as shown in the previous chapter)
- Check whether the object is a mesh object
- For every mesh object call D3DXLoadMeshFromXof

D3DXLoadMeshFromXof takes an IDirectXFileData object and returns a valid mesh, ID3DXMesh. Remember, an IDirectXFileData

is a valid data object contained in an X file. X files can have none, one, or more data objects, and each object may have none, one, or more child objects. `D3DXLoadMeshFromXof` is almost identical to `D3DXLoadMeshFromX`, except it doesn't require a filename; instead, it requires an `IDirectXFileData` pointer representing a valid data object and which is also a mesh. The syntax and parameters for `D3DXLoadMeshFromXof` are listed below.

```
HRESULT WINAPI D3DXLoadMeshFromXof(
    LPD3DXFILEDATA pxofMesh,
    DWORD Options,
    LPDIRECT3DDEVICE9 pD3DDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER *ppEffectInstances,
    DWORD *pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

LPD3DXFILEDATA *pxofMesh*

Pointer to an X file data object.

DWORD *Options*

This can be one or more of the following options:

```
D3DXMESH_32BIT
D3DXMESH_DONOTCLIP
D3DXMESH_DYNAMIC
D3DXMESH_IB_DYNAMIC
D3DXMESH_IB_MANAGED
D3DXMESH_IB_SOFTWAREPROCESSING
D3DXMESH_IB_SYSTEMMEM
D3DXMESH_IB_WRITEONLY
D3DXMESH_MANAGED
D3DXMESH_NPATCHES
D3DXMESH_POINTS
D3DXMESH_RTPATCHES
D3DXMESH_SOFTWAREPROCESSING
D3DXMESH_SYSTEMMEM
```

D3DXMESH_USEHWONLY
D3DXMESH_VB_DYNAMIC
D3DXMESH_VB_MANAGED
D3DXMESH_VB_SHARE
D3DXMESH_VB_SOFTWAREPROCESSING
D3DXMESH_VB_SYSTEMMEM
D3DXMESH_VB_WRITEONLY
D3DXMESH_WRITEONLY

This parameter is used to specify various creation options for the mesh. Typically you will pass **D3DXMESH_SYSTEMMEM**.

LPDIRECT3DDEVICE9 *pD3DDevice*

Pointer to the Direct3D device used to create the mesh.

LPD3DXBUFFER **ppAdjacency*

Address to receive a buffer containing polygonal adjacency information.

LPD3DXBUFFER **ppMaterials*

Address to receive material information.

LPD3DXBUFFER **ppEffectInstances*

Address to receive a buffer of effect instances. This subject is not covered in this book. Just pass **NULL**.

DWORD **pNumMaterials*

Address to receive the number of materials in the buffer *ppMaterials*.

LPD3DXMESH **ppMesh*

Address to receive a valid **ID3DXMesh** interface.

The code below shows how, on every iteration of an X file data object, you can check its GUID to determine whether it's a mesh object. If so, you pass it to **D3DXLoadMeshFromXof** and it returns a valid mesh.

```
const GUID *pType = NULL;

if(FAILED(pDXData->GetType(&pType)))
    return false;
```

```

if(*pType == TID_D3DRMMesh)
{
    MessageBox(NULL, "", "Mesh was found", MB_OK);
}

```

Mesh Materials and Textures

Loading your mesh, whether from a file with `D3DXLoadMeshFromX` or from a data object with `D3DXLoadMeshFromXof`, is just the beginning. At this stage you will have a bunch of buffers and an `ID3DXMesh` interface. `ID3DXMesh` represents the mesh's raw data, such as the XYZ position of vertices, transparency information, and other structural data. The other buffers represent materials and textures, and we'll need to process this information if we expect the mesh to render properly. If you need a recap on materials and textures, take a look at Chapter 5. In short, materials tell Direct3D how the mesh appears when lit, and textures are actual images that cover the model to make it look more realistic, like it's made from real-life objects.

Once the mesh has been loaded, the first thing you'll need to do is create an array of material and texture objects large enough to hold the materials and textures of the mesh. You'll notice from the syntax and parameters in previous sections that each of the mesh loading functions, `D3DXLoadMeshFromX` and `D3DXLoadMeshFromXof`, return the number of textures and materials contained in the mesh. This was returned in `pNumMaterials`. So allocating memory for them is simple. The code to do this follows (where `g_dwNumMaterials` represents the returned number of materials).

```

g_pMeshMaterials = new D3DMATERIAL9[g_dwNumMaterials];
g_pMeshTextures = new LPDIRECT3DTEXTURE9[g_dwNumMaterials];

```

Next, we'll need to grab all the material and texture information about the mesh so we can populate our material and texture arrays with useful information. You'll notice each mesh loading function returns all its material and texture data in a buffer, `ppMaterials`. This buffer is represented by the `ID3DXBuffer` interface.

Essentially, this specific buffer is an array of D3DXMATERIAL structures representing the mesh's material data. To get a pointer to the beginning of this array, we call the **GetBufferPointer** method of ID3DXBuffer. It requires no arguments and can be called like this:

```
D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->
    GetBufferPointer();
```

The D3DXMATERIAL structure contains two members: a D3DMATERIAL9 member (as explained in Chapter 5), which is the actual material; and a string, which is the filename of a texture to load from disk. The structure looks like this:

```
typedef struct D3DXMATERIAL {
    D3DMATERIAL9 MatD3D;
    LPSTR pTextureFilename;
} D3DXMATERIAL;
```

Now, let's actually fill your arrays with something useful. The objective here is simple: You just loop through each material and texture in your arrays, which you created earlier. (Remember, both arrays are the same size.) Then, on every iteration, you copy the corresponding material from the buffer to the material in your array, and you also create a texture from the file specified by pTextureFilename and assign it to the texture in your array. It's that simple. Once the loop has completed, you'll have an array of materials and textures suitable for your loaded mesh. The following code shows this process in action.

```
//Loop for all materials and textures
for(DWORD i=0; i<g_dwNumMaterials; i++)
{
    // Copy the material from the buffer to your array
    g_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;

    // Set the ambient color for the material
    // Don't worry about the technical reasons for this
    // Many 3D modeling apps export the ambient color in the diffuse
    g_pMeshMaterials[i].Ambient = g_pMeshMaterials[i].Diffuse;
```

```

//Set the texture to NULL
g_pMeshTextures[i] = NULL;

if(d3dxMaterials[i].pTextureFilename != NULL &&
    strlen(d3dxMaterials[i].pTextureFilename) > 0)
{
    D3DXCreateTextureFromFile(g_pd3dDevice,
                             d3dxMaterials[i].pTextureFilename,
                             &g_pMeshTextures[i]);
}
}

```

That's it. That's all there is to loading a mesh. You first load the mesh from a file or data object, which gives you an ID3DXMesh interface. Then you create an array for the mesh's materials and textures. Now you're actually ready to render your mesh. Before moving on, then, the following code shows the complete mesh loading routine for loading a single mesh from an X file.

```

LPD3DXBUFFER pD3DXMtrlBuffer;

// Load the mesh from the specified file
D3DXLoadMeshFromX("Tiger.x", D3DXMESH_SYSTEMMEM,
                  g_pd3dDevice, NULL, &pD3DXMtrlBuffer, NULL,
                  &g_dwNumMaterials, &g_pMesh);

D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->
    GetBufferPointer();

g_pMeshMaterials = new D3DMATERIAL9[g_dwNumMaterials];
g_pMeshTextures = new LPDIRECT3DTEXTURE9[g_dwNumMaterials];

for(DWORD i=0; i<g_dwNumMaterials; i++)
{
    // Copy the material
    g_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;

    // Set the ambient color for the material (D3DX does not do this)
    g_pMeshMaterials[i].Ambient = g_pMeshMaterials[i].Diffuse;
}

```

```

g_pMeshTextures[i] = NULL;
if(d3dxMaterials[i].pTextureFilename != NULL &&
    strlen(d3dxMaterials[i].pTextureFilename) > 0)
{
    // Create the texture
    if(FAILED(D3DXCreateTextureFromFile(g_pd3dDevice,
        d3dxMaterials[i].pTextureFilename,
        &g_pMeshTextures[i])))
    }
}

// Done with the material buffer
pD3DXMtrlBuffer->Release();

```

Rendering Meshes

Now it's time to bring your mesh to life and show it on screen. Rendering a mesh is a simple process. You just need to be aware that a mesh is made up from subsets. A *subset* is a collection of polygons, and each subset in the mesh is grouped by texture. So there are as many subsets in the mesh as there are textures — one subset per texture. In other words, all the polygons in subset 1 will share the same texture, and those in subset 2 will share the same texture, and so on.

When you render a mesh you render it subset by subset, every frame. Effectively, all you need to do is loop through each subset in the mesh, set the Direct3D device's texture and material to the corresponding array element in your material and texture array, and then finally render the subset. If that sounds complicated, don't worry; it's not. The following code demonstrates just how easy it really is.

```

for(DWORD i=0; i<g_dwNumMaterials; i++)
{
    // Set the material and texture for this subset
    g_pd3dDevice->SetMaterial(&g_pMeshMaterials[i]);
    g_pd3dDevice->SetTexture(0, g_pMeshTextures[i]);
}

```



```
// Draw the mesh subset
g_pMesh->DrawSubset(i);
}
```

The most noticeable line in the above code fragment contains a call to the DrawSubset method of ID3DXMesh. This renders a specific mesh's subset. It requires only one parameter: the number of the subset to render.

*** NOTE.** Remember, meshes are affected by world transformations like any other geometry. So you can use matrices to scale, move, and rotate your meshes.

Cleaning Up Meshes

You're probably seeing by now that meshes are simple to use. Cleaning up meshes is no more difficult either. It's important, however, to know that you're not just freeing the ID3DXMesh interface, you're also freeing the textures and materials. The code to free a mesh and associated textures and materials looks like this:

```
if(g_pMeshMaterials != NULL)
    delete[] g_pMeshMaterials;

if(g_pMeshTextures)
{
    for(DWORD i = 0; i < g_dwNumMaterials; i++)
    {
        if(g_pMeshTextures[i])
            g_pMeshTextures[i]->Release();
    }
    delete[] g_pMeshTextures;
}

if(g_pMesh != NULL)
    g_pMesh->Release();
```

More on Meshes

The previous sections have demonstrated how to load a mesh, load and store its materials and textures, and then render it to the screen. This means you can go ahead and model your goblins, robots, and whatever other creatures will appear in your game. You now have sufficient knowledge to load them into your DirectX projects and position them in a 3D world.

The remaining sections of this chapter will refine your knowledge of meshes and explain how to perform more complicated tasks with them. Before proceeding, however, it's a good idea to ensure you're familiar with the mesh concepts already presented.

Meshes and Vertex Buffers

It's already been explained that a *mesh* is an arrangement of polygons, and a *polygon* is represented by three vertices. In essence, therefore, a mesh, at its most basic level, is a collection of vertices. Like every other polygon and geometric object in DirectX, meshes store their vertices in a vertex buffer (explained in Chapter 4). Simply put, a *vertex buffer* is an array of vertices.

On occasion, especially if you want to dynamically adjust your meshes at run time, in code, you might want to access the vertices in this buffer and change them. For example, if you wanted to make a goblin's nose gradually longer over time, you could cycle through the mesh's vertex buffer on every frame and increment the nose vertices' XYZ position. This would give the impression of the nose changing size over time.

To access a mesh's vertex buffer you call the `GetVertexBuffer` method of `ID3DXMesh`. It requires only one parameter: an address to receive an `IDirect3DVertexBuffer9` pointer representing the mesh's vertex buffer. Once you have this, you can access the mesh's vertices like you would any normal vertex buffer. This process of locking and unlocking vertex buffers was explained in Chapter 4. The following code obtains a mesh's vertex buffer.

```
LPDIRECT3DVERTEXBUFFER9 VertexBuffer = NULL;
g_pMesh->GetVertexBuffer(&VertexBuffer);
```

*** NOTE.** You can find out how many vertices are in a mesh's vertex buffer by calling the `GetNumVertices` method of `ID3DXMesh`.

Meshes and FVs

Fine, so you know how to access a mesh's vertex buffer; that's good. However, there's a problem. You'll remember from Chapter 4 that vertices are stored using a flexible vertex format (FVF). This means vertices can contain all kinds of data — position data, color data, texture data, and the list goes on. Some vertices may contain only the bare minimum, while other vertices may be more complex. Furthermore, the programs to create meshes — like 3ds max and LightWave — vary greatly and, depending on the mesh itself, the vertex format is likely to differ from one mesh to another. Simply put, this means that after loading in your mesh, you can't immediately tell which format its vertices are in. So, as you cycle through your vertex buffer to read in each vertex, you won't know the size of each one (the stride), meaning you won't know where one vertex ends and the next begins.

There are several ways to handle this problem. First, you can manually work out the FVF of any mesh and thus determine what data its vertices contain. To do this, you call the `GetFVF` method of `ID3DXMesh`. This function requires no parameters and returns a `DWORD`, which is the FVF descriptor. For more information on FVFs, please refer to Chapter 4. The code below retrieves a mesh's FVF.

```
DWORD FVF = g_Mesh->GetFVF();
```

*** NOTE.** To get the actual size in bytes of the data structure used to hold information matching this FVF vertex format, you can call `D3DXGetFVFVertexSize`.

The second way to handle this problem is to programmatically create a cloned mesh (a duplicate mesh) and set its FVF to an FVF you specify. That's right; you can say, "Hey Direct3D, I have my own

FVF and I want to create a duplicate mesh in memory that uses my FVF for all its vertices.” This way you can know the exact format of a mesh’s vertices, and iterating through them in the vertex buffer will be quite simple. This means that in future operations — such as iterating through the vertex buffer or animating — you’ll be dealing with the cloned mesh, not the original mesh. To create a cloned mesh that is set to an FVF of your choosing, you call the **CloneMeshFVF** method of **ID3DXMesh**. This function requires several parameters, including the FVF descriptor the closed mesh will use and an address to receive an **ID3DXMesh** pointer representing the cloned mesh. The syntax and parameters for **CloneMeshFVF** are listed below, and then some sample code demonstrates how this function can be used.

```
HRESULT CloneMeshFVF(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXMESH *ppCloneMesh
);
```

DWORD Options

Options for the cloned mesh. Typically you will use a value such as **D3DXMESH_SYSTEMMEM**. This value can be any of the following:

```
D3DXMESH_32BIT
D3DXMESH_DONOTCLIP
D3DXMESH_DYNAMIC
D3DXMESH_IB_DYNAMIC
D3DXMESH_IB_MANAGED
D3DXMESH_IB_SOFTWAREPROCESSING
D3DXMESH_IB_SYSTEMMEM
D3DXMESH_IB_WRITEONLY
D3DXMESH_MANAGED
D3DXMESH_NPATCHES
D3DXMESH_POINTS
D3DXMESH_RTPATCHES
D3DXMESH_SOFTWAREPROCESSING
D3DXMESH_SYSTEMMEM
```

```

D3DXMESH_USEHWONLY
D3DXMESH_VB_DYNAMIC
D3DXMESH_VB_MANAGED
D3DXMESH_VB_SHARE
D3DXMESH_VB_SOFTWAREPROCESSING
D3DXMESH_VB_SYSTEMMEM
D3DXMESH_VB_WRITEONLY
D3DXMESH_WRITEONLY

```

DWORD FVF

FVF descriptor the cloned mesh is to use.

LPDIRECT3DDEVICE9 pDevice

Pointer to a Direct3D device.

LPD3DXMESH *ppCloneMesh

Address to receive a cloned mesh.

Sample code:

```

LPD3DXMESH ClonedMesh = NULL;
g_Mesh->CloneMeshFVF(D3DXMESH_SYSTEMMEM, My_Custom_FVF, g_pD3DDevice,
                    &ClonedMesh);

```

Bounding Boxes and Spheres

When you produce your own games and begin to use meshes for your game characters you'll want to know how to perform collision detection. Wait a second. What's collision detection? Well, the actual process and calculations behind collision detection are beyond the scope of this book, but briefly, *collision detection* is being able to determine when your meshes and objects touch things. For example, if the player uses the arrow keys to move a character around a room he might move the character into objects like desks, walls, chairs, etc. Now, in most cases, it would look rather stupid if your character could walk straight through walls. Most of the time you'll want to block the character's path and prevent it from walking through solid items. Thus, you'll need to determine when the

character touches a wall or some other object, like another character. Once you determine when a collision has occurred, you can prevent any further movement closer to the object to make your game more realistic.

There are many ways collision detection can be performed; some are very fast and others very slow. For example, if you wanted to test whether a character mesh has collided with an enemy character you could do the following:

Cycle through every vertex in your character's vertex buffer and test to see whether any vertex falls between any other vertices in the enemy's vertex buffer. If it does, then a collision has occurred. If not, then both meshes are a sufficient distance from one another.

The biggest problem with this method concerns its speed. It's very, very slow, especially when you're performing this kind of calculation every frame. It simply isn't feasible to check every vertex in a scene against every other vertex just to see whether a collision has occurred. In most cases a collision won't have happened, so such processing would simply be a waste.

One good alternative, which gets around checking every vertex although it isn't quite as accurate, is to use a bounding box or bounding sphere. Both are ways to approximate the shape of your mesh. Consider Figures 7.3 and 7.4. A bounding box is an invisible box surrounding your mesh and is sized to the greatest extents of the mesh. A bounding sphere is an invisible sphere that surrounds your mesh. You only need to use one of these — either a bounding box or bounding sphere. Now, let's say we're using bounding spheres. If you generate a bounding sphere for each mesh in your level, you can determine whether a collision occurred by quickly comparing the spheres, rather than the meshes, to see whether they intersect. By comparing an approximated box or sphere, you can easily tell whether two meshes collide.

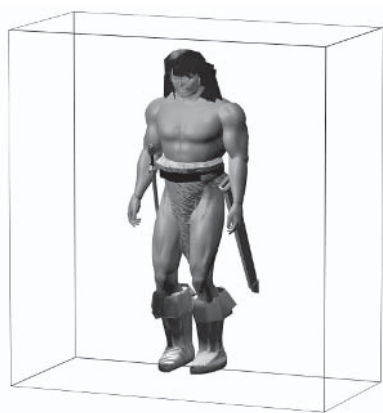


Figure 7.3:
Bounding box

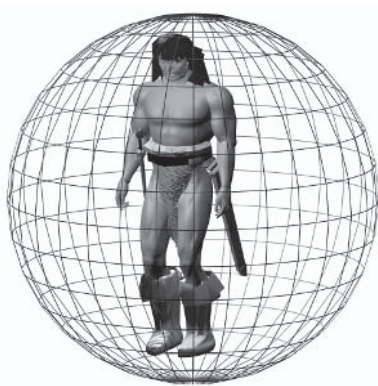


Figure 7.4:
Bounding sphere

To generate a bounding box for a mesh you call the **D3DXComputeBoundingBox** function. This function requires:

- A pointer to the first vertex in the mesh's vertex buffer
- The total number of the vertices in the mesh's vertex buffer
- The size in bytes of each vertex

This function returns two 3D vectors that describe the bounding box. One vector represents the bottom-left corner of the bounding box, and the other vector represents the top-right corner of the bounding box. The syntax and parameters for **D3DXComputeBoundingBox** follow, and then some code that demonstrates how to use this function.

```

HRESULT WINAPI D3DXComputeBoundingBox(
    const D3DXVECTOR3 *pFirstPosition,
    DWORD NumVertices,
    DWORD dwStride,
    D3DXVECTOR3 *pMin,
    D3DXVECTOR3 *pMax
);

```

const D3DXVECTOR3 *pFirstPosition

Pointer to the first vertex in the mesh's vertex buffer.

DWORD NumVertices

The number of vertices in the mesh's vertex buffer. You can find out how many vertices are in a mesh's vertex buffer by calling the `GetNumVertices` method of `ID3DXMesh`. It requires no parameters and returns the number of vertices in the mesh's vertex buffer.

DWORD dwStride

The size in bytes of a single vertex in the vertex buffer. You can find the size of any vertex from an FVF by calling the `D3DXGetFVFVertexSize` function. It requires one parameter: the FVF `DWORD` descriptor.

D3DXVECTOR3 *pMin

Address of a `D3DXVECTOR3` structure to receive the bottom-left corner of the computed bounding box.

D3DXVECTOR3 *pMax

Address of a `D3DXVECTOR3` structure to receive the top-right corner of the computed bounding box.

Sample code:

```

LPDIRECT3DVERTEXBUFFER9 VertexBuffer = NULL;
D3DXVECTOR3* Vertices = NULL;
D3DXVECTOR3 LowerLeftCorner;
D3DXVECTOR3 UpperRightCorner;
DWORD FVFVertexSize = D3DXGetFVFVertexSize(g_pMesh->GetFVF());
g_pMesh->GetVertexBuffer(&VertexBuffer);
VertexBuffer->Lock(0,0, (VOID**) &Vertices, D3DLOCK_DISCARD);

```



```

D3DXComputeBoundingBox(Vertices, g_pMesh->GetNumVertices(),
                        FVFVertexSize, &LowerLeftCorner,
                        &UpperRightCorner);
VertexBuffer->Unlock();
VertexBuffer->Release();

```

To generate a bounding sphere you call the **D3DXComputeBoundingSphere** function. This function requires exactly the same arguments as **D3DXComputeBoundingBox**, except it does not return two vectors. Instead, it returns a single vector, which represents the center point of the bounding sphere, and a distance from the center to the edge (radius) of the sphere. The syntax and parameters for **D3DXComputeBoundingSphere** follow, then some sample code.

```

HRESULT WINAPI D3DXComputeBoundingSphere(
    const D3DXVECTOR3 *pFirstPosition,
    DWORD NumVertices,
    DWORD dwStride,
    D3DXVECTOR3 *pCenter,
    FLOAT *pRadius
);

```

const D3DXVECTOR3 *pFirstPosition

Pointer to the first vertex in the vertex buffer.

DWORD NumVertices

Number of vertices in the vertex buffer.

DWORD dwStride

Size in bytes of a single vertex.

D3DXVECTOR3 *pCenter

Address of a **D3DXVECTOR3** structure to receive the center of the sphere.

FLOAT *pRadius

Address of a **FLOAT** to receive the radius of the sphere.

Sample code:

```
LPDIRECT3DVERTEXBUFFER9 VertexBuffer = NULL;
D3DXVECTOR3* Vertices = NULL;
D3DXVECTOR3 Center;
FLOAT Radius;
DWORD FVFVertexSize = D3DXGetFVFVertexSize(g_pMesh->GetFVF());
g_pMesh->GetVertexBuffer(&VertexBuffer);
VertexBuffer->Lock(0,0, (VOID**) &Vertices, D3DLOCK_DISCARD);
D3DXComputeBoundingSphere(Vertices, g_pMesh->GetNumVertices(),
                          FVFVertexSize, &Center, &Radius);
VertexBuffer->Unlock();
```

Rays Intersecting Meshes

Another useful thing to know is how to determine whether a ray intersects a mesh. When I say *ray* I mean an infinite line in 3D space. It has a start point and a normalized direction vector, indicating in which direction the ray extends. It'll also be useful to calculate at exactly which point a ray intersects a mesh. This is especially helpful if you're creating a shooting game and want to know whether a character has been hit by a weapon like a ray gun. See Figure 7.5 to see what I mean.

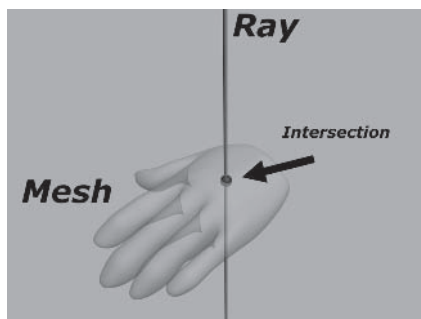


Figure 7.5

To determine whether a ray intersects a mesh you call the **D3DXIntersect** function. This function requires several arguments, including a pointer to the mesh to check for intersection, the starting point of the ray, and the direction of the ray. This

function returns a Boolean value indicating whether an intersection occurred. It also can return additional values, such as the distance along the ray where intersection occurs and an array of points representing every location on the mesh where intersection occurs. The syntax and parameters for `D3DXIntersect` follow. Then some sample code.

```
HRESULT WINAPI D3DXIntersect(
    LPD3DXBASEMESH pMesh,
    const D3DXVECTOR3 *pRayPos,
    const D3DXVECTOR3 *pRayDir,
    BOOL *pHit,
    DWORD *pFaceIndex,
    FLOAT *pU,
    FLOAT *pV,
    FLOAT *pDist,
    LPD3DXBUFFER *ppAllHits,
    DWORD *pCountOfHits
);
```

LPD3DXBASEMESH pMesh

Pointer to a mesh to check for intersection.

const D3DXVECTOR3 *pRayPos

Starting point of the ray.

const D3DXVECTOR3 *pRayDir

Normalized vector representing the direction of the ray.

BOOL *pHit

Address of a Boolean to receive True or False, indicating whether the ray intersects the mesh.

DWORD *pFaceIndex

If an intersection occurs, this value indicates which polygon the ray passes through.

FLOAT *pU

Pointer to a barycentric hit coordinate. This value is beyond the scope of this book.

FLOAT *pV

Pointer to a barycentric hit coordinate. This value is beyond the scope of this book.

FLOAT *pDist

Distance along the ray until the first intersection occurs. (Remember, the ray may pass through multiple places on the mesh.) With this value you can work out the exact location of the first intersection by using this formula:

$$\text{HitPoint} = \text{RayStart} + (\text{RayDir} * \text{pDist})$$

Check out the sample code to see this in action.

LPD3DXBUFFER *ppAllHits

This value will typically be NULL if you're only interested in the first point of intersection on the mesh. Most of the time this will be enough; however, if you need every place of intersection you can pass the address of an ID3DXBuffer interface to receive an array of intersection points. This will be array of D3DXINTERSECTINFO structures. This structure contains the face index, barycentric coordinates, and the distance along the ray until the point of intersection. D3DXINTERSECTINFO looks like this:

```
typedef struct _D3DXINTERSECTINFO {
    DWORD FaceIndex;
    FLOAT U;
    FLOAT V;
    FLOAT Dist;
} D3DXINTERSECTINFO, *LPD3DXINTERSECTINFO;
```

DWORD *pCountOfHits

Can be NULL; however, if you pass the address of a DWORD, you'll receive the number of intersections between the ray and the mesh.

Sample code:

```
D3DXVECTOR3 RayPos(0,0,0);
D3DXVECTOR3 RayDir(0,0,1.0f);
BOOL bHit = false;
```

```

DWORD FaceIndex = 0;
FLOAT pU = 0;
FLOAT pV = 0;
FLOAT pDist = 0;
D3DXVECTOR3 IntersectPoint(0,0,0);

D3DXIntersect(Mesh, &RayPos, &RayDir, &bHit, &FaceIndex,
              &pU, &pV, &pDist, NULL, NULL);

if(bHit)
    IntersectPoint = RayPos + (RayDir * pDist);

```

Vertex Interpolation

Imagine this... You have two meshes of a wizard with a cape, hat, and a long white beard. One of the meshes (mesh 1) is of the wizard with his arms outstretched and the other (mesh 2) is of the wizard with his arms by his sides. Now, using these two meshes you can calculate every pose of the mesh between these two positions (keyframes). For example, I could dynamically generate a wizard mesh with his arms half outstretched — halfway between mesh 1 and mesh 2.

You could even gradually change the mesh's posture over time and animate it from one pose to the next (Fig. 7.6). This process is known as *vertex interpolation*, because we're interpolating from

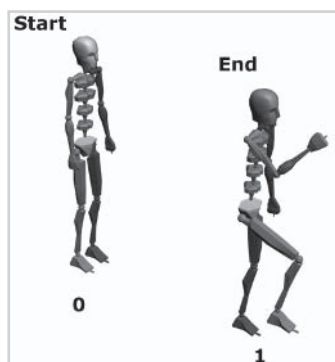


Figure 7.6

one mesh state to another. It's actually a very simple thing to do. You simply need to generate a scalar between 1 and 0, like 0.5, and using a clever little DirectX function you can generate a pose between mesh 1 and mesh 2. That's right, it's *that* simple. So 0.5 would create a mesh halfway between mesh 1 and mesh 2, 1.0 would generate the final mesh, and 0 would be the starting mesh. In short, every value between 0 and 1 builds a mesh that much closer to the destination mesh from the starting mesh.

*** NOTE.** The one rule you must keep in mind when interpolating between two meshes is that both meshes must have the same number of vertices.

The function to interpolate between two vertices is **D3DXVec3Lerp**. It requires four parameters: a vector where the result is to be returned, a starting vector, an ending vector, and a scalar that is the interpolation factor. So, if I wanted to build a vector that was three-quarters between the start and end vectors, the scalar would be 0.75. The syntax and parameters for D3DXVec3Lerp follow.

```
D3DXVECTOR3 *D3DXVec3Lerp(
    D3DXVECTOR3 *pOut,
    CONST D3DXVECTOR3 *pV1,
    CONST D3DXVECTOR3 *pV2,
    FLOAT s
);
```

D3DXVECTOR3 *pOut

Address to receive the resultant vector.

CONST D3DXVECTOR3 *pV1

First vector from where interpolation should begin.

CONST D3DXVECTOR3 *pV2

End vector where interpolation may end.

FLOAT s

Scalar that determines the interpolation factor between the start and end vectors.

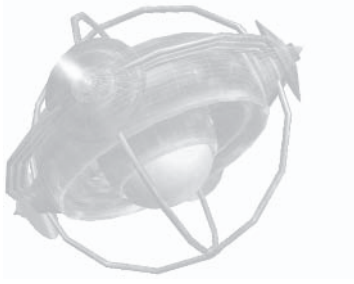
You may notice that `D3DXVec3Lerp` does not interpolate between two meshes but rather between two vectors. However, it gives us the potential to interpolate between two meshes. We simply need to lock the vertex buffers for each mesh, then cycle through every vertex and, one by one, interpolate every vertex and store the result in the vertices of a cloned mesh, which will become the final, interpolated mesh. The code below deals with three vectors (the source vector, the destination vector, and a final resulting vector), then it interpolates halfway between them.

```
D3DXVECTOR3 Source(0,0,0);  
D3DXVECTOR3 Destination(10,10,10);  
D3DXVECTOR3 Result(0,0,0);  
  
D3DXVec3Lerp(&Result, &Source, &Destination, 0.5f);
```

Conclusion

This chapter presented an overview on how to deal with meshes. Specifically it has explained how to load meshes from an X file or from X file data objects. Furthermore, it investigated how you can cycle through the vertices of a mesh, how you can set a mesh's FVF, and, finally, how you can interpolate between two meshes.

The next chapter explores 3D worlds and the view matrix further. To be more exact, it explains how to create a first-person camera with some nice and advanced features.



Chapter 8

Cameras — First-Person and More

I mentioned earlier that cameras are controlled by the view matrix. Until now, we have seen only how to move the camera to specific locations and look at certain points in 3D space using the `D3DXMatrixLookAtLH` function. This chapter expands on this knowledge by implementing a fully functional 3D camera, which is the kind of first-person camera used in games like *Doom*, *Descent*, and *Unreal*. Once completed, the camera will be encapsulated into a single class featuring the following:

- The ability to look up, down, left, and right
- The ability to move forward and backward, and strafe left and right
- The ability to determine whether meshes and other geometry are in the camera's view

The Problem

Imagine for a moment that you wanted to create a first-person game — in other words, a game where the player sees through the eyes of a character. You already know how 3D space is measured, how positions are set using vectors, and how rotations are specified in radians. Therefore, you may already have some ideas on how to create such a camera. Initially, it seems a good idea to maintain a position vector for the camera and to continually throw this at the `D3DXMatrixLookAtLH` function whenever it changes, as the player moves around. For example, if the player moves forward three units you might increment his position's *Z* component by three; in other words, three units forward. Likewise, if the player side-stepped (strafed) left five units, you'd expect to decrement his position's *X* component by five. So far so good, but there's a terrible problem you're likely to run into should you adopt this method. To illustrate, picture the following scenario: The player moves forward by five, sidesteps by three, rotates around the *Y* axis by $-\pi/2$ radians (turns left) and is now facing a new direction, and finally moves forward by another five units. Notice that once rotation has occurred, the player is no longer moving exclusively along the *Z* axis. Instead, moving forward affects movement along both the *X* and the *Z* axes. This is because the direction in which he's now facing is at an angle. The problem would be even trickier were this a space simulator where you can also move up and down, look around, and move in a potentially infinite number of directions. We'll address these issues as we create a camera that can accommodate all these scenarios and more.

Overview

Probably the best way to visualize a first-person camera is to think of the human head. You may recall from biology lessons at school an illustration of a human head with three arrows pointing in different directions. These arrows are all perpendicular to each other and together form our axes of orientation. There is one arrow pointing straight ahead in the direction we're looking, known as the direction or look at vector; one arrow pointing directly to the right, called the right vector; and one arrow pointing straight upward, unsurprisingly named the up vector. Finally, the origin — the point where these axes intersect — represents the position of the camera in 3D space (i.e., the place you're standing).

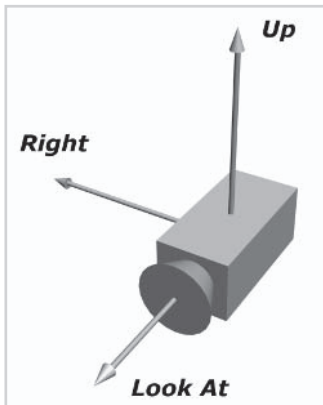


Figure 8.1:
First-person
camera

Looking Around

Three axes extending outward from our position in 3D space (up, right, and look at) have now been defined. Using these axes it is possible to rotate the camera — our head — in any direction by an arbitrary number of radians and to move the camera in the direction we’re facing. First, let’s examine rotation. Because we have three axes there are essentially three possible ways we can rotate. We can rotate around the look at axis, the up axis, and the right axis, or any combination of these, and the axis we rotate about will influence where the camera will face. The next three sections examine the different types of rotation.

Pitch

For the moment, pretend you’re a first-person camera. Start by looking straight in front of you. Now, rotate your head to look down about 45 degrees. Rotate back up by 45 degrees to look in front of you again. Finally, rotate your head to look upward by 45 degrees. In doing this you have effectively rotated your view about the right axis — the axis sticking out the side of your head — by both positive and negative quantities. Rotation about this axis is known as pitch; so, to pitch your head by some angle — say 45 degrees — means to rotate it up or down by 45 degrees, around the right axis.

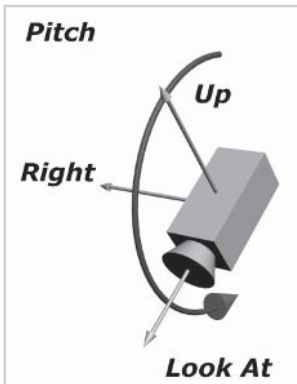


Figure 8.2

Roll

Facing straight in front of you again, tilt your head to the right by 45 degrees, and then back again, looking straight ahead. Finally, roll your head the other way, to the left by 45 degrees. Here, you have rotated your head about the look at axis. This is known as roll.

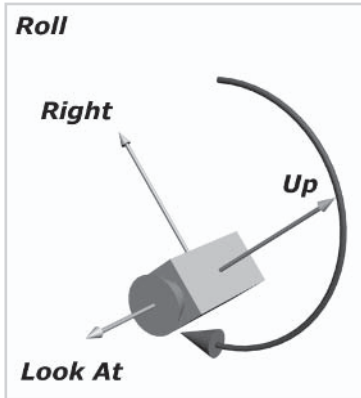


Figure 8.3

Yaw

The final form of rotation occurs about the up axis and involves you turning your head left or right. Start again facing forward and look left. Now look ahead again, then look right. This kind of rotation is known as yaw.

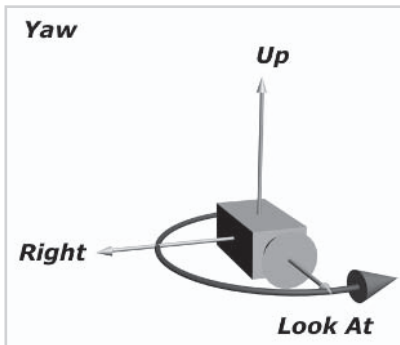


Figure 8.4

Combining Rotations

You now understand what it means to rotate a camera in terms of pitch, roll, and yaw. Pitch is to look up and down, roll is to tilt the view one way or the other, and yaw is to look left and right. In the context of a camera, several things occur at the time of rotation. For example, you're standing at the origin and are looking straight ahead. Our camera vectors begin normalized. Thus, your position is $(0,0,0)$, your look at vector is $(0,0,1)$, your right vector is $(1,0,0)$, and your up vector is $(0,1,0)$. Simple so far. Now, we want to look up by 45 degrees (pitch) and we then want to look right by 45 degrees (yaw). These two rotations can be seen as follows.

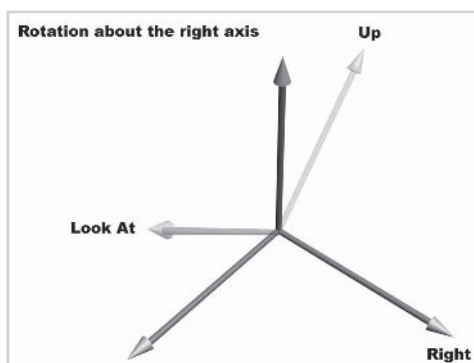


Figure 8.5

Notice what occurs to the vectors. By pitching about the right axis in our first transformation we are affecting our other two vectors — up and look at. To put it another way, the right vector is the only unaffected vector here; the other two vectors (axes) are *rotated* about the right vector. For example, if I look down, I am no longer looking straight ahead; thus, my look at vector changes because I'm facing a new location. Likewise, my up vector — pointing from the top of my head — also changes because since I'm looking downward it can no longer be pointing directly upward like it was before. It rotates. Hence, a pattern begins to emerge. We can conclude that, when rotating about an axis, it is the other two axes that are transformed about that axis; the axis of rotation itself remains unchanged. In short, rotation involves transforming two perpendicular vectors about a common axis by a specified number of radians.

Moving the Camera

In theory we now know how to rotate a camera around any axis by a specified number of radians using the ideas of pitch, roll, and yaw. This section examines how to move the camera around; for example, moving it forward. Previously, it seemed like a good idea to move the camera forward by incrementing the Z component of our position vector. But we then realized that if the camera were facing an angle its movement would involve changing more than simply the Z component. A forward movement like this would require an increment of the Z component as well as a change to X and, if looking up, it could involve incrementing Y too.

To move forward therefore, we effectively want to move in the direction we're facing by a specified amount rather than by an absolute position. In other words, we want to move in the direction of our look at vector at some speed. We actually know how to do this already using basic vector mathematics. Let's imagine we're standing at (0,0,0) and we're facing (5,0,5). We want to move forward by one unit in the direction we're facing. To compute our new position, we do the following:

1. Subtract the look at vector from our position vector.

Example: $(5,0,5) - (0,0,0)$

2. Normalize the resultant vector.
3. Multiply the normalized vector by the speed.
4. Add the resultant vector to the current position.

You see the magic here? We're effectively saying, "We have a direction, and to move forward we simply scale the direction by some amount to move and finally add this onto our current position."

We can apply this movement concept to any of our other vectors too. For example, to strafe right by some amount, we simply scale the right vector — as we did with our look at vector — and add it to our current position. To walk backward or to move in a negative direction we can simply subtract from our current position, rather than add.

Making the Camera

All of this theory is no doubt whetting your appetite and you're probably eager to get into coding this camera class. The following declaration is a good place to begin. For convenience, I have not included the method declarations (except one), just the class members. The one method that is shown will be called on each frame to build a view matrix according to axis information — up, look at, right, etc. The following sections examine this class and its methods in more detail.

```
class CXCamera
{
    protected:
        //Stores the position and three vectors
        D3DXVECTOR3 m_Position;
        D3DXVECTOR3 m_LookAt;
        D3DXVECTOR3 m_Right;
        D3DXVECTOR3 m_Up;

        //Stores whether the camera has changed since last update.
        //By change, we mean whether the camera has moved
        //or rotated. If so, we'll recompute the matrix.
        bool m_bChanged;

        //Stores the rotation to apply to a given axis
        float m_RotateAroundUp;
        float m_RotateAroundRight;
        float m_RotateAroundLookAt;

        //Final view transformation matrix
        D3DXMATRIX m_MatView;

        //Called each frame to update the total view matrix
        VOID Update();
};
```

Initializing the Camera Class

The first thing to do, before any of this mathematics stuff and before creating any functions, is to ensure our class members are set to starting values. This process will occur in the class constructor, when the class is created in memory. Here we set our position to (0,0,0), and our look at, right, and up vectors to equally appropriate values. This can be seen below.

```
CXCamera::CXCamera()
{
    m_Position = D3DXVECTOR3(0.0f,0.0f,0.0f); //Set position to 0,0,0
    m_LookAt = D3DXVECTOR3(0.0f,0.0f,1.0f); //Set look at to 0,0,1
    m_Right = D3DXVECTOR3(1.0f,0.0f,0.0f); //Set right to 1,0,0
    m_Up = D3DXVECTOR3(0.0f,1.0f,0.0f); //Set up to 0,1,0
    m_RotateAroundUp = m_RotateAroundRight = m_RotateAroundLookAt = 0;
    D3DXMatrixIdentity(&m_matView);
}
```

Moving the Camera

In terms of moving the camera, we want to be able to do several things: move forward and backward, sidestep left and right, and set the camera to some arbitrary position. Remember, moving forward in any direction does not mean just moving along the Z axis. The camera could be at an angle, in which case movement affects other axes too. Once a position has been set, you'll notice our methods below, then set class member `m_bChanged` to `True`. This is a flag. A value of `True` means a change of position or rotation has occurred and the view matrix should be updated on the next frame. This process will occur in the `Update` method, shown later. For now, the class methods to achieve movement follow:

```
void CXCamera::SetPosition(D3DXVECTOR3 *Pos)
{
    m_Position = *Pos;
    m_bChanged= true;
}
```



```

void CXCamera::MoveForward(float Dist)
{
    m_Position += Dist*m_LookAt;
    m_bChanged = true;
}

void CXCamera::MoveRight(float Dist)
{
    m_Position += Dist*m_Right;
    m_bChanged = true;
}

void CXCamera::MoveUp(float Dist)
{
    m_Position += Dist*m_Up;
    m_bChanged = true;
}

void CXCamera::MoveInDirection(float Dist, D3DXVECTOR3 *Dir)
{
    m_Position += Dist*(*Dir);
    m_bChanged = true;
}

```

Rotating the Camera

Rotation occurs in radians about any of the three axes, as mentioned previously. You can rotate around the right axis, the look at axis, and the up axis; these rotations are known as pitch, roll, and yaw. Like before, the following rotation methods set our member variables ready for rotation, and afterward set member `m_bChanged` to `True`. Then, the view matrix is constructed on the next frame, using `Update`. The following methods prepare the class for such rotations:

```

void CCamera::RotateDown(float Angle)
{
    m_RotateAroundRight += Angle;
    m_bChanged = true;
}

```

```

void CCamera::RotateRight(float Angle)
{
    m_RotateAroundUp += Angle;
    m_bChanged = true;
}

void CCamera::Roll(float Angle)
{
    m_RotateAroundLookAt += Angle;
    m_bChanged = true;
}

```

Building the View Matrix

The most important process occurs in the **Update** method, which is intended to be called on each frame. If the position or rotation of the camera hasn't changed since Update was last called, then the view matrix doesn't need to be rebuilt and the existing view matrix can safely be used. If, however, `m_bChanged` has been set to `True`, then the view matrix needs to be rebuilt. Rebuilding the view matrix basically means telling Direct3D where we're standing and where we're facing, and we can specify this using our member vectors. Previously, we used the `D3DXMatrixLookAtLH` function to build the view matrix. For this process, however, we'll need to build the view matrix manually. The following code shows the definition for the Update method and an explanation follows.

```

bool CXCamera::Update()
{
    if(m_bChanged)
    {
        //Matrices to store the transformations about our axes
        D3DXMATRIX MatTotal;
        D3DXMATRIX MatRotateAroundRight;
        D3DXMATRIX MatRotateAroundUp;
        D3DXMATRIX MatRotateAroundLookAt;
    }
}

```

```

//Get the matrix for each rotation
D3DXMatrixRotationAxis(&MatRotateAroundRight,
                      &m_Right, m_RotateAroundRight);

D3DXMatrixRotationAxis(&MatRotateAroundUp,
                      &m_Up, m_RotateAroundUp);

D3DXMatrixRotationAxis(&MatRotateAroundLookAt,
                      &m_LookAt, m_RotateAroundLookAt);

//Combine the transformations into one matrix

D3DXMatrixMultiply(&MatTotal, &MatRotateAroundUp,
                  &MatRotateAroundRight);

D3DXMatrixMultiply(&MatTotal, &MatRotateAroundLookAt,
                  &MatTotal);

//Transforms two vectors by our matrix and computes the third by
//cross product

D3DXVec3TransformCoord(&m_Right, &m_Right, &MatTotal);
D3DXVec3TransformCoord(&m_Up, &m_Up, &MatTotal);
D3DXVec3Cross(&m_LookAt, &m_Right, &m_Up);

//Check to ensure vectors are perpendicular
if (fabs(D3DXVec3Dot(&m_Up, &m_Right)) > 0.01)
{
    //If they're not
    D3DXVec3Cross(&m_Up, &m_LookAt, &m_Right);
}

//Normalize our vectors
D3DXVec3Normalize(&m_Right, &m_Right);
D3DXVec3Normalize(&m_Up, &m_Up);
D3DXVec3Normalize(&m_LookAt, &m_LookAt);

//Compute the bottom row of the view matrix
float fView41, fView42, fView43;
fView41 = -D3DXVec3Dot(&m_Right, &m_Position);
fView42 = -D3DXVec3Dot(&m_Up, &m_Position);

```

```

        fView43 = -D3DXVec3Dot(&m_LookAt, &m_Position);

        //Fill in the view matrix
        m_matView = D3DXMATRIX(m_Right.x, m_Up.x, m_LookAt.x, 0.0f,
                                m_Right.y, m_Up.y, m_LookAt.y, 0.0f,
                                m_Right.z, m_Up.z, m_LookAt.z, 0.0f,
                                fView41, fView42, fView43, 1.0f);

    }

    //Set view transform
    m_pDevice->SetTransform(D3DTS_VIEW, &m_matView);

    //Reset update members
    m_RotateAroundRight = m_RotateAroundUp = m_RotateAroundLookAt = 0.0f;
    m_bChanged = false;
}

```

Although this method is comparatively long it is generally clear and simple. It can be broken down into the following stages:

1. It begins by determining whether the Boolean member `m_bChanged` is `True`.
2. If not, then we transform the 3D device using `SetTransform` and passing it our old view matrix. Otherwise, we need to rebuild the view matrix and then set our device using our new version.
3. The rebuilding process begins by calling `D3DXMatrixRotationAxis`. This will create a transformation matrix to rotate geometry about a specified axis by a specified number of radians. In our case, we call it three times — once for each axis (up, right, and look at) — and we'll pass our rotation values for each axis. The value for each may be 0, or some other value if the camera has actually rotated since the last frame.
4. Once a rotation matrix has been constructed for each axis, the transformations are then combined into a single transformation. This is accomplished by multiplying the matrices. The resultant matrix is stored in local variable `MatTotal`.

5. Our axes of orientation — right, up, and look at — are then transformed by our rotation matrices using `D3DXVec3TransformCoord`. In other words, they are actually rotated by the combined matrix to a new orientation. The third vector is generated by the cross product of the other two. Effectively, at this point our axes have been adjusted to their new orientations, based on the rotation of the camera. However, we still need to construct a view matrix to pass on to our 3D device.
6. Before building the view matrix we check to see that our vectors are perpendicular to one another. Due to floating-point mathematics, it's possible that very long decimals may have been truncated and over time have grown more inaccurate. Thus, we need to perform this test and, should they be found not perpendicular, we must regenerate another vector by cross product.
7. In preparation for constructing the view matrix, we then normalize our vectors using `D3DXVec3Normalize`.
8. Here, the process for building the view matrix begins. This starts by computing the bottom row — the negative dot product between the position and every other vector. Afterward, this process continues by filling in the matrix with the right, up, and look at vector coordinates until the view matrix is completed.

Test Drive

The hard work pays off here. You can now take this camera for a test whirl around a 3D world. You have seen from previous chapters how to set up an environment, load a few models, and set some lights. So go ahead and place this camera right in the center of things, move it around, and take a tour.

Viewing Frustum

At the beginning of this chapter I promised to explain how you can determine whether objects are visible to the camera. For example, if you move your camera about a 3D world where there are 3D models like castles, doors, monsters, etc., it'll be useful to know what is in the camera's view and what is not. We can assume that objects visible to the camera are situated in front of it and are not too distant from it, and that objects not visible to the camera are either behind it or are too far from it. Ultimately, we want to be able to ask questions like, "Hey, camera, can you see this mesh?" and receive an accurate answer. This section explains how to achieve this.

It should be obvious from Figure 8.6 and your own knowledge that the camera can only see whatever is contained inside the 3D volume in front of its lens. For most cases, this 3D volume can be thought of as an elongated pyramid, one whose smallest point is protruding from the camera's lens. If an object is inside the pyramid, it is visible, and everything outside cannot be seen. This pyramid is known as the *viewing frustum*.

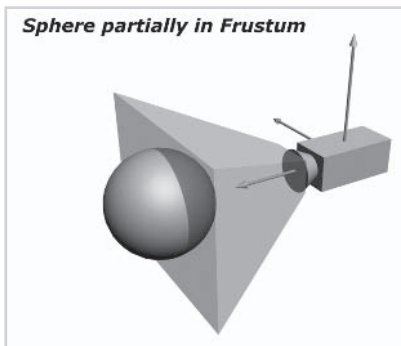


Figure 8.6

From a mathematical perspective, the viewing frustum is represented by six planes, with each plane representing one side of the frustum's extents. These planes are known as the *clipping planes*. One defines the far clipping plane — the horizon plane beyond which objects are not visible — and another is the near clipping plane — objects in front of this are not visible because they are

behind the camera. The other planes are the bottom, top, left, and right sides between the near and far clipping planes. The entire frustum is shown in Figure 8.6.

***NOTE.** Planes are explained in Chapter 3.

You may remember from earlier in this book that planes are like infinitely long sheets of paper. Using planes, we can classify points. Specifically, we can determine whether a point in 3D space resides on the plane, in front of the plane, or behind the plane. In the context of the frustum, therefore, we can test whether a point resides behind the near clipping plane and in front of the far clipping plane. Similarly, we can also test if a point resides between each of the other sides. If it does, then the point is in the frustum's volume (visible), and if not, the point is outside the frustum (not visible).

If we then extend this technique to test three points, we can effectively determine whether a polygon resides in the frustum — fully, partially, or not at all — and, at a higher level, we can then determine whether a mesh is visible. The next section concentrates on adding functionality to the existing camera class to encapsulate the frustum. We'll also add some other methods to test whether a specified point, cube, sphere, and mesh are in the frustum.

Constructing the Frustum

To ensure our camera class can also behave as the frustum, we'll need to add some members to the class. These will be `m_Planes[6]` — a plane representing each side of the frustum — and each must be initialized in the class constructor. We must also multiply our view matrix by our projection matrix. The resultant matrix will contain the coordinates needed to generate our planes at the right place and orientation. Initialization can be seen below.

```
D3DXMATRIX Matrix, ViewMatrix, ProjectionMatrix;

pDevice->GetTransform(D3DTS_VIEW, &ViewMatrix);
pDevice->GetTransform(D3DTS_PROJECTION, &ProjectionMatrix);
```

```

D3DXMatrixMultiply(&Matrix, &ViewMatrix, &ProjectionMatrix);

m_Planes[0].a = Matrix._14 + Matrix._13;
m_Planes[0].b = Matrix._24 + Matrix._23;
m_Planes[0].c = Matrix._34 + Matrix._33;
m_Planes[0].d = Matrix._44 + Matrix._43;
D3DXPlaneNormalize(&m_Planes[0], &m_Planes[0]);

m_Planes[1].a = Matrix._14 - Matrix._13;
m_Planes[1].b = Matrix._24 - Matrix._23;
m_Planes[1].c = Matrix._34 - Matrix._33;
m_Planes[1].d = Matrix._44 - Matrix._43;
D3DXPlaneNormalize(&m_Planes[1], &m_Planes[1]);

m_Planes[2].a = Matrix._14 + Matrix._11;
m_Planes[2].b = Matrix._24 + Matrix._21;
m_Planes[2].c = Matrix._34 + Matrix._31;
m_Planes[2].d = Matrix._44 + Matrix._41;
D3DXPlaneNormalize(&m_Planes[2], &m_Planes[2]);

m_Planes[3].a = Matrix._14 - Matrix._11;
m_Planes[3].b = Matrix._24 - Matrix._21;
m_Planes[3].c = Matrix._34 - Matrix._31;
m_Planes[3].d = Matrix._44 - Matrix._41;
D3DXPlaneNormalize(&m_Planes[3], &m_Planes[3]);

m_Planes[4].a = Matrix._14 - Matrix._12;
m_Planes[4].b = Matrix._24 - Matrix._22;
m_Planes[4].c = Matrix._34 - Matrix._32;
m_Planes[4].d = Matrix._44 - Matrix._42;
D3DXPlaneNormalize(&m_Planes[4], &m_Planes[4]);

m_Planes[5].a = Matrix._14 + Matrix._12;
m_Planes[5].b = Matrix._24 + Matrix._22;
m_Planes[5].c = Matrix._34 + Matrix._32;
m_Planes[5].d = Matrix._44 + Matrix._42;
D3DXPlaneNormalize(&m_Planes[5], &m_Planes[5]);

```

That's all we need to do. Now we can test whether a point, a polygon, a cube, or anything else intersects the frustum.

Testing for a Point

Testing for whether a point is on, behind, or before a plane or group of planes is what this section is about. Is a point in the frustum? The following code uses the `D3DXPlaneDotCoord` function to find out.

```
bool CXCamera::TestPoint(FLOAT XPOS, FLOAT YPOS, FLOAT ZPOS)
{
    for(short Counter = 0; Counter < 6; Counter++)
    {
        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XPOS, YPOS, ZPOS)) < 0.0f)
            return false;
    }

    return true;
}
```

Testing for a Cube

A *cube* is a 3D primitive that can be defined by eight points, one for each corner. To test whether a cube is visible (in the frustum) we can use the following function.

```
bool CXCamera::TestCube(float XCenter, float YCenter, float ZCenter,
                        float Size)
{
    for(short Counter = 0; Counter < 6; Counter++)
    {
        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter - Size, YCenter - Size,
                                             ZCenter - Size)) >= 0.0f)
            continue;

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter + Size, YCenter - Size,
                                             ZCenter - Size)) >= 0.0f)
            continue;
    }
}
```

```

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter - Size, YCenter + Size,
                                             ZCenter - Size)) >= 0.0f)
            continue;

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter + Size, YCenter + Size,
                                             ZCenter - Size)) >= 0.0f)
            continue;

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter - Size, YCenter - Size,
                                             ZCenter + Size)) >= 0.0f)
            continue;

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter + Size, YCenter - Size,
                                             ZCenter + Size)) >= 0.0f)
            continue;

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter - Size, YCenter + Size,
                                             ZCenter + Size)) >= 0.0f)
            continue;

        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                             &D3DXVECTOR3(XCenter + Size, YCenter + Size,
                                             ZCenter + Size)) >= 0.0f)
            continue;

        return false;
    }

    return true;
}

```

Testing for a Sphere

A sphere can be defined by a center point and a radius. To determine whether a sphere intersects the frustum, we simply test to see whether the `D3DXPlaneDotCoord` function returns a value greater than the sphere's negative radius. The following code achieves this:

```
bool CXCamera::TestSphere(float XCenter, float YCenter, float ZCenter,
                          float Radius)
{
    for(short Counter = 0; Counter < 6; Counter++)
    {
        if(D3DXPlaneDotCoord(&m_Planes[Counter],
                            &D3DXVECTOR3(XCenter, YCenter, ZCenter))
            < -Radius)
            return false;
    }

    return true;
}
```

Testing for a Mesh

A good way to check a mesh's visibility is to test whether its bounding sphere intersects the frustum. The `D3DXComputeBoundingSphere` function calculates the bounding sphere of a mesh, as mentioned previously. Thus, a function to determine the visibility of a specified mesh can be coded as below.

```
bool CXCamera::TestMesh(LPD3DXBASEMESH pMesh)
{
    if(pMesh)
    {
        DWORD NumVertices = pMesh->GetNumVertices();
        DWORD FVF = pMesh->GetFVF();
        UINT FVFSize = D3DXGetFVFVertexSize(FVF);
        LPVOID ppData = NULL;
```

```
pMesh->LockVertexBuffer(D3DLOCK_READONLY, &ppData);

if(ppData)
{
    D3DXVECTOR3 Center(0,0,0);
    FLOAT Radius = 0;

    D3DXComputeBoundingSphere((D3DXVECTOR3*)ppData,
                              NumVertices, FVFSize, &Center, &Radius);

    if(TestSphere(Center.x, Center.y, Center.z, Radius))
    {
        pMesh->UnlockVertexBuffer();
        return true;
    }
}

pMesh->UnlockVertexBuffer();
}

return false;
}
```

This page intentionally left blank.



Chapter 9

Timing and Animation

This chapter begins the topic of animation in Direct3D. Animation is discussed throughout the remainder of this book. This chapter aims to explain the basics behind animation. Specifically, it will teach you:

- What animation is
- What keyframed animation is
- Timing in animation
- Animation and frames

Animation is about how things change over time. It doesn't matter how much something changes and it doesn't matter how long it takes, but if something changes over time it can be said to be animating. In computer games, animation is no different. If a game character is walking around the screen killing baddies or doing something else, then the character is being animated. Ultimately, the character is probably animating in two ways: 1) It's moving its arms and firing a gun, or 2) It's walking around the screen. Ultimately, it gives you the illusion of movement and this enhances realism and your game-playing experience. After all, it would probably look quite dull if everything in your game was static and inanimate.

So how do we get started with animation? Well, first we'll need to discuss the most important aspect of animation: time. That's right, time. After all, animation won't work unless we establish some concept of time and some way to measure time.

Time

In animation, time is a unit of measurement. Animations have a start time and an end time, and every in-between action in an animation occurs at a specific time. Ultimately, time allows us to measure how long it has been since the animation started.

Timing is important to animations for many reasons. A long time ago, many games just played animations as fast as they could run, frame by frame, and everything seemed fine. However, there was a problem: Because the speeds of computers vary from machine to machine (some machines are fast and some are slow), it meant animations played at different speeds depending on which machine you used. Thus, different gamers had different experiences; for some the animations were too slow and jerky and for others the animations were too fast. Thus, a way to standardize the speed of animations began. The way to solve this problem is to use time as a reference for your animation. Time is constant; a second is exactly one second. One second is the same on every machine. So, by using time to measure and play your animations you ensure that every user will experience the same things.

There are many ways a programmer can measure time in an application. Although DirectX provides no function to measure time, we can use one that is included with Visual C++. I'm talking about the `timeGetTime` function. This function returns a `DWORD` value representing the elapsed time in milliseconds since Windows started. This value is not in itself useful, but since it acts like a counter we'll be able to use it to measure the relative time since an animation began. We'll look more closely at this later. The following code shows you how this function is used.

```
DWORD StartTime = timeGetTime();
```

Once you have a time value from the `timeGetTime` function, you can call it again, at a later time, and subtract the two times to find out the elapsed time in milliseconds, like this:

```
DWORD ElapsedTime = timeGetTime() - StartTime;
```

So, by logging the start time of any animation, you can subsequently call the `timeGetTime` function and subtract the start time from it to work out how far into the animation you've reached (in milliseconds).

*** NOTE.** To convert milliseconds into seconds, you divide the time by 1000, like this:

```
TimeInSeconds = TimeInMilliseconds/1000
```

To convert seconds into milliseconds, you multiply the time by 1000, like this:

```
TimeInMilliseconds = TimeInSeconds * 1000
```

Another nice trick you can do using `timeGetTime` for animation timing is to represent, using a scalar, how far into an animation you are. For example, let's say 0 represents the beginning of the animation, 1 represents the end, and any value between represents how close to the end you've reached. So, halfway into the animation would be 0.5. To compute this scalar you use the following formula:

```
Scalar = CurrentTime / (EndTime - StartTime)
```

*** NOTE.** We'll see more on how this is useful later in the chapter.

So now you know how time can be measured. That's good. But we still want to apply it to animation specifically. We'll cover this in the next section as we explore further how an application implements animation. There are many ways a program can implement animation; some are simple and some are complex. The next section begins by looking at a simple technique called keyframe animation.

Keyframe Animation

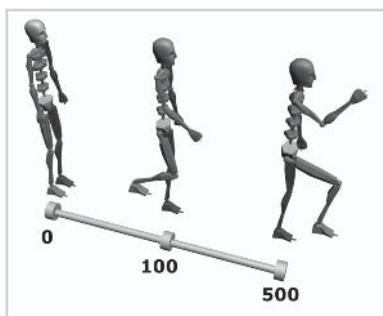


Figure 9.1

One of the most common and useful animation techniques is keyframe animation. This works by keeping track of only key points in the animation (keyframes) and then linearly interpolating between those frames as time elapses. Consider Figure 9.1. This demonstrates keyframe animation. For example, let's say we have a monster that begins with his arms by his sides at frame 0. Then, by frame 100, his arms are stretched out in front of him. Frame 0 occurs at 0 milliseconds and frame 100 occurs at 5000 milliseconds (5 seconds). We therefore know that from frame 0 to 100 (within 5 seconds) the monster will move his arms from his sides to out-stretched, starting from the position in frame 0 and moving closer to frame 100, frame by frame. In this case, we don't need to store how the monster will look in every frame because we already know the intervening movements involved. Instead, we can just store the positions for both frame 0 and frame 100, and we can simply compute the position of the mesh in all frames in between. This is why it's called keyframe animation; you only store the important, defining frames of the animation sequence.

Let's take another example and look at how it can be implemented practically from what we know already. For example, take a man who walks in a straight line. The animation begins at frame 0 (0 seconds) where the man is standing at position 0 on the X axis. The animation lasts 5 seconds (frame 100) and by this time he has completed his walk along the X axis line to position 10. We know, therefore, that the man spends the animation (100 frames, 5

seconds) walking along the line, step by step. We therefore know the following:

1 frame = 5 (seconds) / 100 (frames)

1 frame = 0.05 seconds

1 step (along the X axis) = 5 (seconds) / 10 (steps)

To walk 1 step = 0.5 seconds (10 frames)

This means by frame 10 the man has walked one step, and by frame 70 the man has walked to the seventh step, and finally by frame 100 he has walked to the end. That's fine, we know this, but we also want to be able to work this out mathematically so we can automatically update the position of the man as the animation elapses.

So how can we work out the position of the man on every frame? Simple. First we need to compute a scalar between 0 and 1, which represents how far into the animation we have reached. We saw the theory on how to do this in the previous section. In the context of this example we can compute this value as follows:

```
DWORD AnimLength = 5000;
DWORD CurrentAnimPos = timeGetTime() - AnimStart
FLOAT Scalar = AnimLength / CurrentAnimPos
```

Next, once a scalar has been computed (for example, this will be 0.5 if we're halfway through the animation) we need to use this value to calculate the position to which the character must travel. Let's assume the time into the animation is 3500, which means the scalar will be 0.7. To work out where along the 1 to 10 number line our character must be at this time, we simply multiply 10 (the total distance to move) by 0.7 (our scalar). This gives us 7. So 7 is the new position for the character. Easy stuff!

So, if the time was 5000 (the animation length) then our scalar would be 1, and 1 x 10 (on the number line) is 10. This is correct because it means our character is where he should be: at the end of the number line. Again, if the time was 0 (the beginning) then our scalar would be 0, and this puts our character right at the start of the number line.

And there we are; that's how keyframe animation works. Of course, in real life you won't be simply interpolating between just two keyframes. Full animation sequences may require you to interpolate between hundreds of keyframes. But no matter how many keyframes you use, the principle is the same. You're just interpolating between a start frame and an end frame.

Hierarchical Animation

Another useful animation technique for synchronously transforming a large number of different but related objects in an animation is known as *hierarchical animation*. Imagine this: You have a magic carpet that can fly around a scene. On top of this carpet you have a wizard who controls the carpet, and sitting next to him you have his pet monkey. Now, if the wizard or monkey moves on top of the carpet — if they walk around, for example — they do not affect the movement of the carpet itself. But, if the carpet moves — say it moves forward — then any object seated upon the carpet will naturally move with it. This may seem obvious but it poses a real problem for animation. It means that as the carpet moves, you must also animate the objects seated upon it. This is because the carpet and its passengers are related. So your application needs to keep track of all these relationships and ensure objects are animated according to their relationships. Now, there are many ways to handle this scenario; some are really, really easy and some are really, really hard. The simplest and best method is called hierarchical animation.

Using hierarchical animation every object in the animation (the carpet, the wizard, and the monkey) has its own transformation matrix (matrices were explained in earlier chapters). Furthermore, every object is part of a hierarchy — they have parent-child relationships. So, in this case, the carpet is the parent because its position and orientation affect its dependents (children), which are the wizard and monkey. As the carpet moves, so do all of its children. Examine Figure 9.2.

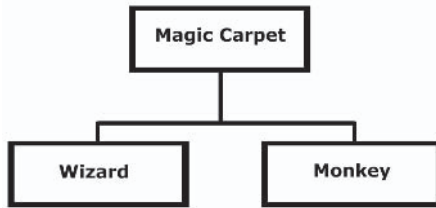


Figure 9.2

Now, when that carpet moves, wouldn't it be great if every object on the carpet updated its position automatically? This way, we wouldn't need to worry about which objects should be repositioned as a result of the magic carpet moving. This is exactly what the next section shows you how to do.

Linked Lists

The first thing we'll need to consider if we expect objects to keep track of their relationships with one another is how to represent these relationships in code. Just like the X file format, objects in our scene will have the potential to have none, one, or more children. Effectively, each object will need to maintain a list of child objects. In other words, how can we code a class to hold a list of child objects? Well, we could create a fixed array. But this requires we either know exactly how many children the class will hold or we must create a maximum sized array. If we choose the latter, it's likely a lot of memory will be wasted since most classes probably won't have that many children. Instead, it'd be better if we could have a dynamically sized list, one that shrinks and expands intelligently as objects are added or removed. There are many ways we could code such a list, but this chapter examines just one of these methods: a linked list.

A *linked list* is a linear list of items, one after the next. In a linked list every item keeps track of the next item in the list. It does this by keeping a pointer to the next item. If this pointer is NULL, then there is no next item, which means it's the last item in the list. Figure 9.3 shows how a linked list looks diagrammatically.

Some sample code follows, which shows two simple data structures set up to keep a linked list.

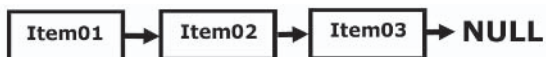


Figure 9.3

```

class CItem
{
    CItem* m_Next;
}

class CLinkedList
{
    CItem* m_Items;
}
  
```

Adding Items to the List

To add an item to the end of the list you simply cycle through every item from beginning to end, testing each item to see whether its next pointer is NULL. If so, then you've reached the last item. You then change the next pointer from NULL to point to the new item.

```

VOID CLinkedList::AddItem(CItem* Item)
{
    If(!Item)
        return;

    if(!m_Items)           //if list is empty
        m_Items = Item;    //Make first item
    else                   //Add to end of list
    {
        CItem* List = m_Items;
        while(!List->m_Next) //Find list end
            List = List->m_Next;

        List->m_Next = Item; //Add to end
    }
}
  
```

Clearing a Linked List

To clear the list of all items, you begin with the first item in the list. You keep a reference to the next item. You delete the first item and then move on to the next. Once there, you keep a reference to the next item, delete the current item, and move on. And the process continues.

```
VOID CLinkedList::ClearItems()
{
    CItem* Item = m_Items;

    While(Item)
    {
        CItem* Next = Item->m_Next; //Get next item
        delete Item;               //Delete the item
        Item = Next;               //Set next item
    }
}
```

Simple, isn't it? So now we've seen how a class can store a list of objects and, what's more, a list that is always just the right size for the number of items to hold. We'll use this linked list mechanism to store the parent-child relationship for objects in our scene, such as our magic carpet and wizard. The next section looks at the specific implementation details on how to do this.

Object Hierarchies for Animations

Now it's time to take our magic carpet example with the wizard and monkey riding on top and, using linked lists, animate them properly. This means that as the carpet moves, dependent objects follow. To do this, we'll create a data structure to encode our object hierarchy. There'll be one structure for each object. We'll call this structure a frame. It will contain a mesh pointer representing the actual object, like the magic carpet or wizard. Next, it will contain a transformation matrix that'll store the specific transformation for that mesh.

Then, we'll store an additional matrix that will represent the final transformation used to actually transform the mesh. It'll also contain a string to give the frame a name of our choosing. And finally, it'll contain a next pointer (so it can become part of a linked list) and it'll contain a pointer to its first child object, if it has any. Take a look at the code fragment below.

```
class CFrame
{
    LPD3DXMESH m_Mesh;
    D3DXMATRIX m_Matrix;
    D3DXMATRIX m_FinalTransformation;
    CString m_Name;
    Frame* m_Next;
    Frame* m_Children;
}
```

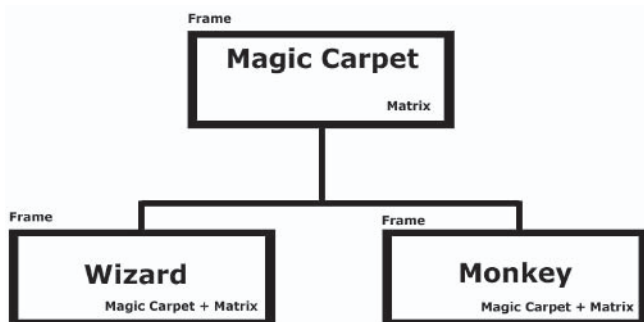


Figure 9.4

Before doing anything more, let's add some helper methods to this class to make dealing with frames simpler. We'll start by adding a search function. This method will take a frame name to search for, then it'll search that frame and all its children recursively until a match is found, and finally it'll return either a pointer to the matching frame or NULL if no frame is found.

```
CFrame CFrame::SearchFrame(CString Name)
{
    If(m_Name == Name)
        return this;
```

```

return NULL;

CItem* Item = m_Children;

While(Item)
{
    CItem* Next = Item->m_Next; //Get next item

    CFrame* Frame = Item->SearchFrame(Name);

    if(Frame)
        return Frame;

    Item = Next; //Set next item
}
}

```

Now we'll code an update function for our frame class. This will typically be called on every frame of an animation to update the transformations of all objects in the scene as the animation elapses. In other words, this update function updates the position and orientation of all objects in the scene. By calling this function on the topmost parent frame, it'll recursively update the transformation matrix of every frame in our hierarchy. The trick to making all child objects follow their parent is simple: You simply combine every object's transformation matrix with its parent's transformation matrix. It's really *that* easy. This means that any transformations applied to parent objects are always passed down to children and combined with the child's own individual transformation matrix.

***NOTE.** Remember, to combine two transformation matrices, you simply multiply them together.

Let's see an example of how transformations are iteratively passed down the hierarchy and specifically how they affect the transformations finally applied to child objects. To do this we'll first examine the Update method.


```

VOID CFrame::Update(D3DXMATRIX ParentTransform)
{
    m_FinalTransformation = ParentTransform * m_Matrix;

    While(Item)
    {
        CItem* Next = Item->m_Next; //Get next item
        Item->Update(m_FinalTransformation);
        Item = Next; //Set next item
    }
}

```

Now let's think about how this function works practically. First, the magic carpet will be in the top frame object and it will have two child frame objects: the wizard and the monkey. To move the carpet, and consequently all its child objects, you simply need to call the Update function on the magic carpet and pass it, as an argument, to a transformation matrix. The Update function will then automatically update every child, combining its individual transformations with its parent's. Easy.

***NOTE.** DirectX provides a premade frame class for you to use, called D3DXFRAME. Furthermore, it provides some utility functions to find frames, such as D3DXFrameFind, and to add children, such as D3DXFrameAppendChild.

Conclusion

This chapter has provided some insight into how animation works at a basic level. Using the knowledge featured in this chapter you can create many kinds of animations and can start making your games look very professional. The next chapter explores animation further by examining animated textures created with point sprites and particle systems.



Chapter 10

Point Sprites and Particle Systems

Hold on; don't give up! The name of this chapter makes it sound more complicated than it really is. This chapter discusses point sprites and particle systems. Basically, you'll learn how to include rain, snow, fog, sparks, and all kinds of other special effects in your games. Overall, this chapter explains the following:

- Direct3D point sprites
- Particle system properties
- Particle properties
- Amending vertex formats

Particle Systems Overview

So far this book has explained how to create 3D primitives and load in 3D meshes. Furthermore, it has explained how to create an advanced 3D camera that can move around a 3D world. OK, that's great, but wouldn't it be even better if you could walk around a world where it rained and snowed? Well, if you think so, then this chapter is for you.

Essentially a *particle system* is a collection of small pieces, or particles. In this case, a particle is something small like a raindrop, snowflake, or spark. The particle system is responsible for emitting

and controlling the particles, such as their movement, behavior, and lifetime. Now, we'll look at how Direct3D supports particle systems.

Particles in Direct3D — Point Sprites

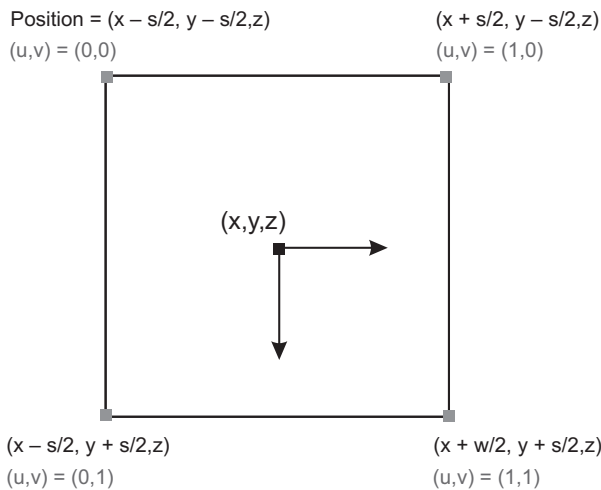


Figure 10.1

The previous section explained that a particle system is a collection of particles, and a particle is something small like a snowflake. Direct3D represents particles using point sprites. Technically, a *point sprite* is a vertex. That's right; just a normal, ordinary vertex, like the kind you pump into vertex buffers. However, Direct3D won't render point sprites as ordinary vertices. Instead, it renders a texture in place of every vertex. In other words, you make a texture for your particles and Direct3D will substitute point sprite vertices with your texture. You see what's happening here? The vertices no longer represent corners of a triangle; instead, each vertex represents the center of a particle. And that's how particle systems work. Simple! Using this mechanism you can make snow, rain, magic effects, and more. So let's see how point sprites

themselves work, and then we'll examine how point sprites work in the context of particle systems.

Point sprites are created in Direct3D much like normal vertices. You create a vertex structure, define an FVF, and copy the vertices into a vertex buffer, then render them to the display during the application's rendering loop. Figure 10.1 shows an example of point sprites. The main differences between rendering vertices and rendering point sprites are as follows:

- The vertex structure itself will be slightly different since we'll be holding more than just positional data.
- Because the vertex structure is different, our FVF will also be different.
- Finally, when rendering the point sprites, we'll need to set a few specific render states using the `SetRenderState` method. This method was used earlier in this book.

Creating Point Sprites

As explained previously, the vertex structure and FVF for point sprites are different from standard vertices. A standard vertex might look like the following. You'll notice the structure contains members for position and texture coordinates, and the FVF descriptor reflects this.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;           // The position for the vertex
    DWORD color;             // The vertex color
    FLOAT tu, tv;            // The texture coordinates
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1)
```

For point sprites, however, the vertex structure changes. Point sprites don't need texture coordinates since the vertices don't represent the corners of a triangle. Point sprites require position, size,

and color. The structure to hold this and the corresponding FVF appears below.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z-
    DWORD PSIZE;
    DWORD Diffuse;
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_PSIZE | D3DFVF_DIFFUSE)
```

Once you've defined a new vertex structure and FVF, you can just load them into the vertex buffer like any normal vertices. You saw how to do this in Chapter 4. The code to create some point sprites and load them into a vertex buffer appears below.

```
CUSTOMVERTEX vertices[] =
{
    {150.0f, 50.0f, 0.5f, 10, 0xffff0000},
    {250.0f, 250.0f, 10.5f, 10, 0xff00ff00},
    {50.0f, 250.0f, 50.5f, 10, 0xff00ffff},
    {73.0f, 150.0f, 10.5f, 10, 0xff00ffff},
};

g_p3dDevice->CreateVertexBuffer(4*sizeof(CUSTOMVERTEX), 0,
    D3DFVF_CUSTOMVERTEX, D3DPPOOL_DEFAULT, &g_pVB, NULL);

VOID* pVertices;
if(FAILED(g_pVB->Lock(0, sizeof(vertices), (void**)&pVertices, 0)))
    return E_FAIL;
memcpy(pVertices, vertices, sizeof(vertices));
g_pVB->Unlock();
```

*** NOTE.** You may be wondering how Direct3D knows how to texture your point sprites since no texture coordinates are included in the vertex structure. Simply put, Direct3D takes the position of your vertex and uses this as the center point for a quad (rectangle). Then it uses the point size as a distance to calculate the position of each edge of the rectangle. Finally, it assigns its own texture coordinates to each of those edges to ensure the texture is mapped correctly across the quad, from left to right.

Rendering Point Sprites

Rendering point sprites is almost the same as rendering ordinary vertices except you'll need to set a few additional render states. Apart from that, everything else is the same; you still perform your drawing in the rendering loop and you still use `SetTexture` and `DrawPrimitive`. The first thing you need to do is use `SetRenderState` to set the render states that prepare DirectX for drawing point sprites. The code to do this appears below, then each parameter is explained.

```
g_pd3dDevice->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
g_pd3dDevice->SetRenderState(D3DRS_POINTSIZE_MIN, *((DWORD*)&MinSize));
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALE_A, *((DWORD*)&Scale));
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALE_B, *((DWORD*)&Scale));
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALE_C, *((DWORD*)&Scale));
```

D3DRS_POINTSPRITEENABLE

Typically this will be `True`. When `True`, DirectX will automatically map the texture across the point sprite. If `False`, DirectX will use the texture coordinates for each vertex.

D3DRS_POINTSCALEENABLE

If this value is `True`, DirectX positions and scales point sprites in camera space. If this value is `False`, DirectX will position and scale point sprites in screen space.

D3DRS_POINTSIZE_MIN

Defines the minimum size of a point sprite.

D3DRS_POINTSCALE_A

D3DRS_POINTSCALE_B

D3DRS_POINTSCALE_C

If `D3DRS_POINTSCALEENABLE` is `True`, this value determines how point sprites are scaled. See the SDK documentation for more information.

The rest of the render procedure you'll already recognize. It looks like the following:

```
g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(CUSTOMVERTEX));
g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
g_pd3dDevice->SetTexture(0, g_Texture);
g_pd3dDevice->DrawPrimitive(D3DPT_POINTLIST, 0, 4);
```

*** NOTE.** Unfortunately, not all graphics cards support point sprites, meaning that on some systems nothing will appear or an error may occur when running a point sprite application.

Particle Systems

Now that you've seen how to create and render point sprites in DirectX we can consider them specifically for particle systems. Remember, particle systems will use point sprites to create the illusion of rain, fog, sparks, and more, depending on which textures your application is using.

So let's examine particle systems more closely by studying the following facts about them:

- A particle system is a collection of particles — like raindrops or sparks.
- A particle system is responsible for emitting and destroying particles, and is also responsible for controlling them.

Each particle in a particle system will correspond to a point sprite vertex in the vertex buffer. Typically each particle will have the following properties, if not more:

■ Position

A particle will have a physical position in 2D/3D space. In terms of point sprites, this position directly corresponds to the position of a point sprite vertex, such as (0,5,0).

■ Duration

Most particles do not last forever. They are created and destroyed, often within seconds. For example, in a rain particle

system, the drops begin in the air and gradually fall to the ground under the effects of gravity. Then, as the drops hit the ground they disperse. So, the particles are emitted in the sky and destroyed as they hit the ground. Typically this process will repeat itself to make the rain appear constant. In other words, particles will continually be emitted and destroyed.

■ **Trajectory**

The direction a particle will be moving. For example, rain falls from top to bottom and sparks might spew this way and that. Regardless, each particle will usually be following some predetermined trajectory.

■ **Size**

Finally, a particle will have a size. This is rather self explanatory really. Simply put, particles can be different sizes.

Before moving on to create a sample particle system, let's consider a few properties and behaviors specific to particle systems. In addition to being a collection of particles, a particle system must emit and destroy the particles it contains. As it does so, it must also control the particles (move them about) according to a specific behavior and under the influence of various factors. Let's take the rain example again. Here, the particle system must emit a specified number of raindrops in the air and, under the influence of gravity, must draw the raindrops closer to the ground at a specified speed and then destroy the particles as they reach the ground. To summarize, then, a particle system has two primary roles:

■ **Particle generation and destruction**

A particle system creates particles either randomly or according to a specific generation rate, and then destroys particles in a similar way, usually when a condition is met; for example, when the rain hits the ground.

■ **Particle behavior**

Particle systems also control the behavior of a particle. Specifically this means moving the particle around. In some cases the particle might move in any random direction; however, to

be more realistic, the particles will probably react to a variety of forces like gravity or wind.

Creating a Particle System

Using our knowledge of point sprites and our theoretical understanding of particle systems, we can now create a real particle system for ourselves. This involves creating two classes: one for the particle system itself, which will be called `CParticleSystem`, and one for a single particle, which will be named `CParticle`. The main class will be `CParticleSystem` and will contain a collection of `CParticle` instances. Furthermore, `CParticleSystem` is responsible for emitting, destroying, and controlling particles.

The class declaration for `CParticle` appears below, and a brief explanation follows.

```
class CParticle
{
    private:
    protected:
    public:
        D3DXVECTOR3 m_Position;           //Position of the particle
        D3DXVECTOR3 m_Trajectory;        //Normalized trajectory of the particle
        DWORD m_Size;                   //Particle size
        DWORD m_Color;                   //Particle color
        DWORD m_Duration;                //Particle duration after emission
        DWORD m_Age;                     //How long the particle lasts
        DWORD m_StartTime;               //Time the particle was created
        DWORD m_Speed;

        CParticle()
        {
            m_Position = m_Trajectory = D3DXVECTOR3(0,0,0);
            m_Color = D3DCOLOR_XRGB(0,0,0);
            m_Size = m_Duration = m_Age = m_Speed = m_StartTime = 0;
        }
}
```

```

VOID Update(DWORD Time)
{
    m_Age = Time - m_StartTime;
    m_Position += m_Trajectory * m_Speed;
}
};

```

As you can see, not much is actually going on in this small class. It represents a single particle. It has a position, an age, a speed, and a trajectory. Nearly all of its values are set to 0 in the constructor. The Update function updates the time, recalculates the particle's age, and then updates the particle's trajectory by a specific speed. The Update function will be called by the CParticleSystem class, as shown in a moment. Notice that the particle class doesn't draw itself either; it simply updates its position. The owning CParticleSystem class will be responsible for maintaining the particle texture and drawing the particles themselves. Take a look at the particle system class below.

```

class CParticleSystem
{
private:
protected:
public:
    CParticle *m_Particles;
    DWORD m_NumParticles;
    LPDIRECT3DVERTEXBUFFER9 m_pVB;
    LPDIRECT3DTEXTURE9 m_Texture;

    CParticleSystem(DWORD NumParticles)
    {
        m_NumParticles = NumParticles;
        m_Particles = new CParticle[m_NumParticles];
        m_pVB = NULL;
        m_Texture = NULL;

        g_pd3dDevice->CreateVertexBuffer(m_NumParticles*sizeof(CUSTOMVERTEX),
            0, D3DFVF_CUSTOMVERTEX, D3DPPOOL_DEFAULT, &m_pVB, NULL);

        D3DXCreateTextureFromFile(g_pd3dDevice, "Sprite.bmp", &m_Texture);
    }
};

```

```

}

~CParticleSystem()
{
    if(m_pVB)
        m_pVB->Release();

    if(m_Texture)
        m_Texture->Release();

    if(m_Particles)
        delete [] m_Particles;
}

VOID Update()
{
    DWORD Time = timeGetTime();
    CUSTOMVERTEX *vertices = new CUSTOMVERTEX[m_NumParticles];

    for(DWORD Counter = 0; Counter < m_NumParticles; Counter++)
    {
        if(m_Particles[Counter].m_Age >= m_Particles[Counter].m_Duration)
        {
            m_Particles[Counter].m_StartTime = timeGetTime();
            m_Particles[Counter].m_Position = D3DXVECTOR3(0, 100, 0);
            m_Particles[Counter].m_Trajectory = D3DXVECTOR3(0, -1, 0);
            m_Particles[Counter].m_Speed = 5;
            m_Particles[Counter].m_Duration = 1000;
        }

        m_Particles[Counter].Update(Time);

        vertices[Counter].x = m_Particles[Counter].m_Position.x;
        vertices[Counter].y = m_Particles[Counter].m_Position.y;
        vertices[Counter].z = m_Particles[Counter].m_Position.z;
    }

    VOID* pVertices = NULL;
    m_pVB->Lock(0, sizeof(CUSTOMVERTEX) * m_NumParticles,
        (void**)&pVertices, 0);

```

```

memcpy(pVertices, vertices, sizeof(vertices));
m_pVB->Unlock();

g_pd3dDevice->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
g_pd3dDevice->SetRenderState(D3DRS_POINTSIZE_MIN,
                             *((DWORD*)&MinSize));
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALE_A, *((DWORD*)&Scale));
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALE_B, *((DWORD*)&Scale));
g_pd3dDevice->SetRenderState(D3DRS_POINTSCALE_C, *((DWORD*)&Scale));

g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(CUSTOMVERTEX));
g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);
g_pd3dDevice->SetTexture(0, m_Texture);
g_pd3dDevice->DrawPrimitive(D3DPT_POINTLIST, 0, m_NumParticles);
}
};

```

That's it; that's all the code there is. The particle system in this example simply moves particles from top to bottom like raindrops. After examining the above code you'll notice several key features about the particle system:

- It keeps a dynamically sized array of particle classes. These represent the particles of the particle system.
- It maintains a vertex buffer, which is created in the constructor and is continually repopulated in the Update function with an array of updated particles.
- CParticleSystem also holds the texture to be used for each particle (point sprite).

As CParticleSystem is created the constructor is called. Here, the class creates a vertex buffer and a texture from a file, and then finally generates a batch of particles. All that an application needs to do then is continually call the Update method during the render loop. As the Update method is called, CParticleSystem is responsible for moving particles from top to bottom. During this method, CParticleSystem checks to see whether the lifetime of any particle has expired. If it has, then the particle will have reached the

bottom, so `CParticleSystem` resets the particle by moving it back to the top and resetting its age and position. If, on the other hand, the particle has not expired, the `Update` function moves it farther along its trajectory. Finally, the `Update` method loads the particles into the vertex buffer and renders them to the display where they're seen as point sprites. It's really that simple.

Conclusion

This chapter was comparatively short and focused on a very specific feature of Direct3D: point sprites and their practical use in particle systems. Of course, particle systems go beyond simply rendering point sprites that move from top to bottom. They're also used in particle systems that simulate behavior far more complex than this. In real-life particle systems, developers tend to use real physics to simulate true-life forces like wind, gravity, and inertia. I used a simplified system for the purposes of this book to concentrate on the Direct3D-related implementation rather than the mathematical concepts for coding forces. For a more involved examination on exactly how to implement natural forces for simulating physics, I recommend picking up a high school- or university-level mathematics textbook.



Chapter 11

Playing Video and Animating Textures

This chapter discusses two different subjects, both related to animations. First, it examines how video is played using DirectX and how sound and other kinds of streamable media are played. Second, this chapter examines how pixel data on textures can be animated.

Playing Video Using DirectShow

Part of DirectX, DirectShow is an API used for playing streamable media like movies and music, among other things. Using DirectShow it's possible to play movie files like MPGs, AVIs, and WMVs and sounds like WAVs, MP3s, and WMAs. Ultimately, DirectShow works by using a COM interface called a *filter graph*. This component can be thought of as a giant chart made up of various blocks all connected to one another by pins or wires. This graph represents how the bytes are read in from a media file, like an MP3, and then processed through the graph into something playable, such as sound. In other words, the filter graph is the engine by which media data is taken from the file and then played on the monitor or speakers. So let's look at how to play a file using DirectShow.

To begin working with DirectShow you'll need to include some extra libraries and include files in your project. These are listed below.

Includes

Dshow.h

Libs

Strmiids.lib

Quartz.lib

The 1, 2, 3 of Playing a File

Once you've linked to your libraries and included your headers, the next step in playing a media file using DirectShow is to build a filter graph using the various graph building functions DirectX exposes. That's right; we need to build a filter graph to play a file. Don't worry, though, it's really simple to do. The following sections show you how.

*** NOTE.** Remember to call the COM function `Colnitialize()` before using DirectShow, and `CoUninitialize()` after using DirectShow.

Creating the Filter Graph

The first step in the graph building process is to create a filter graph using the graph builder interface, `IGraphBuilder`. This interface provides a variety of methods to construct a filter graph that plays media files. To create a graph builder interface you simply call the standard COM creating method `CoCreateInstance`. The following code demonstrates how to create a graph builder.

```
IGraphBuilder *pGraph = NULL;  
HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,  
CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
```

Media Control and Event System

At this stage you've created a blank filter graph and now it's time to prepare it for playing a media file. The first stage of this involves obtaining two other interfaces that are part of the filter graph: the media control interface (IMediaControl) and the media event interface (IMediaEventEx). The media control can be thought of as a standard media player and is responsible for starting and stopping playback on the filter graph. When it comes to playing a media file, we'll use the media control to start playback. The other interface, the media event, is responsible for notifying an application when events in the filter graph occur, such as when playback has completed or when an error occurs. To obtain interface pointers to these two interfaces you'll need to call the standard COM method `QueryInterface` on the `IGraphBuilder` interface. The code below demonstrates how this is done.

```
IMediaControl *pControl = NULL;
IMediaEvent *pEvent = NULL;
hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
```

Loading a Media File

Now that you've created a filter graph and obtained the media control and event interfaces, you can start to assemble the graph for playing a file. There are many ways to construct a filter graph, some simple and some complex. If you just want to play a media file, then building a filter graph is really simple. In fact, `DirectShow` will do it automatically for you in one call. This method is called **RenderFile**, and it accepts a filename to load. Once called successfully, `DirectShow` will have constructed a filter graph to play the file, ready to start. The syntax and parameters for `RenderFile` follow, along with some code that demonstrates how to call this function.


```
HRESULT RenderFile(
    LPCWSTR lpwstrFile,
    LPCWSTR lpwstrPlayList
);
```

LPCWSTR lpwstrFile

Filename of the media to load.

LPCWSTR lpwstrPlayList

This is a reserved value and must be set to NULL.

Sample code:

```
hr = pGraph->RenderFile(L"test.mpg", NULL);
```

Configuring Events

The previous sections have prepared the file for playing on the filter graph. Before we use the media control to start playback, however, we'll configure our event interface to ensure the application receives events as playback elapses. To do this, we must define a custom window message. This message will be sent to our application's window whenever the event interface needs to be polled for filter graph events. A custom window message can be defined in our source like this:

```
#define WM_GRAPHNOTIFY WM_APP + 1
```

Once a custom message has been defined by your app, like the one above, you must then register this message and your application's window handle with the event interface using the **SetNotifyWindow** method. The syntax and parameters for this method appear below. Then some code shows how message registration is performed.

```
HRESULT SetNotifyWindow
(
    OAHWND hwnd,
    long lMsg,
    LONG_PTR lInstanceData
);
```

OAHWND *hwnd*

Handle of the window where the custom message is to be sent when events occur in the filter graph.

long *lMsg*

Custom message to be sent when events occur. This will be your custom defined message.

LONG_PTR *lInstanceData*

This is an additional data parameter that can be passed with your message. This can also be set to NULL.

Sample code:

```
g_pEvent->SetNotifyWindow((OAHWND)g_hwnd, WM_GRAPHNOTIFY, 0);
```

Once you've registered the window handle and associated message to be sent when events occur in the filter graph, you'll also need to amend your WndProc procedure. Remember, WndProc is where application messages are sent. You'll want to add an extra switch case statement to handle your custom message, like this:

```
LRESULT WINAPI MwndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
            Cleanup();
            PostQuitMessage(0);
            return 0;

        case WM_GRAPHNOTIFY:
            HandleGraphEvent();    //Do stuff here
            return 0;
    }
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Once an application has received a custom message from the filter graph, it means one or more events have occurred during media playback. To process these events you need to call the **GetEvent** method of **IMediaEventEx**. To process all events (because there could be more than one event queued) you simply call this method repeatedly until it fails. The function fails when there are no more events to process. The syntax and parameters for **GetEvent** are listed below, and the code that follows demonstrates how to use this function.

```
HRESULT GetEvent(
    long *lEventCode,
    LONG_PTR *lParam1,
    LONG_PTR *lParam2,
    long msTimeout
);
```

long *lEventCode

Address to receive the event code. This code describes the event that occurred. This can be any of these values:

EC_ACTIVATE

A video window is being activated or deactivated.

EC_BUFFERING_DATA

The filter graph is buffering or has stopped buffering data.

EC_COMPLETE

All data from the stream has been rendered. In other words, the video or sound has finished playing.

EC_DEVICE_LOST

The device was lost.

EC_DISPLAY_CHANGED

The display mode has changed.

EC_ERRORABORT

The filter graph has aborted the current operation. Simply put, an error occurred.

EC_GRAPH_CHANGED

The filter graph has changed.

EC_LENGTH_CHANGED

The length of the media has changed.

EC_OPENING_FILE

The graph is opening a file or has finished opening a file.

EC_PAUSED

The graph has been paused.

EC_USERABORT

The user has terminated playback.

EC_VIDEO_SIZE_CHANGED

The video size has changed.

LONG_PTR *lParam1

Address to receive the first event parameter.

LONG_PTR *lParam2

Address to receive the second event parameter.

long msTimeout

Timeout interval in milliseconds to wait, or use INFINITE to wait until an event occurs.

Sample code:

```
// Get all the events
long evCode;
LONG_PTR param1, param2;
HRESULT hr;
while (SUCCEEDED(g_pEvent->GetEvent(&evCode, &param1, &param2, 0)))
{
    g_pEvent->FreeEventParams(evCode, param1, param2);
    switch (evCode)
    {
        case EC_COMPLETE:    // Fall through
        case EC_USERABORT:   // Fall through
        case EC_ERRORABORT:
            RunMyFunction(); // Do stuff here
    }
}
```

```

        return;
    }
}

```

Playing a File

Once you've loaded a filter graph from a file and configured an event mechanism, you can start using the media control to play the media. To do this you call the `Run` method of `IMediaControl`. The code for building and running the filter graph appears below.

```

IGraphBuilder *pGraph = NULL;
IMediaControl *pControl = NULL;
IMediaEventEx *pEvent = NULL;

//Create filter graph
HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,
                             CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);

//Create media control and events
hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
hr = pGraph->QueryInterface(IID_IMediaEventEx, (void **)&pEvent);

//Load a file
hr = pGraph->RenderFile(L"test.mpg", NULL);

//Set window for events
pEvent->SetNotifyWindow((OAHWND)g_hwnd, WM_GRAPHNOTIFY, 0);

//Play media control
pControl->Run();

```

Playing Video — Further Information

Now that you can play a video you may wonder how you can play this video on the surfaces of textures. In other words, you want to copy the video pixel data onto a texture so you can have animated textures. There are several ways you can do this; however, this topic is beyond the scope of this book. Instead I'll give you a few pointers and places you can look if you're interested.

- First, if you have the video pixel data (either by using GDI or DirectX) and you simply want to copy the pixels onto a valid Direct3D surface, then the `IDirect3DSurface9` interface provides a really convenient way of doing this. To retrieve a valid device context associated with a Direct3D surface, you can call the `GetDC` method of `IDirect3DSurface9`. Then you can copy over your pixel data. And finally, once you're finished, you must be sure to call the `ReleaseDC` method. Here's an example:

```
Surface->GetDC(&hdc);
SetDIBitsToDevice(hdc, rect.left, rect.top, rect.right - rect.left + 1,
                  rect.bottom - rect.top + 1, 0, 0, 0, (UINT) 1pbmi ->
                  bmiHeader.biHeight, 1pvbits, 1pbmi, DIB_RGB_COLORS);
Surface->ReleaseDC(hdc);
```

- The other method of copying video data onto a surface involves creating a custom DirectShow filter that plugs into the filter graph and copies video pixel data as it's streamed through the pipeline. Then, as it encounters video data, it transfers it onto a texture. A DirectShow sample application that accompanies the DirectX SDK can be found in the `Samples\DirectShow\Players\Texture3D` folder. This sample demonstrates how to render video data from a filter graph onto a Direct3D texture.

Animating Textures

In addition to playing video and playing video on textures, you can also transform the pixel data on textures by using matrices. That's right; you can actually transform the pixels on a texture by a matrix, just like you'd transform ordinary geometry by using a transformation matrix. This process is called texture transformation, and it's extremely simple to do. For example, if you wanted to create a scrolling sky or rushing stream of water, you could create a tileable texture that you could scroll repeatedly along by using a transformation matrix.

To begin, you simply create a standard Direct3D texture and load an image onto it. Then you can use the texture as you would normally, by either applying it to some geometry or by rendering the texture individually using the `ID3DXSprite` interface.

Next, you need to build a transformation matrix to transform the texture. You can do this using the standard matrix building functions. The only thing you need to remember is that textures are flat surfaces, not 3D, so your transformation matrix needs to be a 3x3 (2D) matrix. In other words, your transformation can manipulate the texture along the X and Y axes but not on the Z axis. Earlier in this book, when examining textures in conjunction with the `ID3DXSprite` interface, you were introduced to a way to build 2D matrices. DirectX provides a function to do this called `D3DXMatrixTransformation2D`. To quickly recap, its syntax is featured below. (For a more thorough explanation, please refer to the section in Chapter 5 called “`ID3DXSprite` — Drawing Textures in 2D.”)

```
D3DXMATRIX *WINAPI D3DXMatrixTransformation2D
(
    D3DXMATRIX *pOut,
    CONST D3DXVECTOR2 *pScalingCenter,
    FLOAT *pScalingRotation,
    CONST D3DXVECTOR2 *pScaling,
```

```

CONST D3DXVECTOR2 *pRotationCenter,
FLOAT Rotation,
CONST D3DXVECTOR2 *pTranslation
);

```

Once a transformation matrix has been constructed, you can proceed to apply it to a texture by performing the following steps:

1. Call the `SetTransform` method of `IDirect3DDevice9`, passing `D3DTS_TEXTURE0` as the first parameter and then your matrix as the second. This will tell Direct3D that you're transforming a texture. Some sample code to do this appears below.

```
pDevice->SetTransform(D3DTS_TEXTURE0, &Transform);
```

2. Call the `SetTextureStageState` function to inform Direct3D that you'll be using a 2D matrix to transform the texture. Here's some code to do this:

```

pDevice->SetTextureStageState(
    0, D3DTSS_TEXTURETRANSFORMFLAGS,
    D3DTTFF_COUNT2
);

```

3. Once the texture has been rendered and all texture transformations are completed, you can disable texture transformation by calling `SetTextureStageState` again. This time you'll pass different parameters. Take a look at the code below to see how this is done.

```

pDevice->SetTextureStageState(
    0, D3DTSS_TEXTURETRANSFORMFLAGS,
    D3DTTFF_DISABLE
);

```


Conclusion

You've reached the end of this chapter and this concludes video playing and texture animation. Now, as you approach the next chapter in this book, you should prepare yourself for some comparatively taxing challenges as we begin to examine further texture work, specifically video textures.



Chapter 12

More Animated Texturing

This chapter expands upon the work of the previous chapter and examines textures that can play video, that is, textures that are not simply static images like JPEGs or bitmaps, but textures that are loaded from sources like AVIs or MPGs. In other words, this chapter examines animated textures that can be loaded from video files. Specifically, the following points are studied:

- The DirectShow base classes
- The filter graph plug-in framework
- Custom renderers
- Surface copying
- Playing videos as textures



TIP. The work covered in this chapter relies upon subject matter covered in the previous chapter. Specifically, you should already know how to create a filter graph and play media files.

Movie Files (MPG, AVI, and More)

OK, so far this book has taught you how to load static textures onto a polygon and how to animate those textures in terms of texture transformation. Using texture transformation you can move a static image — like an image of water — and scroll or move it about the surface of a polygon to simulate movement using matrices. However, thus far you have not seen how actual video (moving images) can be loaded onto a texture. When I say video, I mean movie files like AVIs and MPGs. These can be captured from standard video cameras or produced using rendering software like 3ds max.

***NOTE.** Although movie files can contain a plethora of additional data like compression information, etc., you will find that throughout this chapter it will be useful to think of movie files as simply an array of bitmaps that are cycled through as time elapses.

Playing Video on Textures in Theory

As I mentioned, in this chapter we'll think about a movie file as simply an array of static images. These images are cycled through, one by one, as time elapses. Under this assumption, the theory behind playing movie files on textures is painfully simple. It goes roughly as follows:

1. Open the movie file using the DirectShow filter graph and read in the video dimensions (the width and height of the video). According to these dimensions you should then create a texture of sufficient size to hold these dimensions. It is therefore important at this stage to note that your videos should ideally be sized to a power of two (256x256, 512x512, etc.). This is because, as mentioned earlier, most cards only support textures of this size, and textures of any other size will be stretched or shrunk to the nearest valid size.
2. Once the texture has been created, you can set the video file playing using the filter graph.

3. As playback elapses, you access the video's pixels for the current frame (the actual image data of the bitmap currently being shown) and then copy them over onto the texture. Because the pixel data can be stored in various formats, you'll need to copy over those pixels manually too.

Playing Video on Textures in Practice

As you can see, the theory behind playing video on textures is really simple. Unfortunately, the reality behind achieving this isn't. In order to achieve this practically, you must implement a DirectShow filter graph plug-in called a custom renderer. Basically, this means you need to create a class that plugs into and interacts with the filter graph as it plays back video. By doing this, the filter graph can notify you, through the methods of your class, of streaming events as playback occurs and can provide you with access to the actual pixel data of the current video frame. The class that you'll need to implement should be derived from a class (included with the DirectX SDK) called `CBaseVideoRenderer`. This class, among many others, is part of the DirectShow base classes, which are used with the filter graph in different ways. For the purposes of this chapter, only `CBaseVideoRenderer` needs to be used.

In order to use this base class, or any other, you'll need to include `streams.h` in your projects. This header file can be found in the DirectX SDK folder at: `\Samples\C++\DirectShow\BaseClasses`. However, in addition to the `streams.h` file, you'll also need to include a complementary `.lib` file (`strmbase.lib` or `strmbasd.lib`). This `.lib` file is not included prebuilt with the DirectX SDK. Instead, you'll need to build the `.lib` file yourself by compiling the `BaseClasses` project included in the same folder as `streams.h`.

Take a look at Figure 12.1 to see how `CBaseVideoRenderer` fits into the DirectShow base class framework. You don't actually need to know this hierarchy intimately, but it's important to understand that `CBaseVideoRenderer` inherits functionality from other base classes.

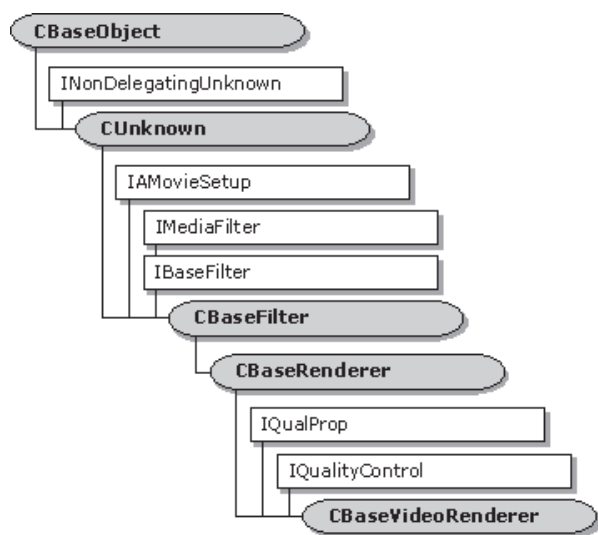


Figure 12.1: CBaseVideoRenderer hierarchy

Creating the Base Video Renderer

As mentioned previously, the base video renderer class is a point from which developers should derive their own class to plug into the filter graph. This class will be used to stream in the pixel data as video is played through the filter graph and then to copy it onto a texture on each frame. For the purposes of this chapter, this derived class will be named CTextureRenderer, and I have generated a GUID to represent this class. You saw how to generate GUIDs earlier in this book using GUIDGEN. My GUID looks like this:

```
struct __declspec(uuid("{71771540-2017-11cf-ae26-0020afd79767}"))
    CLSID_TextureRenderer;
```

Let's take a look at the class declaration for CTextureRenderer that appears below. Don't worry if you don't understand it all yet. The class is explained as the chapter progresses.

```
class CTextureRenderer : public CBaseVideoRenderer
{
    public:

    BOOL m_bUseDynamicTextures;
    LONG m_lVidWidth;    // Video width
    LONG m_lVidHeight;   // Video height
    LONG m_lVidPitch;    // Video pitch
    LPDIRECT3DTEXTURE9 m_pTexture;
    D3DFORMAT m_TextureFormat;

    CTextureRenderer::CTextureRenderer(LPUNKNOWN pUnk, HRESULT *phr);
    ~CTextureRenderer();
    HRESULT CheckMediaType(const CMediaType *pmt);
    HRESULT SetMediaType(const CMediaType *pmt);
    HRESULT DoRenderSample(IMediaSample *pSample);
};
```

Implementing the Base Video Renderer

Before continuing, it's important to remember that the CBaseVideoRenderer class will act as a plug-in for the filter graph. This means that as the movie file is played back, some of the methods of your class (such as DoRenderSample) will be used periodically, on each frame, by the filter graph. These methods are overridden from the base class and are implemented and explained in the following sections.

Implementing the Constructor

CTextureRenderer's constructor is where it all begins. It accepts one parameter and must pass on several others to the ancestor class's constructor. All of this is a pretty simple initialization process, as shown below.

```
CTextureRenderer::CTextureRenderer(LPUNKNOWN pUnk, HRESULT *phr)
    : CBaseVideoRenderer(__uuidof(CLSID_TextureRenderer),
        NAME("Texture Renderer"), pUnk, phr),
        m_bUseDynamicTextures(FALSE)
{
    ASSERT(phr);

    if (phr)
        *phr = S_OK;

    m_pTexture = NULL;
    m_bUseDynamicTextures = false;
    m_lVidWidth = m_lVidHeight = m_lVidPitch = 0;
    ZeroMemory(&m_TextureFormat, sizeof(D3DFORMAT));
}
```

Implementing CheckMediaType

The CheckMediaType method is overridden from the CBaseVideoRenderer class and is called by the filter graph as it needs to establish whether the video format is acceptable for our class. Take a look at the code below and then I'll explain it.

```
HRESULT CheckMediaType(const CMediaType *pmt)
{
    HRESULT hr = E_FAIL;
    VIDEOINFO *pvi=0;

    CheckPointer(pmt, E_POINTER);
```

```

// Reject the connection if this is not a video type
if(*pmt->FormatType() != FORMAT_VideoInfo) {
    return E_INVALIDARG;
}

// Only accept RGB24 video
pvi = (VIDEOINFO *)pmt->Format();

if(IsEqualGUID(*pmt->Type(), MEDIATYPE_Video) &&
IsEqualGUID(*pmt->Subtype(), MEDIASUBTYPE_RGB24))
{
    hr = S_OK;
}

return hr;
}

```

This method accepts as its argument an object of `CMediaType`, which describes the media type being loaded and processed. It is the role of the `CheckMediaType` function to determine whether this media type and format is acceptable and to return success or failure accordingly. If the method fails, then no further processing will occur because the media will have been rejected. If this method returns success, then media processing will continue.

Implementing `SetMediaType`

The `SetMediaType` method for the video renderer takes media information that is accepted by `CheckMediaType` and allocates resources accordingly. In other words, this function accepts an argument that contains information about the video format and video size. Then it creates a texture to hold the video data as it's streamed later on. Take a look at the code below to see how this is done.

```

HRESULT SetMediaType(const CMediaType *pmt) // Video format notification
{
    HRESULT hr;

```



```

UINT uintWidth = 2;
UINT uintHeight = 2;

// Retrieve the size of this media type
D3DCAPS9 caps;
VIDEOINFO *pviBmp; // Bitmap info header
pviBmp = (VIDEOINFO *)pmt->Format();

m_lVidWidth = pviBmp->bmiHeader.biWidth;
m_lVidHeight = abs(pviBmp->bmiHeader.biHeight);
m_lVidPitch = (m_lVidWidth * 3 + 3) & ~(3); // We are forcing RGB24

// Here let's check if we can use dynamic textures
ZeroMemory(&caps, sizeof(D3DCAPS9));

hr = g_pd3dDevice->GetDeviceCaps(&caps);

if(caps.TextureCaps & D3DPTURECAPS_POW2)
{
    while((LONG)uintWidth < m_lVidWidth)
    {
        uintWidth = uintWidth << 1;
    }
    while((LONG)uintHeight < m_lVidHeight)
    {
        uintHeight = uintHeight << 1;
    }
}
else
{
    uintWidth = m_lVidWidth;
    uintHeight = m_lVidHeight;
}

// Create the texture that maps to this media type
hr = E_UNEXPECTED;
if(m_bUseDynamicTextures)
{

```

```

        hr = g_pd3dDevice->CreateTexture(uintWidth, uintHeight, 1,
                                         D3DUSAGE_DYNAMIC, D3DFMT_X8R8G8B8, D3DPPOOL_DEFAULT,
                                         &m_pTexture, NULL);
        g_pachRenderMethod = g_achDynTextr;
        if(FAILED(hr))
        {
            m_bUseDynamicTextures = FALSE;
        }
    }
    if(FALSE == m_bUseDynamicTextures)
    {
        hr = g_pd3dDevice->CreateTexture(uintWidth, uintHeight, 1, 0,
                                         D3DFMT_X8R8G8B8, D3DPPOOL_MANAGED,
                                         &m_pTexture, NULL);
        g_pachRenderMethod = g_achCopy;
    }
    if(FAILED(hr))
        return hr;

    // CreateTexture can silently change the parameters on us
    D3DSURFACE_DESC ddsd;
    ZeroMemory(&ddsd, sizeof(ddsd));

    if (FAILED(hr = m_pTexture->GetLevelDesc(0, &ddsd)))
        return hr;

    LPDIRECT3DSURFACE9 Surface = NULL;

    if (SUCCEEDED(hr = m_pTexture->GetSurfaceLevel(0, &Surface)))
        Surface->GetDesc(&ddsd);

    m_TextureFormat = ddsd.Format;

    if (m_TextureFormat != D3DFMT_X8R8G8B8 && m_TextureFormat !=
        D3DFMT_A1R5G5B5)
        return VFW_E_TYPE_NOT_ACCEPTED;

    return S_OK;
}

```

Implementing DoRenderSample

DoRenderSample is a method that occurs on every frame, as the video is played back. This is tricky part, and it's here that you'll need to access the video's pixel data and copy it over to a texture. The code appears as follows and then I'll explain it.

```
HRESULT DoRenderSample(IMediaSample *pSample) // New video sample
{
    BYTE *pBmpBuffer, *pTxtBuffer; // Bitmap buffer, texture buffer
    LONG lTxtPitch;                // Pitch of bitmap, texture

    BYTE * pbS = NULL;
    DWORD * pdwS = NULL;
    DWORD * pdwD = NULL;
    UINT row, col, dwordWidth;

    CheckPointer(pSample, E_POINTER);
    CheckPointer(m_pTexture, E_UNEXPECTED);

    // Get the video bitmap buffer
    pSample->GetPointer(&pBmpBuffer);

    // Lock the texture
    D3DLOCKED_RECT d3dlr;
    if(m_bUseDynamicTextures)
    {
        if(FAILED(m_pTexture->LockRect(0, &d3dlr, 0, D3DLOCK_DISCARD)))
            return E_FAIL;
    }
    else
    {
        if (FAILED(m_pTexture->LockRect(0, &d3dlr, 0, 0)))
            return E_FAIL;
    }

    // Get the texture buffer & pitch
    pTxtBuffer = static_cast<byte *>(d3dlr.pBits);
    lTxtPitch = d3dlr.Pitch;
```

```

// Copy the bits

if (m_TextureFormat == D3DFMT_X8R8G8B8)
{
    dwordWidth = m_lVidWidth / 4;

    for(row = 0; row< (UINT)m_lVidHeight; row++)
    {
        pdwS = (DWORD*)pBmpBuffer;
        pdwD = (DWORD*)pTxtBuffer;

        for(col = 0; col < dwordWidth; col ++)
        {
            pdwD[0] = pdwS[0] | 0xFF000000;
            pdwD[1] = ((pdwS[1]<<8) | 0xFF000000) | (pdwS[0]>>24);
            pdwD[2] = ((pdwS[2]<<16) | 0xFF000000) | (pdwS[1]>>16);
            pdwD[3] = 0xFF000000 | (pdwS[2]>>8);
            pdwD +=4;
            pdwS +=3;
        }

        // We might have remaining (misaligned) bytes here
        pbS = (BYTE*) pdwS;
        for(col = 0; col < (UINT)m_lVidWidth % 4; col++)
        {
            *pdwD = 0xFF000000 |
                (pbS[2] << 16) |
                (pbS[1] << 8) |
                (pbS[0]);
            pdwD++;
            pbS += 3;
        }

        pBmpBuffer += m_lVidPitch;
        pTxtBuffer += lTxtPitch;
    }
    // for rows
}

if (m_TextureFormat == D3DFMT_A1R5G5B5)
{

```

```

    for(int y = 0; y < m_lVidHeight; y++)
    {
        BYTE *pBmpBufferOld = pBmpBuffer;
        BYTE *pTxtBufferOld = pTxtBuffer;

        for (int x = 0; x < m_lVidWidth; x++)
        {
            *(WORD *)pTxtBuffer = (WORD)
                (0x8000 +
                 ((pBmpBuffer[2] & 0xF8) << 7) +
                 ((pBmpBuffer[1] & 0xF8) << 2) +
                 (pBmpBuffer[0] >> 3));

            pTxtBuffer += 2;
            pBmpBuffer += 3;
        }

        pBmpBuffer = pBmpBufferOld + m_lVidPitch;
        pTxtBuffer = pTxtBufferOld + lTxtPitch;
    }
}

// Unlock the texture
if (FAILED(m_pTexture->UnlockRect(0)))
    return E_FAIL;

return S_OK;
}

```

Hmmm, this function looks quite nasty, doesn't it? Actually, go over it a few times and you'll see it's not as bad as it first seems. The first important part is the locking of the texture. This is achieved by calling the **LockRect** method of **IDirect3DTexture9**. Calling **LockRect** on a texture is tantamount to calling **Lock** on a **Direct3D** surface; it accesses the raw pixel data of the texture. Nothing is actually copied yet; it simply prepares the texture for copying pixels onto it. The syntax and parameters for **LockRect** appear below.

```

HRESULT LockRect(
    UINT Level,

```

```

D3DLOCKED_RECT *pLockedRect,
CONST RECT *pRect,
DWORD Flags
);

```

UINT Level

Specifies the level of the texture to lock. For the purposes of this book, you can pass 0.

D3DLOCKED_RECT *pLockedRect

Pointer to a D3DLOCKED_RECT structure to receive information describing the locked region.

CONST RECT *pRect

Pointer to a rectangle to lock, or NULL to lock the entire texture.

DWORD Flags

Can be none or a combination of any of the following flags, indicating the type of lock to perform:

```

D3DLOCK_DISCARD
D3DLOCK_NO_DIRTY_UPDATE
D3DLOCK_NOSYSLOCK
D3DLOCK_READONLY

```

Once the texture has been locked it's ready to receive pixel data. So, the DoRenderSample function moves on. Afterward, the function casts the bits of the texture and then checks for the format of the texture. The format will influence exactly how those raw bytes will be copied in the next process, so this is why it needs to be checked beforehand; otherwise, the image will look all messed up. Here's the isolated texture copying code for your convenience:

```

if (m_TextureFormat == D3DFMT_X8R8G8B8)
{
    dwordWidth = m_lVidWidth / 4;

    for(row = 0; row< (UINT)m_lVidHeight; row++)
    {

```

```

    pdwS = (DWORD*)pBmpBuffer;
    pdwD = (DWORD*)pTxtBuffer;

    for(col = 0; col < dwordWidth; col ++)
    {
        pdwD[0] = pdwS[0] | 0xFF000000;
        pdwD[1] = ((pdwS[1]<<8) | 0xFF000000) | (pdwS[0]>>24);
        pdwD[2] = ((pdwS[2]<<16) | 0xFF000000) | (pdwS[1]>>16);
        pdwD[3] = 0xFF000000 | (pdwS[2]>>8);
        pdwD +=4;
        pdwS +=3;
    }

    // We might have remaining (misaligned) bytes here
    pbS = (BYTE*) pdwS;
    for(col = 0; col < (UINT)m_lVidWidth % 4; col++)
    {
        *pdwD = 0xFF000000 |
            (pbS[2] << 16) |
            (pbS[1] << 8) |
            (pbS[0]);
        pdwD++;
        pbS += 3;
    }

    pBmpBuffer += m_lVidPitch;
    pTxtBuffer += lTxtPitch;
} // for rows
}

if (m_TextureFormat == D3DFMT_A1R5G5B5)
{
    for(int y = 0; y < m_lVidHeight; y++)
    {
        BYTE *pBmpBufferOld = pBmpBuffer;
        BYTE *pTxtBufferOld = pTxtBuffer;

        for (int x = 0; x < m_lVidWidth; x++)
        {
            *(WORD *)pTxtBuffer = (WORD)

```

```

        (0x8000 +
        ((pBmpBuffer[2] & 0xF8) << 7) +
        ((pBmpBuffer[1] & 0xF8) << 2) +
        (pBmpBuffer[0] >> 3));

    pTxtBuffer += 2;
    pBmpBuffer += 3;
}

pBmpBuffer = pBmpBufferOld + m_lVidPitch;
pTxtBuffer = pTxtBufferOld + lTxtPitch;
}
}

```

And that's it! That's how a video is copied onto a texture. However, this plug-in class is no good on its own. It needs a filter graph to load the video file, load the plug-in, and then kick the movie into action. The next section demonstrates how to create a filter graph.

Preparing the Filter Graph

Before our custom renderer can actually render video, it needs to be plugged into a filter graph. In other words, the custom render class relies on the filter graph to receive video data. In the previous chapter it was demonstrated how a filter graph could be created, loaded with streamable media, and set into motion. This process is more or less repeated here with a few exceptions involving our custom renderer. Let's see the individual stages of creating and building a filter graph to use our plug-in renderer.

1. The process begins by creating the filter graph. This is no different from what we're used to. Take a look at the code to do this:

```

HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,
    CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&m_pGraph);

```


2. Next, you should declare an instance of your render class. This is really simple to do, as shown below.

```
m_TextureRenderer = new CTextureRenderer(NULL, &hr);
```

3. After this, you'll want to insert the render class into the filter graph as a valid custom video renderer. From this point onward, the filter graph is able to call upon any of your implemented methods. The filter graph accepts the insertion of plug-ins through the **AddFilter** method. Its syntax and parameters are shown below, and then some sample code follows.

```
HRESULT AddFilter
(
    IBaseFilter *pFilter,
    LPCWSTR pName
);
```

IBaseFilter *pFilter

Pointer to the IBaseFilter interface to add as a plug-in. This is where your video render class is ultimately derived. In other words, this is where you pass your custom render class.

LPCWSTR pName

Pointer to a wide-character string containing a user-defined name for the filter. This can be any name you want.

***NOTE.** It is important to note the possible failure codes for this function. These are as follows:

- S_OK — Success
- VFW_S_DUPLICATE_NAME — Successfully added a filter with a duplicate name
- E_FAIL — Failure
- E_OUTOFMEMORY — Insufficient memory
- E_POINTER — Null pointer argument
- VFW_E_CERTIFICATION_FAILURE — Use of this filter is restricted by a software key
- VFW_E_DUPLICATE_NAME — Failed to add a filter with a duplicate name

Sample code:

```
if(FAILED(hr = m_pGraph->AddFilter(m_TextureRenderer, L"TEXTURERENDERER")))
    return hr;
```

4. Once the custom render class has been added successfully to the filter graph, you need to add the source file. This means actually telling the filter graph which video file to open. For those who remember the filter graph loading process from the previous chapter, this function replaces the `RenderFile` routine. The process of adding the video file to the filter graph is accomplished by calling the **AddSourceFilter** method of the `IGraphBuilder` interface. The syntax and parameters for `AddSourceFilter` are listed below, and then some sample code follows.

HRESULT AddSourceFilter

```
(
    LPCWSTR lpwstrFileName,
    LPCWSTR lpwstrFilterName,
    IBaseFilter **ppFilter
);
```

LPCWSTR lpwstrFileName

Filename of the media file to load.

LPCWSTR lpwstrFilterName

Name of the filter to add. This can be any name you want.

IBaseFilter **ppFilter

Address to receive a pointer to a base filter interface representing the added media.

Sample code:

```
if(FAILED(hr = m_pGraph->AddSourceFilter (wFileName, L"SOURCE", &pFSrc)))
    return hr;
```

5. Great! You've now added the custom video renderer and a source filter representing the loaded media file to the filter graph. Next, you'll need to enumerate through the filter's pins

to find the output pin. The exact reasons for doing this are beyond the scope of this book since such subject matter is specifically related to DirectShow rather than Direct3D. In short, however, as the filters are added to the filter graph, most of them are automatically configured together. And, as video is played back it will travel along the filters and outward through connections called pins. It is therefore our duty, since we're manually building a filter graph, to ensure the output pin — where video and sound will be output — must be the pin that is rendered. If it is not, then we won't see or hear anything from the file. In order to search the filter for the correct output pin, I've coded the following function to do this. As arguments, it takes a filter and a direction to search, and returns its nearest unconnected output pin. This code is as follows:

```
HRESULT GetUnconnectedPin(IBaseFilter *pFilter, PIN_DIRECTION PinDir,
                          IPin **ppPin)
{
    *ppPin = 0;
    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
    HRESULT hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr))
    {
        return hr;
    }
    while (pEnum->Next(1, &pPin, NULL) == S_OK)
    {
        PIN_DIRECTION ThisPinDir;
        pPin->QueryDirection(&ThisPinDir);
        if (ThisPinDir == PinDir)
        {
            IPin *pTmp = 0;
            hr = pPin->ConnectedTo(&pTmp);
            if (SUCCEEDED(hr))
            {
                pTmp->Release();
            }
            else

```

```

        {
            pEnum->Release();
            *ppPin = pPin;
            return S_OK;
        }
    }
    pPin->Release();
}
pEnum->Release();

return E_FAIL;
}

```

6. Once the output pin has been located, it needs to be rendered by the filter graph. This is how you prepare the graph to play your file. This process is performed by calling the `Render` method of `IGraphBuilder`. This method simply takes the output pin as an argument. The code below demonstrates how to use this method.

```

if (FAILED(hr = m_pGraph->Render(pFSrcPinOut)))
    return hr;

```

7. Although the output pin has been rendered, it still hasn't started playing the media file. This will be done when the `Run` method of `IMediaControl` is called. First, however, it's a good idea to get pointers to some more of those standard interfaces. Here's some code to do this:

```

//Create media control and events
if(FAILED(hr = m_pGraph->QueryInterface(IID_IMediaControl, (void
    **)&m_pControl)))
    return hr;

if(FAILED(hr = m_pGraph->QueryInterface(IID_IMediaEventEx, (void
    **)&m_pEvent)))
    return hr;

```

Here's the full code. It creates a filter graph, adds a video renderer, loads a file, connects pins, and then sets a file in motion. And that's

it! That's how video files can be played on textures. A bit complex, I know, but well worth the effort.

```

HRESULT LoadFromFile(char* File)
{
    IBaseFilter* pFSrc = NULL;
    IPin* pFSrcPinOut = NULL;

    HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,
                                   CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&m_pGraph);

    if(FAILED(hr))
        return hr;

    m_TextureRenderer = new CTextureRenderer(NULL, &hr);

    if(FAILED(hr = m_pGraph->AddFilter(m_TextureRenderer,
                                       L"TEXTURERENDERER")))
        return hr;

    WCHAR wFileName[MAX_PATH];
    mbstowcs(wFileName, File, MAX_PATH);

    if(FAILED(hr = m_pGraph->AddSourceFilter (wFileName,
                                             L"SOURCE", &pFSrc)))
        return hr;

    if (FAILED(hr = GetUnconnectedPin(pFSrc, PINDIR_OUTPUT,
                                     &pFSrcPinOut)))
        return hr;

    if (FAILED(hr = m_pGraph->Render(pFSrcPinOut)))
        return hr;

    //Create media control and events
    if(FAILED(hr = m_pGraph->QueryInterface(IID_IMediaControl,
                                             (void **)&m_pControl)))
        return hr;
}

```

```
    if(FAILED(hr = m_pGraph->QueryInterface(IID_IMediaEventEx,  
                                            (void **)&m_pEvent)))  
        return hr;  
  
    if(FAILED(hr = m_pGraph->QueryInterface(IID_IMediaPosition,  
                                            (void **)&m_pMP)))  
        return hr;  
  
    return S_OK;  
}
```

Conclusion

This chapter introduced you to one of the most complex subjects this book covers. Having come this far you should be able to load a video from a file and then copy it to a texture as playback elapses. Now, you should be hungry for some more challenging work. Hopefully, the next chapter will not disappoint as we turn our attention to skeletal animation.

This page intentionally left blank.



Chapter 13

Skeletal Animation

This final chapter covers some pretty advanced subject matter; specifically, skeletal animation. Overall, this chapter aims to cover the following topics:

- What is skeletal animation?
- How to use skinned meshes and bone hierarchies
- How to load frame hierarchies
- How to parse bones
- How to apply keyframed animation to skinned meshes

What Is Skeletal Animation?

If you've played any of the latest video games, I'm sure you've seen skeletal animation at work. Until now we've seen how to move objects from one place to another, and we've also seen how to animate the surfaces of textures. Now, imagine we have a character model — say, a mesh of a woman. We want to show an animation of this woman walking, moving her legs and swinging her arms back and forth as she takes each stride. Up to this point, the only way we could implement this animation would be to export several meshes of the woman at various points in the walk animation and to interpolate between them over time using `D3DXVec3Lerp`. While this is possible, it's also a very, very slow way to implement this animation sequence. A more effective solution would be to employ skeletal animation.

Essentially, *skeletal animation* is a memory-efficient way of animating meshes. For example, if you have a mesh of a person, animal, or some other creature or vehicle whose parts move, then it's likely skeletal animation will prove a real benefit. For skeletal animation, you only need to use one mesh rather than many versions of the same mesh at different keyframes. You encode the animations themselves by way of matrices, which means you can encode potentially many animations and apply them independently to a single mesh (more on this later).

A mesh that is animated using skeletal animation is called a *skinned mesh*, and for the rest of this chapter, we'll be thinking about skeletal animation and skinned meshes in terms of animating a single character mesh of a woman. Being a skinned mesh, it means we can access each part of the mesh's body independently and, furthermore, we can animate each limb using its own transformation matrix. Therefore, we can move the mesh's arms and legs, or other limbs, to our liking without affecting other areas of the mesh. The mesh used for our work in this chapter, like most skinned meshes, will be stored in a Microsoft X file, discussed earlier in this book. This mesh is provided with the Microsoft DirectX SDK and is called Tiny.x. It can be found in the SDK Samples\Media folder.

Skinned Meshes

Tiny.x is a skinned mesh, and skinned meshes are typically stored in X files. The X file will at least contain the following:

- **Mesh**

The mesh itself; this will be the model actually exported from a 3D modeling package. This mesh will contain vertices, materials, and textures.

- **Skin info**

Skinned meshes contain a skin info data structure that goes hand in hand with the mesh. The skin info structure stores a

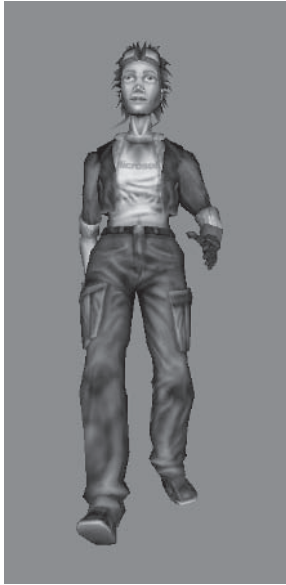


Figure 13.1: *Tiny.x*

collection of objects called bones, which represent the different parts of the mesh, like arms, legs, the torso, etc.

■ **Frame hierarchy**

The file will also contain a hierarchy of frame objects, which each contain a transformation matrix. These frames correspond to each bone in the mesh and represent how the bone — a specific area of the mesh — is to be transformed and posed.

The Structure of a Skinned Mesh

Figure 13.2 contains an illustration of how a skinned mesh, *Tiny.x*, with its frame and bone data can be visualized. As you can see, it's structurally very simple. The file contains a character mesh and a bone hierarchy that corresponds to the various limbs of the mesh. You can think of the bone hierarchy as being the underlying skeleton of the mesh, and the mesh itself as the skin wrapped over the bones. As the skeleton is transformed and moved, the dependent mesh conforms to match the skeleton. So, if you moved the upper arm of the skeleton, the attached lower arm, wrist, and hand would

follow. Additionally, the corresponding character mesh would move according to the skeleton.

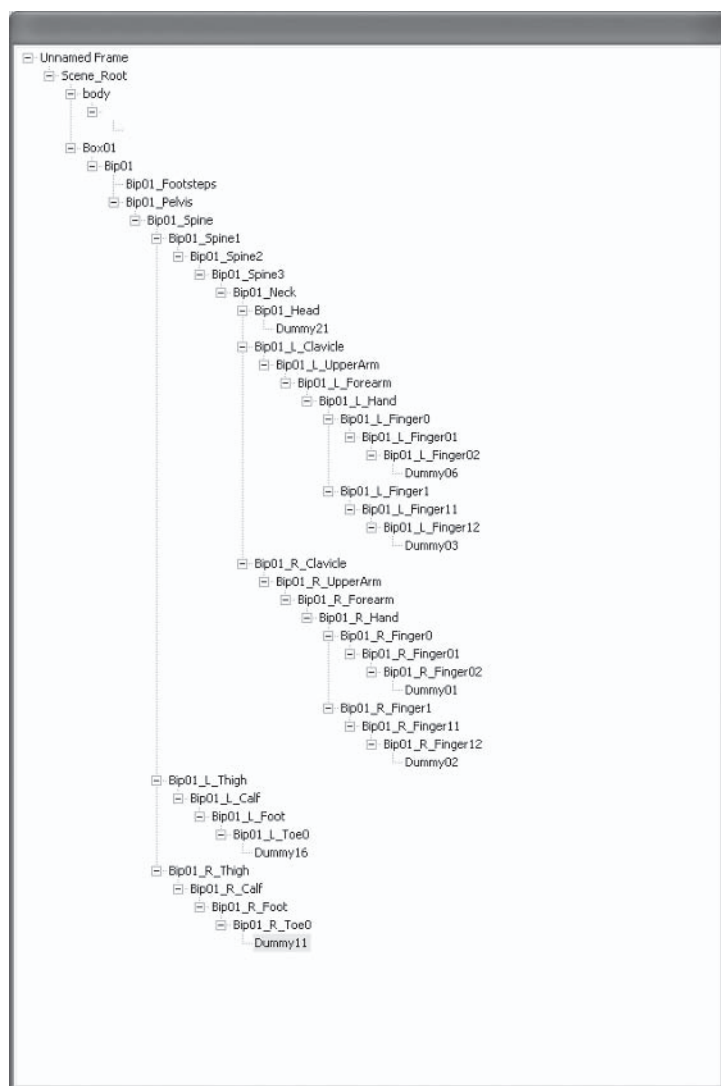


Figure 13.2

Figure 13.2 shows the DirectX mesh viewer with the Tiny.x mesh and its corresponding hierarchy loaded. Now, it should become clear how the skinned mesh file is structured practically. It contains:

- A mesh, which, when loaded, will be represented by a standard ID3DXMesh interface. Furthermore, this mesh will be accompanied by the ID3DXSkinInfo interface, which contains additional information about the skeletal composition of the mesh.
- A bone hierarchy, represented by a hierarchy of D3DXFRAME structures (or frames, as shown earlier in this book when discussing timing and animation). These frames are named after the corresponding bone in the mesh, and also have a transformation matrix representing the pose and movement of the bone. These matrices will be used to change the bones and pose the mesh.

Loading a Skinned Mesh from an X File

The first stage of loading a skinned mesh is to load the actual mesh object and skin info data, and then load the bone hierarchy separately. The reason it's a good idea to keep the mesh and skeleton separate is because you could make many different skeletons, some in a walking pose and some in other poses, and then apply them to the same mesh to pose it differently. Loading the skinned mesh involves a bit more work than just calling a function and passing a filename. Instead you'll need to use the X file interface functions, as discussed in Chapter 6. You'll need to open up the X file manually and cycle through the file's data objects looking for an object whose type matches a mesh. Once found, you can load the skinned mesh and skin info using the **D3DXLoadSkinMeshFromXof** function. This function requires, among other parameters, an X file data object from which to load the mesh, some addresses to receive material and texture data, and an address to receive the mesh's skin info. The syntax and parameters for this function appear below.

```
HRESULT WINAPI D3DXLoadSkinMeshFromXof(
    LPD3DXFILEDATA pxofMesh,
    DWORD Options,
    LPDIRECT3DDEVICE9 pD3DDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER *ppEffectInstances,
```

```

    DWORD *pMatOut,
    LPD3DXSKININFO *ppSkinInfo,
    LPD3DXMESH *ppMesh
);

```

LPD3DXFILEDATA *pxofMesh*

Pointer to an ID3DXFileData object from which to load a mesh.

DWORD *Options*

Specifies creation options for the mesh. Typically, this will be D3DXMESH_SYSTEMMEM. Possible parameters are:

```

D3DXMESH_32BIT
D3DXMESH_DONOTCLIP
D3DXMESH_DYNAMIC
D3DXMESH_IB_DYNAMIC
D3DXMESH_IB_MANAGED
D3DXMESH_IB_SOFTWAREPROCESSING
D3DXMESH_IB_SYSTEMMEM
D3DXMESH_IB_WRITEONLY
D3DXMESH_MANAGED
D3DXMESH_NPATCHES
D3DXMESH_POINTS
D3DXMESH_RTPATCHES
D3DXMESH_SOFTWAREPROCESSING
D3DXMESH_USEHWONLY
D3DXMESH_VB_DYNAMIC
D3DXMESH_VB_MANAGED
D3DXMESH_VB_SHARE
D3DXMESH_VB_SOFTWAREPROCESSING
D3DXMESH_VB_SYSTEMMEM
D3DXMESH_VB_WRITEONLY
D3DXMESH_WRITEONLY

```

LPDIRECT3DDEVICE9 *pD3DDevice*

Pointer to a valid Direct3D device.

LPD3DXBUFFER **ppAdjacency*

Address of a buffer to receive mesh adjacency information.

LPD3DXBUFFER **ppMaterials*

Address of a buffer to receive material information.

LPD3DXBUFFER *ppEffectInstances

Address of a buffer to receive effects information. Can be NULL.

DWORD *pMatOut

Address to receive the number of returned materials.

LPD3DXSKININFO *ppSkinInfo

Address to receive mesh skin information.

LPD3DXMESH *ppMesh

Address to receive a skinned mesh.

The code below demonstrates how to use the `D3DXLoadSkinMeshFromXof` function to load a skinned mesh. The code also loads the mesh's material and textures, and then finally creates a clone of the mesh using the `CloneMeshFVF` method of `ID3DXMesh`. This method was explained in Chapter 7, "Meshes." To recap, the `CloneMeshFVF` method creates a duplicate mesh. The reason the code creates a duplicate mesh is so that as the mesh is animated using skeletal animation later, the original mesh will not be changed and can be reverted to again should it be required.

```
HRESULT Res = D3DXLoadSkinMeshFromXof(m_FileDataObject,
                                     D3DXMESH_DYNAMIC, m_pDevice,
                                     &m_pAdjacencyBuffer,
                                     &m_pMaterialBuffer, NULL,
                                     &NumMaterials, &pSkinInfo, &pMesh);

if(SUCCEEDED(Res))
{
    pMesh->CloneMeshFVF(0, pMesh->GetFVF(), m_pDevice, &pClonedSkinMesh);
}

pMaterials = (D3DXMATERIAL*)m_pMaterialBuffer->GetBufferPointer();
pTextures = new LPDIRECT3DTEXTURE9[NumMaterials];

for(unsigned int Counter = 0; Counter < NumMaterials; Counter++)
{
    pMaterials[Counter].MatD3D.Ambient = pMaterials[Counter].MatD3D.Diffuse;
```

```

    pTextures[Counter] = NULL;
    D3DXCreateTextureFromFile(m_pDevice, pMaterials[Counter].pTextureFilename,
                             pTextures[Counter]);
}

//If pSkinInfo is NULL, then the loaded mesh was not a skinned mesh
return pSkinInfo;

```

Bone Hierarchies

As mentioned previously, in addition to the skinned mesh and skin info stored in a skinned mesh X file, there's also a bone hierarchy. This hierarchy is really a hierarchy of D3DXFRAME objects, which has a name and contains a transformation matrix. Figure 13.2 showed how the frame hierarchy is composed. Take a look at the syntax and parameters below. This shows you how a single D3DXFRAME structure is composed and what its members mean.

```

typedef struct _D3DXFRAME
{
    LPSTR Name;
    D3DXMATRIX TransformationMatrix;
    LPD3DXMESHCONTAINER pMeshContainer;
    struct _D3DXFRAME *pFrameSibling;
    struct _D3DXFRAME *pFrameFirstChild;
}
D3DXFRAME, *LPD3DXFRAME;

```

LPSTR Name

A string member used to store the name of the frame.

D3DXMATRIX TransformationMatrix

A matrix structure reserved for the frame's transformation matrix.

LPD3DXMESHCONTAINER pMeshContainer

Some frames may contain or are associated with one or more meshes. For the purposes of this book, this member will usually be NULL.

```
struct _D3DXFRAME *pFrameSibling
struct _D3DXFRAME *pFrameFirstChild
```

These two parameters are used to keep track of the hierarchy of frames. Remember, the hierarchy is maintained by a linked list. This means each frame needs to maintain a pointer to its next sibling frame, if any, and its first child frame, if any. For more information, review the section in Chapter 9 called “Linked Lists.”

So now that we’ve seen how a frame structure is composed we can simply iterate through the skinned mesh X file, looking for all frame objects, and then create our hierarchy accordingly. This is done exactly how we iterated through X files before, except this time we’ll need to iterate through the objects using recursion to ensure we mimic the frame hierarchy. However, before we dive in and start loading our frame objects right away, we can make our lives a little simpler by deriving a class from the `D3DXFRAME` structure. This will contain some additional members to hold some useful information about our frames. Its class declaration appears below. Afterward we’ll begin to load the frame hierarchy from the X file using our new frame class.

```
class D3DXFRAME2 : public D3DXFRAME
{
    private:
    protected:
    public:
        D3DXMATRIX matCombined;
        D3DXMATRIX matOriginal;

        D3DXFRAME2()
        {
            Name = NULL;
            D3DXMatrixIdentity(&TransformationMatrix);
            D3DXMatrixIdentity(&matOriginal);
            D3DXMatrixIdentity(&matCombined);
            pMeshContainer = NULL;
            pFrameSibling = pFrameFirstChild = NULL;
        }
}
```



```

}

~D3DXFRAME2()
{
    if(pMeshContainer)
        delete pMeshContainer;

    if(pFrameSibling)
        delete pFrameSibling;

    if(pFrameFirstChild)
        delete pFrameFirstChild;
}

D3DXFRAME2* Find(const char* FrameName)
{
    D3DXFRAME2 *pFrame, *pFramePtr;

    if(Name && FrameName && !strcmp(FrameName, Name))
        return this;

    if((pFramePtr = (D3DXFRAME2*)pFrameSibling))
    {
        if((pFrame = pFramePtr->Find(FrameName)))
            return pFrame;
    }

    if((pFramePtr = (D3DXFRAME2*)pFrameFirstChild))
    {
        if((pFrame = pFramePtr->Find(FrameName)))
            return pFrame;
    }

    return NULL;
}

void Reset()
{
    TransformationMatrix = matOriginal;
}

```

```

D3DXFRAME2 *pFramePtr = NULL;

if((pFramePtr = (D3DXFRAME2*)pFrameSibling))
    pFramePtr->Reset();

if((pFramePtr = (D3DXFRAME2*)pFrameFirstChild))
    pFramePtr->Reset();
}

void UpdateHierarchy(D3DXMATRIX *matTransformation = NULL)
{
    D3DXFRAME2 *pFramePtr = NULL;
    D3DXMATRIX matIdentity;

    if(!matTransformation)
    {
        D3DXMatrixIdentity(&matIdentity);
        matTransformation = &matIdentity;
    }

    matCombined = TransformationMatrix * (*matTransformation);

    if((pFramePtr = (D3DXFRAME2*)pFrameSibling))
        pFramePtr->UpdateHierarchy(matTransformation);

    if((pFramePtr = (D3DXFRAME2*)pFrameFirstChild))
        pFramePtr->UpdateHierarchy(&matCombined);
}

void AddChildFrame(D3DXFRAME2 *Frame)
{
    if(Frame)
    {
        if(!pFrameFirstChild)
            pFrameFirstChild = Frame;
        else
        {
            D3DXFRAME* FramePtr = pFrameFirstChild;

            while(FramePtr->pFrameSibling)

```

```

        FramePtr = FramePtr->pFrameSibling;

        FramePtr->pFrameSibling = Frame;
    }
}
};

```

Loading the Bone Hierarchy

In the previous section we developed an extended frame class to hold some additional information about our frames, including an original and combined transformation matrix. This section begins the process of loading a frame hierarchy from the skinned mesh X file. As you will see, the hierarchy is stored like this:

- There will be a root frame at the top of the hierarchy. This will usually correspond to the root bone in the mesh, such as the head or hip.
- Each frame will contain only one transformation matrix as a child. This matrix will be the transformation matrix for the frame and represents the transformation to use for that specific bone.
- Each frame will have none, one, or more child frame objects. The child frame objects represent child bones that are dependent on their parent. As their parent moves, so do all its children.

The function below demonstrates how to cycle through an X file's data objects, finding all frames, and then loads them into the hierarchy. A more detailed explanation of the code follows.

```

HRESULT ProcessObject(LPD3DXFILEDATA DataObject, D3DXFRAME2 *Parent)
{
    //If data object is NULL then exit

    if(!DataObject)
        return E_FAIL;
}

```

```

D3DXFRAME2 *Frame = NULL;

//If data object is a reference then skip object
if(DataObject->IsReference())
    return E_FAIL;

//Is data object a frame?
GUID Type;
DataObject->GetType(&Type)

if(Type == TID_D3DRMFrame)
{
    Frame = new D3DXFRAME2();
    //Get frame name
    SIZE_T Size = 0;
    DataObject->GetName(NULL, &Size);
    Frame->Name = new char[Size];
    DataObject->GetName(Frame->Name, &Size);

    //Is there currently a parent frame?
    if(!Parent)
    {
        if(m_RootFrame)
        {
            //Should not usually occur, but if so, then add as sibling of
            //root frame

            D3DXFRAME* FramePtr = m_RootFrame;

            while(FramePtr->pFrameSibling)
                FramePtr = FramePtr->pFrameSibling;

            FramePtr->pFrameSibling = Frame;
        }
        else
            m_RootFrame = Frame; //Make this frame the root
    }
    else
    {
        Parent->AddChildFrame(Frame); //Add as child of the parent frame
    }
}

```

```

    }
}

//If data object is a transformation matrix
if(Type == TID_D3DRMFrameTransformMatrix)
{
    //Make sure there is a parent frame to add matrix to
    if(Parent)
    {
        D3DXMATRIX *Matrix = NULL;
        SIZE_T Size = 0;

        //Lock matrix data
        DataObject->Lock(&Size, (const void**) &Matrix);

        if(Size == sizeof(D3DXMATRIX))
        {
            //Copy over matrix
            Parent->TransformationMatrix = *(Matrix);
            Parent->matOriginal = Parent->TransformationMatrix;
        }

        //Unlock matrix data
        DataObject->Unlock();
    }
}

//Process child objects

SIZE_T ChildCount = 0;
DataObject->GetChildren(&ChildCount);

for(SIZE_T Counter = 0; Counter < ChildCount; Counter++)
{
    LPD3DXFILEDATA ChildObject = NULL;

    DataObject->GetChild(&ChildObject);

    ProcessObject(ChildObject, Frame);
}

```

```
    return S_OK;  
}  
};
```

This lengthy piece of code is nowhere near as bad as it might first seem. Why not take a moment to go back and read through it again, step by step? One of the main variables in this piece of code is `m_RootFrame`, which is a global `D3DXFRAME*` pointer representing the root (topmost) frame in the hierarchy. This variable is set to `NULL`. Additionally, this function is recursive, and for every frame created it'll be passed on as the parent parameter to which child frames can be added. Overall, you'll notice that loading a frame hierarchy is about going through the following steps every time a frame object is located in the X file.

1. Create a `D3DXFRAME2` object.
2. Get the name of the frame and assign it to the frame object.
3. Check to see whether your root frame object (`m_RootFrame`) has already been assigned a frame object. If it's `NULL`, then there is currently no root frame and this is the first frame you've encountered in the hierarchy. Therefore, this frame should become the root frame. However, if the root frame is not `NULL`, then the root frame has already been assigned a frame object previously, meaning the current frame is not the root frame. So, in this case, you must add the current frame as a child frame to the frame object passed as the parent parameter. In other words, the current frame is a child object to the parent frame.

The other aspect to this function, which is quite self explanatory, is about assigning a matrix to a frame. As this code demonstrates, a frame's matrix is actually stored as a separate child object to the frame. This means that as you cycle through all the X file's data objects, you must also check to see whether a matrix object was encountered. Whenever a matrix object is encountered, you do the following:

1. Check to see that there is a parent frame to which the matrix should belong. If the parent parameter is NULL, then there is no parent frame and the operation can be ignored. However, if the parent parameter points to a valid frame, then the matrix can be processed.
2. Lock the data and validate its size.
3. Copy the matrix data over into the frame.

Mapping the Bone Hierarchy to the Mesh

At this point you've loaded two distinct items from the skinned mesh X file: the mesh itself and the associated bone hierarchy. However, there's currently no discernible linkage between the two. Now it's time to iterate through the frame hierarchy that you've created and associate each frame with a specific area of the mesh so that its respective transformation matrix will later be applied to the correct area. In other words, you need to process all the frames and make sure each frame is associated with a specific bone of the mesh's skeleton. Thus, you're establishing the relationship between the skeleton and the mesh.

To access the skeletal information from the mesh, you call upon the skin info interface (ID3DXSkinInfo). This was loaded along with your mesh and it contains a plethora of data about the skeletal composition underlying the skin of your mesh. Using this interface we can iterate through every bone in the mesh. The following function shows you how to map the frames to the bones, and the code is explained afterward.

```
VOID MapFramesToBones()
{
    if(IsSkinnedMesh())
    {
        DWORD NumBones = m_pMeshContainer->pSkinInfo->GetNumBones();

        m_pMeshContainer->ppFrameMatrices = new D3DXMATRIX*[NumBones];
        m_pMeshContainer->pBoneMatrices = new D3DXMATRIX[NumBones];
```

```

for(DWORD Counter = 0; Counter < NumBones; Counter++)
{
    const char* BoneName = m_pMeshContainer->pSkinInfo-
        >GetBoneName(Counter);

    D3DXFRAME2* FramePtr = m_pFrames->Find(BoneName);

    if(FramePtr)
        m_pMeshContainer->ppFrameMatrices[Counter] =
            &FramePtr->matCombined;
    else
        m_pMeshContainer->ppFrameMatrices[Counter] = NULL;
}
}

```

The above code calls upon the `GetNumBones` function to get the number of bones in the mesh. Then it allocates enough matrices to hold the bone transformations, and finally it cycles through every bone. As it does so, it takes the name of each bone and searches through the corresponding frame hierarchy until it finds a match. Once found, it keeps a pointer to the frame's transformation matrix for later reference.

Updating the Mesh

Now that the bones have finally been mapped to the mesh, it's almost ready to render. Before we can do this, though, we need to make sure the mesh is updated on every frame so that it's set into its correct pose. In other words, what we really mean is we must cycle through all the bones in the hierarchy and apply their specific transformation matrices to the corresponding area of the mesh. For example, the mesh's arm might be raised. This means the arm frame will contain a matrix that'll position the arm correctly. Take a look at the following code to see how the bone hierarchy is updated.


```

VOID Update()
{
    m_pFrames->UpdateHierarchy(m_Transform);

    DWORD NumBones = m_pMeshContainer->pSkinInfo->GetNumBones();

    for(DWORD Counter = 0; Counter < NumBones; Counter++)
    {
        m_pMeshContainer->pBoneMatrices[Counter] =
            (*m_pMeshContainer->pSkinInfo->
             >GetBoneOffsetMatrix(Counter));

        if(m_pMeshContainer->ppFrameMatrices[Counter])
            m_pMeshContainer->pBoneMatrices[Counter] *=
                (*m_pMeshContainer->ppFrameMatrices[Counter]);
    }

    void *SrcPtr, *DesPtr;

    m_pMeshContainer->MeshData.pMesh->LockVertexBuffer(D3DLOCK_READONLY,
        (void**) &SrcPtr);

    m_pMeshContainer->pSkinMesh->LockVertexBuffer(0, (void**) &DesPtr);

    m_pMeshContainer->pSkinInfo->UpdateSkinnedMesh(m_pMeshContainer->
        pBoneMatrices, NULL, SrcPtr, DesPtr);

    m_pMeshContainer->MeshData.pMesh->UnlockVertexBuffer();
    m_pMeshContainer->pSkinMesh->UnlockVertexBuffer();
}

```

First, the above function calls the Update method on the root frame of the hierarchy, passing a transformation matrix to it. The called method then updates the root frame and passes the transformation recursively down the hierarchy to update every other frame. Overall, this process positions and transforms the mesh. Next, the above function calls the GetNumBones method of ID3DXSkinInfo to retrieve the number of bones in the mesh. Then, it cycles through

each bone. As it does so, it retrieves each bone offset matrix using the `GetBoneOffsetMatrix` method of `ID3DXSkinInfo`. This is a matrix that tells us where the bone joints are. Because all transformations are applied at the world origin, we'll need the offset matrix in order to first move the bones to their joints, and then apply the transformations to each one. Then finally you need to apply all these updates to the mesh itself. To do this, you need to lock the vertex buffers to both meshes, the original mesh and the cloned mesh, and call the **UpdateSkinnedMesh** method of `ID3DXSkinInfo`. The syntax and parameters for this function appear below.

```
HRESULT UpdateSkinnedMesh
(
    const D3DXMATRIX *pBoneTransforms,
    const D3DXMATRIX *pBoneInvTransposeTransforms,
    LPCVOID pVerticesSrc,
    PVOID pVerticesDst
);
```

const D3DXMATRIX *pBoneTransforms

Pointer to an array of transform matrices. These will actually be your transform matrices for each bone.

const D3DXMATRIX *pBoneInvTransposeTransforms

Inverse transpose of the bone transform matrix. This can be NULL.

LPCVOID pVerticesSrc

Pointer to the vertices in your source mesh.

PVOID pVerticesDst

Pointer to the vertices in your destination mesh.

Rendering the Mesh

Like updating the mesh, rendering occurs on every frame. Rendering is the actual process of drawing the mesh to the screen, and the process of drawing skinned meshes is just like drawing any other mesh.

```
bool CXSkinnedMesh::Render()
{
    if(m_pMeshContainer->pSkinMesh)
    {
        Update();

        for(unsigned int Counter = 0; Counter < m_pMeshContainer-
            >NumMaterials; Counter++)
        {
            m_pDevice->SetMaterial(&m_pMeshContainer->
                pMaterials[Counter].MatD3D);
            m_pDevice->SetTexture(0, m_pMeshContainer->pTextures[Counter]);
            m_pMeshContainer->pSkinMesh->DrawSubset(Counter);
        }

        return true;
    }

    return false;
}
```

Animating the Skeleton

By this point you should be able to load a skinned mesh and bone hierarchy from an X file and render it to the screen. However, so far the mesh doesn't move; it just stands there, still and motionless in its starting pose. You could choose to manipulate those bone matrices manually and animate the skeleton yourself; however, this would be a long and tedious process. A better way to animate the skeleton is to use keyframed animation. These animations can be

created in a 3D package and exported to the X file format. This means you can create many different keyframed animations where the skeleton of the mesh is animated. You can then load these in separately from the mesh and apply the different animations to the skeleton when required. For example, you could make a keyframed animation of a skeleton jumping, and then another of a skeleton running. Then finally, when you want your mesh to run, you simply apply the running animation. When you want your mesh to jump, you just apply the jumping animation. Using this method, you won't need to change a single line of code when you want to use different animations.

To recap, first let's remember exactly what keyframed animation is. *Keyframed animation* is a memory-efficient way of representing animations. Simply put, you just store the state of the mesh at specific intervals in the animation and, as time advances through the animation during playback, you interpolate between the keyframes. This means you don't need to store the state of the mesh on every frame. That's how keyframed animation works.

Now it's almost time to start animating our skinned mesh. Before we can do this, however, we'll need to code a few classes to hold our animation data, such as the keyframes themselves. Specifically, we'll need to code a class that'll hold all the keyframe data that will be read in from an X file, which can be the same X file as the mesh or a separate X file. Furthermore, we'll need to code an animation controller we can use like a media player to start and stop animation playback. The animation field of data that must be encapsulated can be summarized as follows.

■ **Keyframe**

A *keyframe* records the state of some part of the mesh at a specific time in the animation. For example, if the mesh's forearm raises during the animation, then at various points there will be keyframes recording the state of the forearm and the time at which that change occurs. A class can be coded to hold a keyframe that stores a DWORD time at which the change occurs and a matrix structure that encodes the exact transformation to apply at that time. The class appears as follows:

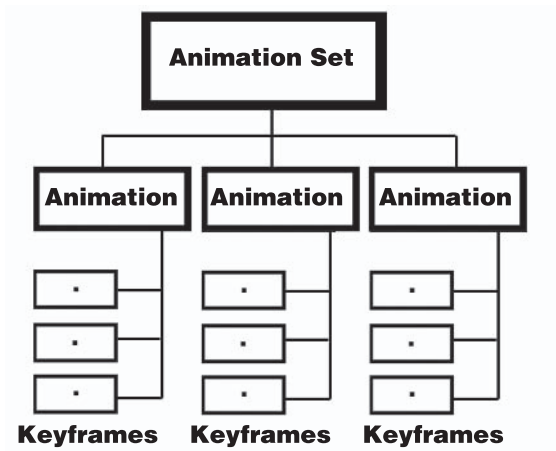


Figure 13.3

```

class CMatrixKey
{
    private:
    protected:
    public:
        DWORD m_Time;           //Time of change
        D3DXMATRIX m_Matrix;    //Recorded keyframe state
};
  
```

■ Animation

In the context of skeletal animation, an *animation* is a collection of related keyframes. Generally, we think of an animation as referring to the entire animation, which is the entire movement of the mesh from swinging arms and running legs. However, in this case, an animation is a collection of all the keyframes for a specific body part. So, all the keyframes that move the forearm form one animation, and all the keyframes that animate the lower leg form another animation. A class to encapsulate an animation is coded below. It stores a pointer to an array of keyframe classes, and also maintains a next pointer so it can be used in a linked list hierarchy. Take a look.

```

class CAnimation
{
    private:
  
```

```

protected:
public:
    CAnimation *m_Next;
    D3DXFRAME *m_AnimBone;
    DWORD m_Length;

    CMatrixKey *m_Keys;
    DWORD m_NumKeys;
    char* m_Name;

    CAnimation()
    {
        m_Next = NULL;
        m_AnimBone = NULL;
        m_Length = m_NumKeys = 0;
        m_Keys = NULL;
        m_Name = NULL;
    }

    ~CAnimation()
    {
        delete [] m_Keys;
    }
};

```

■ Animation set

An animation set is what people generally think of as the animation; it's the entire animation process. Thus, an *animation set* can correctly be defined as a collection of animations. A class to encapsulate an animation set can be coded as shown below.

```

class CAnimationSet
{
private:
protected:
public:
    CAnimationSet *m_Next;
    CAnimation *m_Animations;
    DWORD m_NumAnimations;
    char* m_Name;

```

```

CAnimationSet()
{
    m_Next = NULL;
    m_Name = NULL;
    m_Animations = NULL;
    m_NumAnimations = 0;
}

~CAnimationSet()
{
}

HRESULT AddAnimationObject(CAnimation *Anim)
{
    if(!Anim)
        return E_FAIL;

    if(!m_Animations)
        m_Animations = Anim;
    else
    {
        CAnimation *Anims = m_Animations;

        while(Anims->m_Next)
            Anims = Anims->m_Next;

        Anims->m_Next = Anim;
    }

    m_NumAnimations++;

    return S_OK;
}
};

```

Loading Animations

Loading animations is simply a process of reading through an X file, searching for some keyframe and animation templates, and then building your animation hierarchy using the class just defined. Once you have a valid animation hierarchy constructed in memory, you can then begin the process of animating the skinned mesh.

Typically, the following code for loading animations and the code that follows for playing animations will all be coded into a single class, perhaps named `CAnimationController`. This class can then be used to load and play animations on a skinned mesh. Loading animations from a file might begin like this:

```
HRESULT LoadFromFile(char *File)
{
    if(!FileExists(File))
        return E_FAIL;

    LPD3DXFILE pFile = NULL;
    LPD3DXFILEENUMOBJECT Enum = NULL;

    if(SUCCEEDED(D3DXFileCreate(&pFile)))
    {
        pFile->RegisterTemplates(D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES);

        if(SUCCEEDED(pFile->CreateEnumObject(File,
            D3DXF_FILELOAD_FROMFILE, &Enum))
        {
            SIZE_T Size = 0;

            Enum->GetChildren(&Size);

            for(SIZE_T Counter = 0; Counter < Size; Counter++)
            {
                LPD3DXFILEDATA DataObject = NULL;
                Enum->GetChild(Counter, &DataObject);
                ProcessItems(DataObject);
            }
        }
    }
}
```



```

        DataObject->Release();
    }

    Enum->Release();
}

pFile->Release();
}

return S_OK;
}

```

Nothing too special about this code so far. It just opens an X file and cycles through every top-level object, calling `ProcessItems` each time. The custom-made `ProcessItems` function is where a lot of work will occur. In short, this will be a recursive function designed to load an animation hierarchy. Take a look at its definition, and then see the explanation that follows.

```

HRESULT ProcessItems(LPD3DXFILEDATA Object, CAnimationSet *AnimSet =
                    NULL, CAnimation *Anim = NULL)
{
    if(!Object)
        return E_FAIL;

    CAnimationSet *AnimationSet = AnimSet;
    CAnimation *Animation = Anim;

    GUID guid;

    Object->GetType(&guid);

    if((guid == TID_D3DRMAnimationSet) && (!Object->IsReference()))
    {
        SIZE_T Length = 0;

        AnimationSet = new CAnimationSet();

        Object->GetName(NULL, &Length);
        AnimationSet->m_Name = new char[Length];
    }
}

```

```

Object->GetName(AnimationSet->m_Name, &Length);

//Add animation set
if(!m_AnimationSets)
    m_AnimationSets = AnimationSet;
else
{
    CAnimationSet *AnimSets = m_AnimationSets;

    while(AnimSets->m_Next)
        AnimSets = AnimSets->m_Next;

    AnimSets->m_Next = AnimationSet;
}

m_NumAnimationSets++;
}

if((guid == TID_D3DRMAnimation) && (!Object->IsReference()))
{
    if(AnimSet)
    {
        SIZE_T Length = 0;
        Animation = new CAnimation();
        Object->GetName(NULL, &Length);
        Animation->m_Name = new char[Length];
        Object->GetName(Animation->m_Name, &Length);

        AnimSet->AddAnimationObject(Animation);
        ProcessKeyFrames(Object, AnimationSet, Animation);
    }
}

SIZE_T Size = 0;

Object->GetChildren(&Size);

for(SIZE_T Counter = 0; Counter < Size; Counter++)
{
    LPD3DXFILEDATA DataObject = NULL;

```

```

    Object->GetChild(Counter, &DataObject);
    ProcessItems(DataObject, AnimationSet, Animation);
    DataObject->Release();
}

return S_OK;
}

```

This function looks quite nasty, but just take a moment to read through it again and you'll see it's not as bad as it first seems. The principle is simple. The function does the following:

- It cycles through every data object in the file for an animation set. When an animation set is encountered it creates a corresponding animation set object. Afterward, the animation set is assigned a name and then is added to a list of animation sets. In this case, there will only be one animation set because our mesh has only one walking animation; however, for more advanced scenarios, there could be more. Then processing continues and the function will cycle through the file for every child object because, since we know an animation set has been encountered, there are likely to be animation objects, which in turn will contain keyframe objects. Notice how, when the function is called recursively after an animation set has been encountered, it will pass on the valid animation set (AnimSet) as a function parameter so that each subsequent instance of the function can access the previously created animation set. This is important later when child animation objects need to be added to the parent animation set.
- Once an animation set is encountered it's time to search for animation objects, which are to be added as child objects of the animation set. Every time an animation template is found in the X file (TID_D3DRMAnimation), a corresponding animation object is created. Furthermore, it's given a name and is added to the animation set. Finally, a new function is called (ProcessKeyFrames). This function takes an animation template from an X file, cycles through all child keyframe templates, and creates corresponding keyframe objects.

As I mentioned, the `ProcessKeyFrames` function creates all the keyframes for an animation object. Its definition appears below, and then an explanation follows.

```
HRESULT ProcessKeyFrames(LPD3DXFILEDATA Object, CAnimationSet *AnimSet =
                        NULL, CAnimation *Anim = NULL)
{
    if((!Object) || (!AnimSet) || (!Anim))
        return E_FAIL;

    SIZE_T Size = 0;

    Object->GetChildren(&Size);

    for(SIZE_T Counter = 0; Counter < Size; Counter++)
    {
        LPD3DXFILEDATA DataObject = NULL;
        Object->GetChild(Counter, &DataObject);

        GUID guid;

        DataObject->GetType(&guid);

        if((guid == TID_D3DRMFrame) && (!Object->IsReference()))
        {
            SIZE_T Length = 0;
            char *Name = NULL;

            DataObject->GetName(NULL, &Length);
            Name = new char[Length];
            DataObject->GetName(Name, &Length);
            Anim->m_AnimBone = m_Frames->Find(Name);
            delete [] Name;
        }

        if((guid == TID_D3DRMAnimationKey) && (!Object->IsReference()))
        {
            DWORD *DataPtr = NULL;
            SIZE_T BufferSize = 0;
```

```

        if(SUCCEEDED(DataObject->Lock(&BufferSize, (const
                                         void**) &DataPtr)))
        {
            DWORD Type = *DataPtr++;
            DWORD NumKeys = *DataPtr++;

            if(Type == 4)    //if matrix key
            {
                CMatrixKey *Key = new CMatrixKey[NumKeys];

                for(DWORD Counter = 0; Counter < NumKeys;
                    Counter++)
                {
                    Key[Counter].m_Time = *DataPtr++;

                    if(Key[Counter].m_Time > Anim->m_Length)
                        Anim->m_Length = Key[Counter].m_Time;

                    DataPtr++;

                    D3DXMATRIX *mPtr = (D3DXMATRIX*) DataPtr;

                    Key[Counter].m_Matrix = *mPtr;
                    DataPtr+=16;
                }

                Anim->m_Keys = Key;
                Anim->m_NumKeys = NumKeys;
            }
        }

        DataObject->Unlock();
        ProcessKeyFrames(DataObject, AnimSet, Anim);
    }

    DataObject->Release();
}

return S_OK;
}

```

This function is quite simple overall. It takes an X file data object (which, in this case, should be an animation template) and cycles through all child objects. On each iteration it does the following:

- Checks to see whether a frame object is found (remember that a frame corresponds to a bone in the mesh). If a frame is found, then this means we have not found a keyframe in this case, but instead the frame tells us to which bone in the mesh this animation corresponds. For example, if we found a frame called forearm, then we know this animation, and all its keyframes, relate to the forearm. So, if a frame is found we must take its name. Then, we search through the frame hierarchy of the mesh (which we loaded earlier) and we find a matching frame. We then store a reference to this frame in our animation object so that, later, we can reference this frame as we come to update our matrices as the animation elapses.
- If animation keys are found, then we first retrieve the total numbers of keys associated with this animation (frame/bone). Afterward, we initialize the keys pointer (`m_Keys`) of the animation object to hold this many keys, then we simply lock the data buffer and copy over the keys from the file and into the buffer.

Voilà! And finally the entire animation data has been read in from an X file and into a hierarchy in memory. Once an animation has been loaded it's all simple from now on.

Playing Animations

Here's the fun part, the part where all your hard work pays off and you can start to play your loaded animations. By now you can load a skinned mesh and load in an animation hierarchy, which consists of animation sets, animations, and animation keys. Each animation object relates to a bone in the mesh, and each of its keys refers to the bone's state at a specific point in time. In other words, each key contains a time (the time when the bone is in that state) and a

transformation matrix that contains the exact transformation that should be applied to the bone at that time. However, there will be times when the current time of the animation does not exactly fall into any one key, but between two different keys. For example, there may be a key at 0 seconds into the animation and another key at 100 milliseconds into the animation. The current time might be 50 milliseconds, so we're effectively halfway between the two keyframes. In this case, we must then take both matrices — one from the start key and one from the end key — and interpolate between the two to arrive at the correct matrix, which represents our transformation at that time. To update the animation we must ensure the keyframes are processed like this on every frame. Take a look at the Update function below to see how animations can be applied to the skinned mesh.

```
VOID Update()
{
    m_CurrentTime = timeGetTime() - m_StartTime;

    if(!m_AnimationSets)
        return;

    CAnimationSet *Sets = m_AnimationSets;

    while(Sets)
    {
        CAnimation *Animation = Sets->m_Animations;

        while(Animation)
        {
            CMatrixKey *Keys = Animation->m_Keys;

            CMatrixKey *StartKey = NULL;
            CMatrixKey *EndKey = NULL;

            for(DWORD Counter = 0; Counter < Animation->m_NumKeys; Counter++)
            {
                if(m_CurrentTime >= Keys[Counter].m_Time)
                    StartKey = &Keys[Counter];
```

```

        else
            EndKey = &Keys[Counter];
        }

        if(!EndKey)
            Animation->m_AnimBone->TransformationMatrix =
                StartKey->m_Matrix;
        else
        {
            DWORD TimeDifference = EndKey->m_Time - StartKey->m_Time;

            float Scalar = (float) (m_CurrentTime - StartKey-
                                   >m_Time)/TimeDifference;

            D3DXMATRIX Matrix = EndKey->m_Matrix - StartKey->m_Matrix;

            Matrix *= Scalar;
            Matrix += StartKey->m_Matrix;

            Animation->m_AnimBone->TransformationMatrix = Matrix;
        }

        Animation = Animation->m_Next;
    }

    Sets = Sets->m_Next;
}
}

```

Once again, this is one of those functions that looks worse than it really is on the first occasion you view it. Browse through it a few times, though, and you'll likely see a much simpler picture. This function occurs on every frame and updates the skinned mesh according to the elapsed time in the animation. First, the function cycles through all animation sets (in this example, there's only one). Then it cycles through all animation objects, effectively looping through each bone. Then it cycles through each keyframe. As it encounters each keyframe the function determines, using the elapsed time, which two keyframes the current time falls between. Once this has been established, the function creates a scalar value

between 0 and 1 to determine by how much the matrix should be interpolated to arrive at the final matrix, which will be applied to the skinned mesh. This process is repeated for every bone in the mesh. And that's it. You should now be equipped to render this mesh in its full, animated glory. This is shown in Figure 13.4.

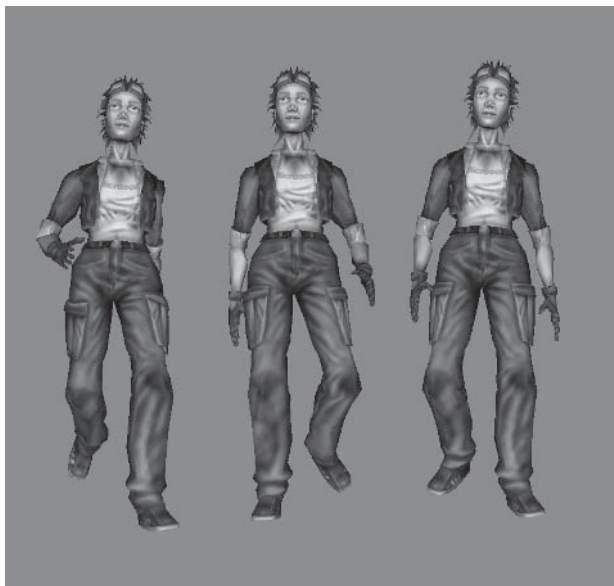


Figure 13.4

Conclusion

Wow, that was a tough chapter overall. Skinned meshes can be a real nightmare when approached unguided, and I hope this chapter provided good and approachable insight into the world of skeletal animation. Since this topic is really tough in comparison to much of the other subject matter presented in this book, I reserved it for last.

Now that you've reached the end of this book and completed one of the toughest challenges it had to offer, I hope your knowledge of DirectX has passed the summit of DirectX knowledge and that you're well on your way to picking up new and interesting things with confidence.

Afterword

Congratulations! You've reached the end of this book and no doubt picked up a lot of useful information. Now, you can go on to start creating some rather advanced products and you're also in a good position to begin learning the remaining facets of Direct3D, namely the programmable pipeline, which includes vertex and pixel shaders. So, where to go from here then? Well, the best place to start is by practicing the subject matter covered in this book by making some sample projects and mini games. Then, once you're confident you've come to grips with the API, I recommend skimming through the recommended reading section to look for books on more advanced topics.

And, on that note, I'll leave you and sign off. I hope my book has proved valuable and has provided you with the inspiration to learn more. Thank you and happy coding.

Alan Thorn

`directx_user_interfaces@hotmail.com`

This page intentionally left blank.



Appendix A

DirectX Q&A

Q. If a DirectX function fails, can I retrieve the error name and description as a string?

A. Yes. DirectX provides two functions: `DXGetErrorDescription9` and `DXGetErrorString9`. These functions accept `HRESULT` values, as returned by DirectX functions, and translate the error code to informative string data.



TIP. These functions are good for displaying DirectX errors to the user in message boxes.

Q. How do I enumerate all the display adapters attached to a user's computer?

A. To enumerate all the display devices attached to a computer you should use a combination of functions.

To retrieve the actual count of display adapters, you call the `GetAdapterCount` method. This function accepts no parameters and returns an unsigned integer representing the number of attached display adapters.

To get driver information about a specific device, such as the device name and driver version, you call the **`GetAdapterIdentifier`** method. This method populates a `D3DADAPTER_IDENTIFIER9` structure. The syntax and parameters for this function follow.

```

HRESULT GetAdapterIdentifier
(
    UINT Adapter,
    DWORD Flags,
    D3DADAPTER_IDENTIFIER9 *pIdentifier
);

```

UINT Adapter

Ordinal number that denotes the display adapter for which we want to retrieve information. 0 is the default adapter, but the number can range to GetAdapterCount–1.

DWORD Flags

Flags can be set to either 0 or D3DENUM_WHQL_LEVEL. If D3DENUM_WHQL_LEVEL is specified, this call can connect to the Internet to download new Microsoft Windows Hardware Quality Labs (WHQL) certificates.

D3DADAPTER_IDENTIFIER9 *pIdentifier

Address to receive a D3DADAPTER_IDENTIFIER9 structure containing information about a specific device. The structure looks like this:

```

typedef struct _D3DADAPTER_IDENTIFIER9 {
    char Driver[MAX_DEVICE_IDENTIFIER_STRING];
    char Description[MAX_DEVICE_IDENTIFIER_STRING];
    char DeviceName[32];
    LARGE_INTEGER DriverVersion;
    DWORD DriverVersionLowPart;
    DWORD DriverVersionHighPart;
    DWORD VendorId;
    DWORD DeviceId;
    DWORD SubSysId;
    DWORD Revision;
    GUID DeviceIdentifier;
    DWORD WHQLLevel;
} D3DADAPTER_IDENTIFIER9;

```

Q. Does Direct3D support multiple monitors?

A. Yes, provided you have a separate display adapter associated with each monitor. To create multimonitor applications, you simply enumerate through the devices and create them. Then you handle each device as you would normally, and each device should render to its specific monitors.

Q. Can I retrieve a device's capabilities? In other words, can I find out what my graphics card can and can't do?

A. Yes. To do this, you should call the `GetDeviceCaps` method of `IDirect3DDevice9`. This method returns the capabilities of the card represented by the `IDirect3DDevice9` interface. The `GetDeviceCaps` method returns a `D3DCAPS9` structure describing the capabilities of the card. You should refer to the SDK documentation for more information. The `D3DCAPS9` structure looks like this:

```
typedef struct _D3DCAPS9 {
    D3DDEVTYPE DeviceType;
    UINT AdapterOrdinal;
    DWORD Caps;
    DWORD Caps2;
    DWORD Caps3;
    DWORD PresentationIntervals;
    DWORD CursorCaps;
    DWORD DevCaps;
    DWORD PrimitiveMiscCaps;
    DWORD RasterCaps;
    DWORD ZCmpCaps;
    DWORD SrcBlendCaps;
    DWORD DestBlendCaps;
    DWORD AlphaCmpCaps;
    DWORD ShadeCaps;
    DWORD TextureCaps;
    DWORD TextureFilterCaps;
    DWORD CubeTextureFilterCaps;
    DWORD VolumeTextureFilterCaps;
    DWORD TextureAddressCaps;
    DWORD VolumeTextureAddressCaps;
    DWORD LineCaps;
```

```
DWORD MaxTextureWidth;  
DWORD MaxTextureHeight;  
DWORD MaxVolumeExtent;  
DWORD MaxTextureRepeat;  
DWORD MaxTextureAspectRatio;  
DWORD MaxAnisotropy;  
float MaxVertexW;  
float GuardBandLeft;  
float GuardBandTop;  
float GuardBandRight;  
float GuardBandBottom;  
float ExtentsAdjust;  
DWORD StencilCaps;  
DWORD FVFCaps;  
DWORD TextureOpCaps;  
DWORD MaxTextureBlendStages;  
DWORD MaxSimultaneousTextures;  
DWORD VertexProcessingCaps;  
DWORD MaxActiveLights;  
DWORD MaxUserClipPlanes;  
DWORD MaxVertexBlendMatrices;  
DWORD MaxVertexBlendMatrixIndex;  
float MaxPointSize;  
DWORD MaxPrimitiveCount;  
DWORD MaxVertexIndex;  
DWORD MaxStreams;  
DWORD MaxStreamStride;  
DWORD VertexShaderVersion;  
DWORD MaxVertexShaderConst;  
DWORD PixelShaderVersion;  
float PixelShaderTexMaxValue;  
DWORD DevCaps2;  
float MaxNpatchTessellationLevel;  
UINT MasterAdapterOrdinal;  
UINT AdapterOrdinalInGroup;  
UINT NumberOfAdaptersInGroup;  
DWORD DeclTypes;  
DWORD NumSimultaneousRTs;  
DWORD StretchRectFilterCaps;  
D3DVSHADERCAPS2_0 VS20Caps;
```

```

D3DPSHADERCAPS2_0 PS20Caps;
DWORD VertexTextureFilterCaps;
DWORD MaxVShaderInstructionsExecuted;
DWORD MaxPShaderInstructionsExecuted;
DWORD MaxVertexShader30InstructionSlots;
DWORD MaxPixelShader30InstructionSlots;
DWORD Reserved2;
DWORD Reserved3;
} D3DCAPS9;

```

Q. By using the color macros `D3DCOLOR_XRGB` and `D3DCOLOR_ARGB`, I can create colors according to my own red, green, blue, and alpha settings. However, if I have a single `DWORD` color value, how I can extract the individual RGBA values?

A. To extract the red, green, blue, and alpha components from a `DWORD` color value, you can code the following macro:

```

#define ExtractAlpha(x) ((x>>24)&255)
#define ExtractRed(x) ((x>>16)&255)
#define ExtractGreen(x) ((x>>12)&255)
#define ExtractBlue(x) (x&255)

```

Q. If I want to render text to the screen, how can I do this?

A. You should use the `ID3DXFont` interface. This is used very similarly to `ID3DXSprite`. You can create this interface by using either `D3DXCreateFont` or `D3DXCreateFontIndirect`.

Q. Can I copy pixels between two surfaces where stretching and resizing is allowed?

A. Yes. To do this, you should call the `StretchRect` method of `IDirect3DDevice9`. This function copies pixels between two surfaces and allows stretching under certain restrictions. To view these restrictions, please consult the DirectX SDK.

Q. My application renders a lot of stuff, and since everything I render requires a lot of changes to my rendering settings, like `SetRenderState`, I need to code an awful lot of lines just to prepare

a device for rendering. Is there any way I can store all my rendering settings and then set the device to these in one single batch?

A. Yes, you can use state blocks. State blocks can save the current state of a device. Then you can change the device settings and restore them to the original settings that were saved in the state block. A state block can save the following settings about a device:

- Vertex state
- Pixel state
- Each texture assigned to a sampler
- Each vertex texture
- Each displacement map texture
- The current texture palette
- For each vertex stream: a pointer to the vertex buffer, each argument from `IDirect3DDevice9::SetStreamSource`, and the divider (if any) from `IDirect3DDevice9::SetStreamSourceFreq`
- A pointer to the index buffer
- The viewport
- The scissors rectangle
- The world, view, and projection matrices
- The texture transforms
- The clipping planes (if any)
- The current material

To create a state block, you call the **CreateStateBlock** method of `IDirect3DDevice9`. This returns a valid `IDirect3DStateBlock9*` pointer, which contains the current state of the device. The syntax and parameters for this function appear below.

```
HRESULT CreateStateBlock(
    D3DSTATEBLOCKTYPE Type,
    IDirect3DStateBlock9 **ppSB
);
```

D3DSTATEBLOCKTYPE Type

Indicates the type of settings you wish to record in the state block. This can be any of the following values:

D3DSBT_ALL

Capture the current device state.

D3DSBT_PIXELSTATE

Capture the current pixel state.

D3DSBT_VERTEXSTATE

Capture the current vertex state.

IDirect3DStateBlock9 **ppSB

Address to receive a valid state block interface pointer.

Once the state block has been created and has captured the current device state, you can proceed to change the device state as needed. Then, once you wish to restore the settings recorded in the state block, you can call the Apply method of IDirect3DStateBlock9. This function requires no arguments.

Q. Can I save the pixel data on a surface or texture to a file?

A. Yes. To save a texture or surface to a file, you call D3DXSaveTextureToFile and D3DXSaveSurfaceToFile respectively.

Q. Can I see the standard Windows GDI dialog boxes when running full-screen Direct3D applications?

A. Yes. To do this you must ensure the dialog boxes are created as child windows of the Direct3D parent window and that you call the SetDialogBoxMode method of IDirect3DDevice9. You must pass True to this function to enable GDI dialog boxes.

Q. How can I set the mouse cursor from a Direct3D surface?

A. To use a Direct3D surface as the mouse cursor, you must call upon two methods from the `IDirect3DDevice9` interface: `SetCursorPosition` and `SetCursorProperties`. `SetCursorPosition` will set the actual X and Y positioning of your cursor. `SetCursorProperties` allows you to specify a surface to use as a cursor and also allows you to specify the X and Y hotspot of the cursor — in other words, the actual tip of the cursor where clicking is activated.

Q. How can I translate the X and Y mouse click in screen space as a position in 3D object space?

A. There are many ways to do this. Effectively you wish to take the XY location of the mouse in screen space and translate this to be a directed ray into 3D object space. You can then test this ray to see what 3D geometry it intersects. To do this, you call upon the **D3DXVec3Unproject** function, which translates coordinates from screen space into object space. You can also translate object space into screen space by calling `D3DXVec3Project`. The syntax and parameters for `D3DXVec3Unproject` are as follows:

```
D3DXVECTOR3 *WINAPI D3DXVec3Unproject
(
    D3DXVECTOR3 *pOut,
    CONST D3DXVECTOR3 *pV,
    CONST D3DVIEWPORT9 *pViewport,
    CONST D3DXMATRIX *pProjection,
    CONST D3DXMATRIX *pView,
    CONST D3DXMATRIX *pWorld
);
```

D3DXVECTOR3 *pOut

Address to receive a translated 3D vector.

CONST D3DXVECTOR3 *pV

Pointer to the coordinate in screen space. This will be the position of your mouse.

CONST D3DVIEWPORT9 *pViewport

Pointer to the Direct3D device's viewport. This can be returned by calling the GetViewport method of IDirect3DDevice9. The viewport structure looks as follows:

```
typedef struct _D3DVIEWPORT9 {
    DWORD X;
    DWORD Y;
    DWORD Width;
    DWORD Height;
    float MinZ;
    float MaxZ;
} D3DVIEWPORT9;
```

CONST D3DXMATRIX *pProjection

Pointer to the projection matrix currently being used by the Direct3D device.

CONST D3DXMATRIX *pView

Pointer to the view matrix currently being used by the Direct3D device.

CONST D3DXMATRIX *pWorld

Pointer to the world matrix currently being used by the Direct3D device.

To use this function to translate the mouse cursor into a ray that extends into 3D space you'll need to call this function twice. Why? Because first you'll notice this function requires that the mouse coordinates in screen space be a 3D vector, not 2D. In short, what you need to do is call the function twice; first you pass $Z = 0$ as the first 2D coordinate, then you pass $Z = 1$. Calling the function twice will give you two returned vectors; the first represents the origin of the ray, and the second will be used to calculate the ray's direction. Here's some code:

```
D3DXVECTOR Result1;
D3DXVECTOR Result2;
D3DXVECTOR VecDir;
```

```

D3DXVec3Unproject(&Result1, &D3DXVECTOR3(X,Y, 0), &Viewport,
                 &Projection, &View, &World);

D3DXVec3Unproject(&Result2, &D3DXVECTOR3(X,Y, 0), &Viewport,
                 &Projection, &View, &World);

VecDir = Result2 - Result1;
D3DXVec3Normalize(VecDir);

```

Q. I'm tired of having to use the old Windows API to create windows and do lots of work just to get a DirectX application running. Is there any way some of this toil can be avoided?

A. Yes, you can use the DirectX sample framework. Take a look in the DirectX SDK for more information. With the sample framework you can create windows and devices, and check mouse clicks and keyboard presses in simple functions. Here's a list of the sample framework functions:

```

DXUTCreateDevice
DXUTCreateDeviceFromSettings
DXUTCreateWindow
DXUTFindValidDeviceSettings
DXUTGetBackBufferSurfaceDesc
DXUTGetD3DDevice
DXUTGetD3DObject
DXUTGetDeviceCaps
DXUTGetDeviceSettings
DXUTGetDeviceStats
DXUTGetElapsedTime
DXUTGetExitCode
DXUTGetFPS
DXUTGetFrameStats
DXUTGetHWND
DXUTGetHWNDDDeviceFullScreen
DXUTGetHWNDDDeviceWindowed
DXUTGetHWNDFocus
DXUTGetPresentParameters
DXUTGetShowSettingsDialog

```

DXUTGetTime
DXUTGetWindowClientRect
DXUTGetWindowTitle
DXUTInit
DXUTIsKeyDown
DXUTIsMouseButtonDown
DXUTIsRenderingPaused
DXUTIsTimePaused
DXUTIsWindowed
DXUTKillTimer
DXUTMainLoop
DXUTPause
DXUTRender3DEnvironment
DXUTResetFrameworkState
DXUTSetCallbackDeviceCreated
DXUTSetCallbackDeviceDestroyed
DXUTSetCallbackDeviceLost
DXUTSetCallbackDeviceReset
DXUTSetCallbackFrameMove
DXUTSetCallbackFrameRender
DXUTSetCallbackKeyboard
DXUTSetCallbackMouse
DXUTSetCallbackMsgProc
DXUTSetConstantFrameTime
DXUTSetCursorSettings
DXUTSetDevice
DXUTSetMultimonSettings
DXUTSetShowSettingsDialog
DXUTSetTimer
DXUTSetWindow
DXUTShutdown
DXUTStaticWndProc
DXUTToggleFullscreen
DXUTToggleREF

This page intentionally left blank.



Appendix B

Recommended Reading

3ds Max 6 Bible. Kelly L. Murdock. Indianapolis, Indiana: Wiley Publishing, Inc., 2004. (ISBN: 0764557637)

Advanced 3D Game Programming with DirectX 9.0. Peter Walsh. Plano, Texas: Wordware Publishing, Inc., 2003. (ISBN: 1556229682)

Essential LightWave 3D 8. Timothy Albee and Steve Warner with Robin Wood. Plano, Texas: Wordware Publishing, Inc., 2005. (ISBN: 1556220820)

Introduction to 3D Game Programming with DirectX 9.0. Frank D. Luna. Plano, Texas: Wordware Publishing, Inc., 2003. (ISBN: 1556229135)

Learning Maya 6: Modeling. Alias. Alameda, California: Sybex International, 2004. (ISBN: 1894893719)

Programming Game AI by Example. Mat Buckland. Plano, Texas: Wordware Publishing, Inc., 2004. (ISBN: 1556220782)

This page intentionally left blank.

Index

1D coordinate system, 52-53
2D coordinate system, 53-56
2D texturing, 145-146
3D coordinate system, 56-57
3D Studio MAX, 189

A

AddDataObject, 185
AddFilter, 288
AddSourceFilter, 289
alpha blending, 144-145
alpha channel, 144
ambient color, 125
ambient lighting, 122
animated textures, 270-271
animation, 237, 316-317
 hierarchical, 242-243
 keyframe, 240-242
 loading, 319
 playing, 325-326
 skeletal, 295-296
animation set, 317-318
anisotropic texture filtering, 139

B

back buffer, 32, 34
 presenting images with, 44-46
base video renderer,
 creating, 276-277
 implementing, 277
Begin, 147
BeginScene, 34
bone hierarchy, 297-299, 302-306
 loading, 306-310
 mapping, 310-311
bounding boxes, 204-206

bounding spheres, 204-206
bump map, 131

C

camera,
 creating, 222-228
 first-person, 217
 moving, 221, 223-224
 rotating, 224-225
CheckMediaType, 278-279
child objects, 175-176
clamp texture addressing, 142
Clear, 32-34
clipping planes, 229-230
CloneMeshFVF, 203-204
collision detection, 83-84, 204
coordinate systems, 52-57
CreateDataObject, 184-185
CreateDevice, 21-25
CreateEnumObject, 173-174
CreateIndexBuffer, 113-114
CreateOffscreenPlainSurface, 37-38
CreateSaveObject, 182-183
CreateStateBlock, 336-337
CreateTexture, 132-134
CreateVertexBuffer, 95-97
cube, testing for, 232-233
cursor, 27, 338
custom video renderer, 288
custom X file templates, 169-170

D

D3DCAPS, 333-335
D3DCOLOR, 122, 126
D3DCOLORVALUE, 124
D3DFORMAT, 23

- D3DLIGHT9, 127-129
- D3DMATERIAL9, 124
- D3DPOOL, 96, 134
- D3DPRESENT_PARAMETERS, 23-25, 47
- D3DPRIMITIVETYPE, 107
- D3DRS_ALPHABLENDENABLE, 145
- D3DVIEWPORT9, 339
- D3DXCOLOR, 33
- D3DXComputeBoundingBox, 206-207
- D3DXComputeBoundingSphere, 208
- D3DXCreateSprite, 146
- D3DXCreateTextureFromFile, 134
- D3DXCreateTextureFromFileInMemory, 134
- D3DXCreateTextureFromFileInMemoryEx, 134
- D3DXCreateTextureFromResource, 134
- D3DXCreateTextureFromResourceEx, 134
- D3DXFileCreate, 170
- D3DXFRAME, 248, 299, 302-303
- D3DXFrameAppendChild, 248
- D3DXFrameFind, 248
- D3DXGetFVFVertexSize, 202
- D3DXGetImageInfoFromFile, 39
- D3DXGetImageInfoFromFileInMemory, 39
- D3DXGetImageInfoFromResource, 39
- D3DXIMAGE_INFO, 39
- D3DXIntersect, 209-211
- D3DXLoadMeshFromX, 191-193
- D3DXLoadMeshFromXof, 193-194
- D3DXLoadSkinMeshFromXof, 299-301
- D3DXLoadSurfaceFromFile, 40-41
- D3DXLoadSurfaceFromFileInMemory, 41
- D3DXLoadSurfaceFromMemory, 41
- D3DXLoadSurfaceFromResource, 41
- D3DXLoadSurfaceFromSurface, 41
- D3DXMATERIAL, 196-197
- D3DXMATRIX, 71
- D3DXMatrixIdentity, 74
- D3DXMatrixInverse, 75
- D3DXMatrixLookAtLH, 102-103, 216, 225
- D3DXMatrixMultiply, 73, 83
- D3DXMatrixPerspectiveFovLH, 103-104
- D3DXMatrixRotationAxis, 80-81
- D3DXMatrixRotationX, 80
- D3DXMatrixRotationY, 80
- D3DXMatrixRotationZ, 80
- D3DXMatrixScaling, 81-82
- D3DXMatrixTransformation2D, 148-149, 270-271
- D3DXMatrixTranslation, 77
- D3DXPLANE, 84
- D3DXPlaneDotCoord, 86-87
- D3DXPlaneFromPointNormal, 85-86
- D3DXPlaneFromPoints, 84-85
- D3DXPlaneIntersectLine, 87-88
- D3DXToDegree, 79
- D3DXToRadian, 79
- D3DXVec3Add, 58, 64-65
- D3DXVec3Cross, 69
- D3DXVec3Dot, 67-68
- D3DXVec3Length, 63-64
- D3DXVec3Lerp, 213
- D3DXVec3Normalize, 67
- D3DXVec3Subtract, 66
- D3DXVec3Unproject, 338-339
- D3DXVECTOR2, 54
- D3DXVECTOR3, 57
- degrees, 79
- device,
 - creating, 21-26
 - querying capabilities of, 27
- diffuse color, 125
- direct lighting, 122, 123, 126-127
- Direct3D, 4
 - creating device, 20-26
 - creating object, 18-19
 - creating program, 16-17
- Direct3DCreate9, 19
- DirectInput, 4
- directional lights, 127
- DirectPlay, 4

- DirectShow, 4, 261-262
 - base classes, 275-276
 - building filter graph, 262
- DirectSound, 4
- DirectX, 4-5
 - configuring, 6-7
 - directories, 7-9
 - installing, 5-6
 - utilities, 9-13
- DoRenderSample, 282-284
- Draw, 149-150
- DrawIndexedPrimitive, 116-117
- DrawPrimitive, 107
- DrawSubset, 200

E

- emissive color, 125
- EndScene, 34
- enumeration, 174-178
- events, configuring, 264-268

F

- filter graph, 261, 263
 - creating, 262
 - preparing, 287-293
- first-person camera, 217
- flat shading, 123
- flexible vertex format, 93-94
 - and meshes, 202-203
 - setting, 106
- frame hierarchy, 297, 325
- frames, 248
- front buffer, 32
- FVF, *see* flexible vertex format

G

- game loop, 29-30
- GDI, 31, 146, 337
- geometric transformations, 57-61, 76
- GetBackBuffer, 44-45
- GetChild, 175
- GetChildren, 174-175
- GetDeviceCaps, 27, 333-335
- GetEvent, 266-267
- GetFVF, 202

- GetName, 178
- GetType, 179
- GetUnconnectedPin, 290-291
- GetVertexBuffer, 201
- Gouraud shading, 123
- graphics, drawing, 92-93

H

- hierarchies
 - bone, 297-299, 302-311
 - frame, 297, 325
 - object, 245-248
 - X file objects, 152, 185

I

- ID3DXSprite, 146-147, 270
- images,
 - loading onto surfaces, 40-42
 - presenting from back buffer, 44-46
- IMediaControl, 263, 268, 291
- IMediaEventEx, 263, 266
- index buffer, 112, 116, 135
 - setting up, 113-115
- indexed primitives, 111-112
 - drawing, 115-116

K

- keyframe, 212, 315-316
- keyframe animation, 240-242

L

- lights, 120-123
 - types, 126-127
- line lists, 110
- line strips, 54-55, 110
- linear texture filtering, 139
- lines, 54
 - intersection with planes, 87
- linked lists, 243-245
- LoadFromFile, 319-320
- Lock, 97-98, 181-182
- LockRect, 284-285
- lost device, 46-48

M

materials, 124-125
 matrix (matrices), 70-71, 99-100
 addition, 72
 components, 71
 concatenation, 73, 82
 identity, 74-75
 inverse, 75-76
 multiplication, 72, 73
 rotation, 78-81
 scaling, 81-82
 structure, *see* D3DXMATRIX
 subtraction, 72
 translation, 76-77
 media files, playing, 261-268
 memcpy, 98-99
 meshes, 188, 196-199
 and FVF, 202-203
 and rays, 209-210
 creating, 188-189
 exporting, 189-190
 loading from X files, 191, 193-194
 rendering, 199
 testing, 190
 testing for, 234-235
 updating, 311-313
 message loop, configuring, 28
 mipmap texture filtering, 139
 mirror texture addressing, 141
 mouse coordinates, projecting, 338
 movie files, 274

N

normal, 85
 nearest-point sampling, 138

O

object hierarchy, 245-248
 objects, enumerating, 174-178

P

particle systems, 249-250, 254-256
 creating, 256-260
 pitch, 218

planes, 83-84

 classifying points, 86
 creating from point and normal,
 85-86
 creating from three points, 84-85
 intersection with lines, 87

point, testing for, 232

point lights, 126

point lists, 110

point sprites, 250-251
 creating, 251-252
 rendering, 253-254

Present, 34-35

primitives, 57, 93
 drawing, 106-107, 109-111
 indexed, 111-113, 115-116

ProcessItems, 320-322

ProcessKeyFrames, 323-325

ProcessObject, 306-309

projection matrix, 100, 103-104

R

radians, 59, 78-79
 ray intersection, 209-210
 RegisterTemplates, 171-172
 render loop, *see* game loop
 RenderFile, 263-264
 roll, 219
 rotation, 58-59, 78, 218, 220

S

scaling, 60-61, 66, 81
 scene, presenting, 32-34
 SetCursorPosition, 338
 SetCursorProperties, 338
 SetFVF, 106
 SetIndices, 116
 SetLight, 130
 SetMaterial, 125
 SetMediaType, 279-281
 SetNotifyWindow, 264-265
 SetRenderState, 121, 145, 253
 SetSamplerState, 139-140, 142-143
 SetStreamSource, 105, 115
 SetTexture, 137-138

- SetTransform, 101-102, 147
- shading modes, 123-124
- skeletal animation, 295-296
- skeleton, animating, 314-318
- skinned mesh, 296
 - loading, 299
- specular color, 125
- sphere, testing for, 234
- spotlights, 127
- state blocks, 336
- stream, 105
- surfaces, 31, 37
 - back buffer, *see* back buffer
 - copying, 42-44
 - creating, 37-40
 - loading from file, 40-42
 - rendering, 31-32

T

- TestCooperativeLevel, 47
- texture addressing modes, 141-143
- texture coordinates, 136
- texture filtering, 138-139
- texture mapping, 135-137
- textures, 131
 - animating, 270
 - creating, 132-134
 - loading from file, 134-135
 - transforming, 270
- time, 238-239
- transform matrix, 100-101
- transformations,
 - combining, 82-83
 - geometric, 57-61
- translation, 58
- triangle fans, 56, 111
- triangle strips, 55-56, 110
- triangles, 55

U

- unit vector, 67
- Update, 326-327
- UpdateSkinnedMesh, 313
- UpdateSurface, 42-43

V

- vectors, 61-62
 - addition, 64
 - cross product, 68-69
 - dot product, 67-68
 - length (magnitude), 63
 - normalization, 67
 - scaling (multiplication), 66
 - subtraction, 65-66
- vertex (vertices), 53, 93
 - using, 94-95
- vertex buffer, 95, 201
 - creating, 95-97
 - locking, 97-99
- vertex interpolation, 212-213
- video, playing, 269, 274-275
- view matrix, 100, 102-103
- viewing frustum, 229-230

W

- window, creating, 17-18
- wrap texture addressing, 141

X

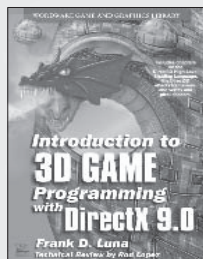
- X file templates, 154-156
 - custom, 169-170
 - registering, 171-172
 - standard, 159-169
- X files, 151-152
 - enumerating, 174-177
 - extracting information from, 178-182
 - loading meshes from, 191, 193-194
 - object hierarchy, 152, 185
 - opening, 173-174
 - reading, 170-171
 - saving data to, 182-186
 - structure, 154-158

Y

- yaw, 219

Looking for more?

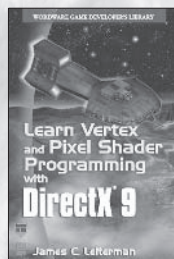
Check these and other titles from
Wordware's complete list.



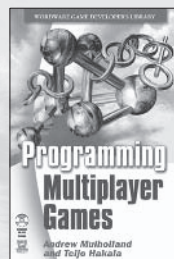
Introduction to 3D Game Programming with DirectX 9.0
1-55622-913-5 • \$49.95
6 x 9 • 424 pp.



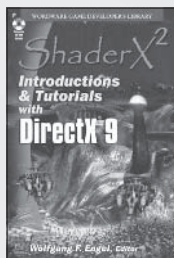
Advanced 3D Game Programming with DirectX 9.0
1-55622-968-2 • \$59.95
6 x 9 • 552 pp.



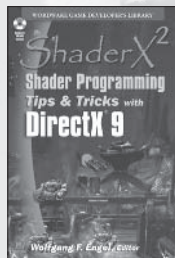
Learn Vertex and Pixel Shader Programming with DirectX 9
1-55622-287-4 • \$34.95
6 x 9 • 304 pp.



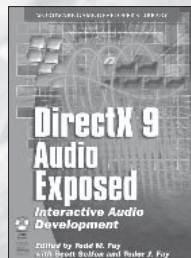
Programming Multiplayer Games
1-55622-076-6 • \$59.95
6 x 9 • 576 pp.



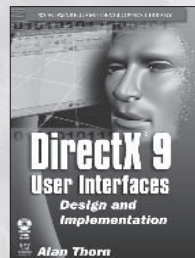
ShaderX2: Introductions & Tutorials with DirectX 9
1-55622-902-X • \$44.95
6 x 9 • 384 pp.



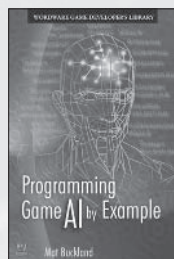
ShaderX2: Shader Programming Tips & Tricks with DirectX 9
1-55622-988-7 • \$59.95
6 x 9 • 728 pp.



DirectX 9 Audio Exposed
1-55622-288-2 • \$59.95
6 x 9 • 568 pp.



DirectX 9 User Interfaces
1-55622-249-1 • \$44.95
6 x 9 • 376 pp.



Programming Game AI by Example
1-55622-078-2 • \$49.95
6 x 9 • 520 pp.



Wireless Game Development in Java with MIDP 2.0
1-55622-998-4 • \$39.95
6 x 9 • 360 pp.



Unlocking Visual C# Programming Secrets
1-55622-097-9 • \$24.95
6 x 9 • 376 pp.

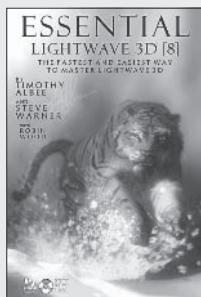


Official Butterfly.net Game Developer's Guide
1-55622-044-8 • \$49.95
6 x 9 • 424 pp.

Visit us online at **www.wordware.com** for more information.
Use the following coupon code for online specials: **dxgraph2297**

Looking for more?

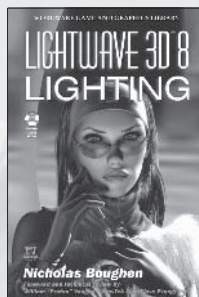
Check out Wordware's market-leading Graphics and Game Programming Libraries featuring the following new releases, backlist, and upcoming titles.



Essential LightWave 3D 8
1-55622-082-0 • \$44.95
6 x 9 • 624 pp.



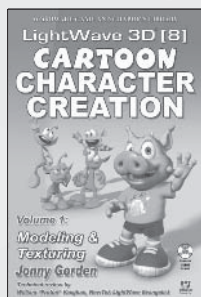
LightWave 3D 7.5 Lighting
1-55622-354-4 • \$69.95
6 x 9 • 496 pp.



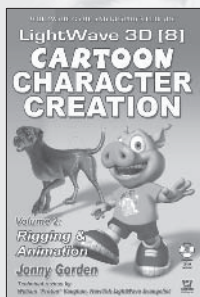
LightWave 3D 8 Lighting
1-55622-094-4 • \$54.95
6 x 9 • 536 pp.



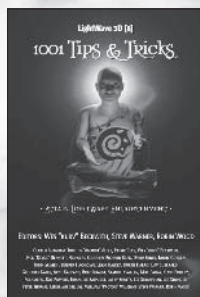
LightWave 3D 8 Texturing
1-55622-285-8 • \$49.95
6 x 9 • 504 pp.



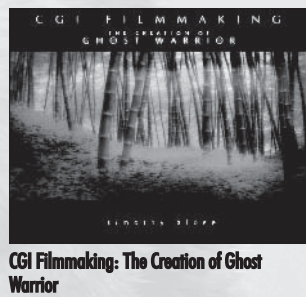
LightWave 3D 8 Cartoon Character Creation: Volume 1 Modeling & Texturing
1-55622-253-X • \$49.95
6 x 9 • 496 pp.



LightWave 3D 8 Cartoon Character Creation: Volume 2 Rigging & Animation
1-55622-254-8 • \$49.95
6 x 9 • 440 pp.



LightWave 3D 8: 1001 Tips and Tricks
1-55622-090-1 • \$39.95
6 x 9 • 648 pp.



CGI Filmmaking: The Creation of Ghost Warrior
1-55622-227-0 • \$49.95
9 x 7 • 344 pp.



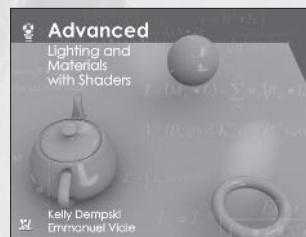
LightWave 3D 8 Character Animation
1-55622-099-5 • \$49.95
6 x 9 • 496 pp.



3ds max Lighting
1-55622-401-X
\$49.95
6 x 9 • 432 pp.



Game Design Theory and Practice 2nd Ed.
1-55622-912-7 • \$49.95
6 x 9 • 728 pp.



Advanced Lighting and Materials with Shaders
1-55622-292-0 • \$44.95
9 x 7 • 360 pp.

Visit us online at www.wordware.com for more information.
Use the following coupon code for online specials: **dxgraph2297**