

[\[Team LiB \]](#)

NEXT ►



- [Table of Contents](#)
- [Index](#)
- [Examples](#)

Managed DirectX® 9 Kick Start: Graphics and Game Programming

By [Tom Miller](#)

Start Reading ►

Publisher: Sams Publishing

Pub Date: October 22, 2003

ISBN: 0-672-32596-9

Pages: 432

Managed DirectX was released with the latest version of the core DirectX libraries in DirectX9. It enables developers using the new .NET languages (i.e. C#, VB.NET, etc.) to develop rich multimedia applications with DirectX. Unfortunately the Managed DirectX runtime was released without adequate documentation, and developers are having a hard time figuring out the best way to write managed applications. This book covers how to use the Managed DirectX objects, how they differ from the core DirectX libraries, and how to create these rich multimedia applications in C#. It also covers in depth graphics techniques and the new high-level shader language shipping with DirectX9.

[\[Team LiB \]](#)

NEXT ►



- [Table of Contents](#)
- [Index](#)
- [Examples](#)

Managed DirectX® 9 Kick Start: Graphics and Game Programming

By [Tom Miller](#)

Start Reading ▶

Publisher: Sams Publishing

Pub Date: October 22, 2003

ISBN: 0-672-32596-9

Pages: 432

[Copyright](#)

[Foreword](#)

[About the Author](#)

[Acknowledgments](#)

[We Want to Hear from You!](#)

[Introduction](#)

[The History of Managed DirectX](#)

[Namespaces Included](#)

[Getting Started](#)

[Part I. Beginning Graphics Concepts](#)

[Chapter 1. Introducing Direct3D](#)

[Getting Started](#)

[The Direct3D Device](#)

[Making Our Triangle Three Dimensional](#)

[Automatic Device Resets During a Resize](#)

[We've Got Camera and Action; What About Lights?](#)

[Device States and Transforms](#)

[Swapchains and RenderTargets](#)

[In Brief](#)

[Chapter 2. Choosing the Correct Device](#)

[Enumerating Your System's Adapters](#)

[Determining Whether a Hardware Device Is Available](#)

[Checking Device Capabilities](#)

[In Brief](#)

[Chapter 3. Rendering Using Simple Techniques](#)

[Using Vertex Buffers](#)

[Texturing Our Objects](#)

[In Brief](#)

[Chapter 4. More Rendering Techniques](#)

[Rendering Other Primitive Types](#)

[Using Index Buffers](#)

[Using Depth Buffers](#)

[In Brief](#)

[Chapter 5. Rendering with Meshes](#)

[Defining the Mesh](#)

[Using Materials and Lighting](#)

[Using Meshes to Render Complex Models](#)

[In Brief](#)

[Chapter 6. Using Managed DirectX to Write a Game](#)

[Choosing the Game](#)

[Writing the Game](#)

[Adding a Movable Car into Your Scene](#)

[Adding Obstacles](#)

[Implementing the Finishing Touches](#)

[In Brief](#)

[Part II. Intermediate Graphics Concepts](#)

[Chapter 7. Using Advanced Mesh Features](#)

[Cloning Mesh Data](#)

[Optimizing Mesh Data](#)

[Simplifying Existing Meshes](#)

[Welding Vertices in a Mesh](#)

[Making Lots of Little Meshes Out of One Big One](#)

[In Brief](#)

[Chapter 8. Understanding Resources](#)

[Starting with the Resource Class](#)

[Using the Vertex and Index Buffers](#)

[Locking Our Buffers](#)

[Controlling How Buffers Are Locked](#)

[Using Texture Resources](#)

[Locking Textures and Getting Descriptions](#)

[In Brief](#)

[Chapter 9. Using the Other Mesh Types](#)

[Simplifying Meshes](#)

[Controlling the Level of Detail with Progressive Meshes](#)

[Rendering Patch Meshes](#)

[Seeing the Tessellation Levels](#)

[In Brief](#)

[Chapter 10. Using the Helper Classes](#)

[Drawing Lines](#)

[Drawing Text](#)

[Rendering to Surfaces](#)

[Rendering Environment Maps](#)

[In Brief](#)

[Part III. Advanced Graphics Concepts](#)

[Chapter 11. Introducing the Programmable Pipeline with the High Level Shader Language](#)

[Rendering a Single Triangle Without Using the Fixed-function Pipeline](#)

[Rendering Shader Programs with Techniques](#)

[Rendering Meshes Using the Programmable Pipeline](#)

[Using HLSL to Write a Pixel Shader](#)

[In Brief](#)

[Chapter 12. Using the High Level Shader Language](#)

[Using Simple Formulas to Simulate Animation](#)

[Determining Color by Blending Textures](#)

[Lighting Textures](#)

[Adding Specular Highlights](#)

[In Brief](#)

[Chapter 13. Rendering Skeletal Animation](#)

[Creating the Frame Hierarchy](#)

[Loading Meshes with Animation](#)

[Rendering Animated Meshes](#)

[In Brief](#)

[Part IV. Sound and Input](#)

[Chapter 14. Discovering the Wonders of Sound](#)

[Including the Sound Namespace](#)

[Loading and Playing a Static Sound](#)

[Using Sounds in 3D](#)

[Manipulating the Listener](#)

[Using Effects with Your Sounds](#)

[In Brief](#)

[Chapter 15. Controlling User Input](#)

[Detecting the Devices You Can Use](#)

[Using the Keyboard Device](#)

[Using the Mouse Device](#)

[Using Game Pads and Joysticks for User Input](#)

[Using Force Feedback](#)

[In Brief](#)

[Part V. 2D Graphics](#)

[Chapter 16. Using Direct3D for 2D Graphics](#)

[Creating a Full Screen Rendering Device](#)

[Rendering Sprites](#)

[Animating Your Sprites](#)

[In Brief](#)

[Chapter 17. Using DirectDraw for 2D Rendering](#)

[Creating a Full Screen DirectDraw Device](#)

[Animating Your Sprites](#)

[In Brief](#)

[Part VI. Adding Networking](#)

[Chapter 18. Implementing Peer-to-Peer Networking Using DirectPlay](#)

[Understanding DirectPlay Addresses](#)

[Creating a Peer Connection](#)

[Getting into a Session](#)

[Using the Event Model](#)

[Performing Actions in Our Session](#)

[Handling Lost Sessions](#)

[In Brief](#)

[Chapter 19. Creating a Client/Server Session](#)

[Creating a Dedicated Server Session](#)

[Connecting a Client](#)

[Detecting Users Joining and Leaving Your Session](#)

[Sending Data Packets](#)

[Making the Client React](#)

[Handling the Server Lost Event](#)

[In Brief](#)

[Chapter 20. Understanding Advanced Networking Features](#)

[Having a Detailed Look at the Event Model](#)

[Detecting Bandwidth and Network Statistics](#)

[Launching Applications Using a Lobby](#)

[Making Your Application Lobby Aware](#)

[Adding Voice Chat to Your Sessions](#)

[In Brief](#)

[Chapter 21. Achieving Maximum Performance](#)

[Using Value Types as Objects](#)

[Understanding the Performance Implications of the Event Model](#)

[Understanding the Cost of Methods](#)

[In Brief](#)

[Part VII. Appendices](#)

[Appendix A. Using the Diagnostics Assemblies](#)

[Enumerating Everything in the System](#)

[Enumerating Specific Items You Care About](#)

[Appendix B. Playing Music and Videos](#)

[Playing Back an Audio or Video File Simply](#)

[Using the Features of Video Files](#)

[Using Video as Textures](#)

[Index](#)

[\[Team LiB \]](#)

[◀ PREVIOUS](#) [NEXT ▶](#)

[\[Team LiB\]](#)

Copyright

Copyright © 2004 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Library of Congress Catalog Card Number: 2003111747

Printed in the United States of America

First Printing: October 2003

06 05 04 03 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

1-317-428-3341

international@pearsontechgroup.com

Credits

Associate Publisher

Michael Stephens

[\[Team LiB\]](#)

Foreword

Since the release of the Microsoft DirectX 9.0 SDK, which included the initial release of DirectX for Managed Code, developers have been demanding a book dedicated to this technology. There have been a few attempts thus far, but none that will match the level of knowledge and expertise you will find in this book.

How can I say this? Easy! During the development of DirectX for Managed Code, I was the development lead responsible for the DirectX SDK. Included in this role was the management of the massive development team responsible for making DirectX for Managed Code a reality. Okay, so it was two people. Regardless, the one guy primarily responsible for the effort is the author of this book.

As Tom's manager I witnessed his work and dedication first hand. I can tell you, if there's one person who has depth of knowledge with managed code, DirectX, and the .NET Framework, it's Tom. The information you'll find in this book comes from the source responsible for the development of DirectX for Managed Code API itself. You won't find a more knowledgeable person in this area.

While this book covers every component of DirectX for Managed Code, the majority of it focuses on the most used API, Direct3D. It will cover everything from networking, audio, and input devices to progressive meshes, shaders, and character animation. If you're looking for the ultimate programmer's guide for developing DirectX for Managed Code applications, there's no need to look any further.

—Bob Gaines

Microsoft Corporation—Lead Program Manager

[\[Team LiB \]](#)

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

About the Author

Tom Miller is the development lead for Managed DirectX, as well as the designer of the API. He has worked on the DirectX team for the last four years, including the SDK team writing samples, as well as the DirectX for Visual Basic team. Previously at Microsoft, he worked on the Visual Basic and Office teams.

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Acknowledgments

To Bob Gaines, my manager while Managed DirectX was under development, who always had the encouragement I needed.

To the rest of the Managed DirectX team: Adam Lonnberg, Dominic Riccetti, Randy Russel, Rich Sauer, and Rick Hoskinson. The hard work everyone put into the project really paid off.

To Phil Taylor, the program manager for Managed DirectX, who got me out of many meetings and allowed me to continue working.

To Jason Kepner, who helped with the review of the book, and did most of the original artwork for the book.

To the beta testers of DirectX9, whose invaluable feedback helped shape the look and feel of the final Managed DirectX API.

To the management in DirectX who believed in me enough to allow my vision of Managed DirectX to be realized.

To my mom Vicki, who raised me and gave me the work ethic that works so well for me now.

To my sister Jessica. She'd be upset if I didn't mention her, especially since I mentioned Mom.

And last, but certainly not least:

To my wife Tanya, my son Lucas, and my daughter Samantha. Thank you for encouraging me to write this book, and dealing with the long periods of time when I was too "busy working" to be much fun. I couldn't have done this without your support.

I've run out of people to thank, so I'll just thank everyone else for reading this book.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@sampublishing.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

For more information about this book or another Sams Publishing title, visit our Web site at www.sampublishing.com. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

Introduction

DirectX is a rich multimedia API that gives developers the ability to write high-performance applications using a standardized set of interfaces. The first release of DirectX coincided with Windows 95, and was called the "Games SDK." Since then, more than eight new versions of the API have been released; however, these APIs were designed for developers who used C and C++. There was an entire group of developers who had no easy access to the functionality DirectX provided.

[\[Team LiB\]](#)

The History of Managed DirectX

The first public release of Managed DirectX happened on December 20, 2002, with the release of DirectX 9; however, the project had been in development for quite some time. Late in the development cycle for DirectX 8.1, while I was working on DirectX for Visual Basic, I started playing with the first beta of Visual Studio.NET and the .NET runtime, and it was obvious to me that this was the future of development. I began working on a prototype of "DirectX.NET," which would eventually become Managed DirectX, during this time.

From the beginning, Managed DirectX was designed to be just as powerful as the "core" API, with little to no overhead. In the past, Visual Basic users of DirectX lacked the ability to develop applications of similar quality when compared to the core DirectX API. Some of this was due to the Visual Basic runtime itself, while some was due to the overhead of the wrapper DLLs. In DirectX 8, we tried to eliminate some of these performance bottlenecks by removing the wrapping layer for the most performance critical API, Direct3D. Instead of having a proxy DLL that did data marshalling, we moved the Direct3D API directly into a type library. While this did improve performance slightly, in many cases all it did was overly complicate the API, particularly for the Visual Basic developers we were targeting.

After the release of DirectX 8, it was apparent that the API was not very easy to use. The samples were hard to follow and the code didn't look like "normal" Visual Basic applications. Most Visual Basic developers found the API too complicated to use; C++ developers had no reason to switch since there was no benefit in it for them. We were popular with the home enthusiasts, but game studios weren't even aware we existed. Our new API needed to address these issues. It needed to be fast, it needed to be easy to use, and it needed to have a reason for developers to switch from the C++ code they had been writing. The concept for Managed DirectX was born; now we just needed to implement it.

At the Game Developers Conference (GDC) in San Jose in 2002, the first alpha of Managed DirectX was released. It was based on the DirectX 8.1 runtime, and written exclusively in C#. We had most of the core DirectX components in this release, including DirectMusic, which was excluded in the final Managed DirectX release. The only DirectX components not represented in this first alpha were DirectDraw and DirectShow, mainly due to the fact that we hadn't decided whether or not we really would include these components or not. Looking back at that first release, it still amazes me how far it's come. I hadn't really addressed any of the problems that the Visual Basic DLLs exposed yet; all I had really done was expose the DirectX API to C#. There was a one-to-one mapping of every function, constant, and structure in DirectX.

This release was immensely helpful to us. It demonstrated the feasibility of the Managed DirectX runtime, and the feedback we got was surprisingly positive. One of the first things this release showed us was that the performance of this layer was lacking immensely. In best-case scenarios, we could get performance matching the native API, but in even a simple scenario (such as the Cube Map sample), the managed version would run at about 40% of the frame rate that the C++ version did. We had our work cut out for us.

We spent the next few months getting ready for our next release, the first beta of DirectX 9. We addressed all the major performance bottlenecks, added DirectDraw support, removed DirectMusic support, and had begun our ease-of-use improvements. We also made sure we had developers familiar with C# and managed code on the beta list. They would actually use the API to gauge how much we really had made the API simpler. The feedback we received was quick, and unanimous. Performance was fine; the API design, on the other hand, left much to be desired. Managed DirectX looked nothing like other managed APIs; it looked like a COM API. All of the components in the .NET runtime had a similar look and feel, while the Managed DirectX DLLs were completely different. We had two goals for this project: Make it fast, and make it easy to use. While we had delivered on the former, it was evident we hadn't delivered the latter.

A total redesign of the API commenced directly after Beta1. We talked with the .NET runtime team, we talked to the beta users, we gathered feedback from any place we could think of to find out the things that were wrong with our current design, and went about fixing it. I wrote a set of design guidelines that all components would need to adhere to, and the task of actually making our current API follow these guidelines started. Some of the changes were simple to implement, such as casing changes. In DirectX (and COM in general), constants and structure names were normally described in all caps, in the .NET runtime, things were cased properly. Other changes were more in depth in nature, such as adding enumeration support to classes. We released Beta2 and anxiously awaited the feedback from

[\[Team LiB\]](#)

Namespaces Included

The Managed DirectX API has a relatively obvious set of namespaces, breaking down the functionality into the various "core" components housed in DirectX. There is also the generic "Microsoft.DirectX" namespace that houses the common functionality. The full list of namespaces included in Managed DirectX is listed in [Table 1](#).

Table 1. Managed DirectX Namespaces

Microsoft.DirectX	Parent namespace, holds all common code.
Microsoft.DirectX.Direct3D	Direct3D graphics API, as well as the D3DX helper library.
Microsoft.DirectX.DirectDraw	DirectDraw graphics API.
Microsoft.DirectX.DirectPlay	DirectPlay networking API.
Microsoft.DirectX.DirectSound	DirectSound audio API.
Microsoft.DirectX.DirectInput	DirectInput user input API.
Microsoft.DirectX.AudioVideoPlayback	Simple audio and video playback API.
Microsoft.DirectX.Diagnostics	Simple diagnostics API.
Microsoft.DirectX.Security	Underlying structure for DirectX code access security.
Microsoft.DirectX.Security.Permissions	Permission classes for DirectX code access security.

As you can see, this encompasses most functionality included with DirectX. During the course of this book, we will deal extensively with the Direct3D namespace, but will cover all of the other areas as well.

Getting Started

So now that we've got a little groundwork information, we have enough information to actually get started writing some Managed DirectX applications! Before you begin writing the next big game using Managed DirectX, there are a few things you need to have.

First, you will need a source code editor and the runtime environment. I would recommend Visual Studio.NET 2003, as that is what Microsoft supports. Regardless of what editor you use, you will need version 1.1 of the .NET runtime, which you can install from the included CD (it also comes automatically with Visual Studio.NET 2003).

Next you will need to have the DirectX 9.0 SDK Update developer runtime installed. I would recommend installing the DirectX 9.0 Software Development Kit Update located on the included CD. This will give you the developer runtime, as well as many other samples, and documentation on Managed DirectX.

The graphics samples in the first few chapters should run on just about any modern graphics card. However, later samples will require a more advanced video card, capable of using vertex and pixel shaders. The GeForce3 and above should be sufficient for these samples; however, I would recommend a card capable of using the shader model 2.0 (Radeon 9700 and above).

Before we actually delve into Managed DirectX coding, there are some things it will be assumed you know. If you are new to programming in general, this book probably isn't for you. The book is targeted for developers who already have general programming experience and are looking for information on building rich multimedia applications using Managed DirectX. The code written in the text of this book will be C#, but the included CD will also contain versions of the code written in Visual Basic.NET.

The included CD contains the DirectX 9.0 SDK Update, as well as all of the source code discussed in this book, both in C# and Visual Basic.NET. You will also find the .NET runtime version 1.1 included as well. You can download the DirectX 9 SDK Update or the .NET runtime at <http://msdn.microsoft.com> as well.

With these things out of the way, we are ready to start using Managed DirectX. Now, we just need to figure out how to actually write the code for this next big game.

Part I: Beginning Graphics Concepts

- 1 [Introducing Direct3D](#)
- 2 [Choosing the Correct Device](#)
- 3 [Rendering Using Simple Techniques](#)
- 4 [More Rendering Techniques](#)
- 5 [Rendering with Meshes](#)
- 6 [Using Managed DirectX to Write a Game](#)

Chapter 1. Introducing Direct3D

IN THIS CHAPTER

- [Getting Started](#)
- [The Direct3D Device](#)
- [Making Our Triangle Three Dimensional](#)
- [Automatic Device Resets During a Resize](#)
- [We've Got Camera and Action; What About Lights?](#)
- [Device States and Transforms](#)
- [Swapchains and RenderTargets](#)
- [In Brief](#)

One of the most popular aspects of "rich multimedia" applications would be the addition of 3D graphics. With the processing power of today's modern GPU (graphics processing unit), realistic scenes can be rendered in real time. Have you ever been playing the latest 3D game and wondered, "How do they do that?," only to assume it must be too complicated? Managed Direct3D allows developers an easy and fast way to get complex (or simple) graphics and animations on the screen. In this chapter we will cover the following:

- Creating a default Direct3D device
- Rendering a triangle in both screen coordinates and in world space
- Adding a light to our triangle

Getting Started

Before we begin developing our first Managed DirectX application, we need to get our environment ready. To accomplish this, follow these steps:

1.

The first thing we'll want to do will be to load Visual Studio.NET and create a new project.

2.

Let's select the Visual C# projects area, and create a new Windows Application project. We will need a place to do our rendering, and the standard Windows Form that comes with this project works perfectly.

3.

Name the project whatever you like and create the new project.

After the project is created, we will need to make sure the Managed DirectX references are added to the project so we can use the components. Click the Add References menu selection in the project menu, and add the Microsoft.DirectX as well as the Microsoft.DirectX.Direct3D reference. For now, that's all we'll need.

With the references now loaded into the project, we could get right into writing some Managed DirectX code, but before we do that, we should add two new items into our "using" clauses so we don't need to fully qualify everything we'll be using. You can do this by opening the code window for the main windows form in your application (by default form1.cs) and adding the following lines at the end of the using clauses:

```
using Microsoft.DirectX;  
using Microsoft.DirectX.Direct3D;
```

While this step isn't necessary, it saves a lot of typing, since you won't need to fully qualify each item before using it. Now, we are ready to start writing our first Managed DirectX application.

[\[Team LiB\]](#)

The Direct3D Device

The root of all drawing in Direct3D is the device class. You can think of this class as analogous to the actual graphics device in your computer. A device is the parent of all other graphical objects in your scene. Your computer may have zero or many devices, and with Managed Direct3D, you can control as many of them as you need.

There are three constructors that can be used to create a device. For now, we're only going to use one of them; however we will get to the others in later chapters. The one we're concerned about takes the following arguments:

```
public Device ( System.Int32 adapter , Microsoft.DirectX.Direct3D.DeviceType deviceType ,
    System.Windows.Forms.Control renderWindow , Microsoft.DirectX.Direct3D.CreateFlags
    behaviorFlags, Microsoft.DirectX.Direct3D.PresentationParameters presentationParameters )
```

Now, what do all of these arguments actually mean, and how do we use them? Well, the first argument "adapter" refers to which physical device we want this class to represent. Each device in your computer has a unique adapter identifier (normally 0 through one less than the number of devices you have). Adapter 0 is always the default device.

UNDERSTANDING THE DEVICE CONSTRUCTOR OVERLOADS

The second overload for the device is identical to the first, with the exception of the renderWindow parameter, which takes an IntPtr for an unmanaged (or non-windows form) window handle. The final overload takes a single IntPtr that is the unmanaged COM pointer to the IDirect3DDevice9 interface. This can be used if you have a need to work with an unmanaged application from your managed code.

The next argument, DeviceType, tells Direct3D what type of device you want to create. The most common value you will use here is DeviceType.Hardware, which means you want to use a hardware device. Another option is DeviceType.Reference, which will allow you to use the reference rasterizer, which implements all features of the Direct3D runtime, and runs extremely slowly. You would use this option mainly for debugging purposes, or to test features of your application that your card doesn't support.

The "renderWindow" argument binds a window to this device. Since the windows forms control class contains a window handle, it makes it easy to use a derived class as our rendering window. You can use a form, panel, or any other control-derived class for this parameter. For now, we'll use forms.

USING A SOFTWARE DEVICE

You should note that the reference rasterizer is only distributed with the DirectX SDK, so someone with only the DirectX runtime will not have this feature. The last value is DeviceType.Software, which will allow you to use a custom software rasterizer. Unless you know you have a custom software rasterizer, ignore this option.

The next parameter is used to control aspects of the device's behavior after creation. Most of the members of the CreateFlags enumeration can be combined to allow multiple behaviors to be set at once. Some of these flags are mutually exclusive, though, and I will get into those later. We will only use the SoftwareVertexProcessing flag for now. This flag specifies that we want all vertex processing to happen on the CPU. While this is naturally slower than having the vertex processing happen on the GPU, we don't know for sure whether or not your graphics card supports this feature. It's safe to assume your CPU can do the processing for now.

The last parameter of this constructor controls how the device presents its data to the screen. Every aspect of the

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Making Our Triangle Three Dimensional

Looking back at our application, it doesn't appear to be very three dimensional. All we did was draw a colored triangle inside a window, which could just as easily have been done with GDI. So, how do we actually draw something in 3D and have it look a little more impressive? Actually, it's relatively simple to modify our existing application to accommodate us.

If you remember, earlier when we were first creating the data for our triangle, we used something called transformed coordinates. These coordinates are already known to be in screen space, and are easily defined. What if we had some coordinates that weren't already transformed though? These untransformed coordinates make up the majority of a scene in a modern 3D game.

When we are defining these coordinates, we need to define each vertex in world space, rather than screen space. You can think of world space as an infinite three-dimensional Cartesian space. You can place your objects anywhere in this "world" you want to. Let's modify our application to draw an untransformed triangle now.

We'll first change our triangle data to use one of the untransformed vertex format types; in this case, all we really care about is the position of our vertices, and the color, so we'll choose `CustomVertex.PositionColored`. Change your triangle data code to the following:

```
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[3];
verts[0].SetPosition(new Vector3(0.0f, 1.0f, 1.0f));
verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
verts[1].SetPosition(new Vector3(-1.0f, -1.0f, 1.0f));
verts[1].Color = System.Drawing.Color.Black.ToArgb();
verts[2].SetPosition(new Vector3(1.0f, -1.0f, 1.0f));
verts[2].Color = System.Drawing.Color.Purple.ToArgb();
```

And change your `VertexFormat` property as well:

```
device.VertexFormat = CustomVertex.PositionColored.Format;
```

Now what's this? If you run the application, nothing happens; you're back to your colored screen, and nothing else. Before we figure out why, let's take a moment to describe the changes. As you can see, we've switched our data to use the `PositionColored` structure instead. This structure will hold the vertices in world space as well as the color of each vertex. Since these vertices aren't transformed, we use a `Vector3` class in place of the `Vector4` we used with the transformed classes, because transformed vertices do not have an `rhw` component. The members of the `Vector3` structure map directly to the coordinates in world space; `x`, `y`, and `z`. We also need to make sure `Direct3D` knows we've changed the type of data we are drawing, so we change our fixed function pipeline to use the new untransformed and colored vertices by updating the `VertexFormat` property.

So why isn't anything displayed when we run our application? The problem here is that while we're now drawing our vertices in world space, we haven't given `Direct3D` any information on how it should display them. We need to add a camera to the scene that can define how to view our vertices. In our transformed coordinates, a camera wasn't necessary because `Direct3D` already knew where to place the vertices in screen space.

The camera is controlled via two different transforms that can be set on the device. Each transform is defined as a 4x4 matrix that you can pass in to `Direct3D`. The projection transform is used to define how the scene is projected onto the monitor. One of the easiest ways to generate a projection matrix is to use the `PerspectiveFovLH` function on the `Matrix` class. This will create a perspective projection matrix using the field of view, in a left-handed coordinate system.

What exactly is a left-handed coordinate system anyway, and why does it matter? In most Cartesian 3D coordinate systems, positive `x` coordinates move toward the right, while positive `y` coordinates move up. The only other coordinate left is `z`. In a left-handed coordinate system, positive `z` moves away from you, while in a right-handed coordinate system, positive `z` moves toward you. You can easily remember which coordinate system is which by

[\[Team LiB\]](#)

Automatic Device Resets During a Resize

Anyone who's ever written a Direct3D application in C++ or DirectX for Visual Basic understands that normally, when the window of your application is resized, the device needs to be reset; otherwise, Direct3D will continue to render the scene at the same resolution as when the device was created, and the resulting image will be copied (and stretched to fit) onto the newly sized window. Managed DirectX is smart enough to realize when you've created your device with a Windows Form control, and can automatically reset the device for you when this control is resized. Naturally, you can revert back to the normal behavior and handle device resets yourself quite easily. There is an event attached to the device called "DeviceResizing", which occurs right before the automatic device reset code. By capturing this event, and setting the Cancel member of the EventArgs class passed in to true, you can revert to the default behavior. Add the following function to your sample:

```
private void CancelResize(object sender, CancelEventArgs e)
{
    e.Cancel = true;
}
```

As you can see, this function does nothing more than say yes, we do want to cancel this action. Now, we need to hook up this event handler to our device so it knows not to perform this action. Add the following line of code immediately after the device creation:

```
device.DeviceResizing += new CancelEventHandler(this.CancelResize);
```

Now run the application once more and after it starts, maximize the window. As you can easily see, the triangle, while still there and in the same spot, looks horrible. The edges are jagged, and it's just not very nice looking at all. Go ahead and delete the last two sections of code we added. The default behavior of Managed DirectX handles the device resizing for you, and we might as well take advantage of it.

[\[Team LiB\]](#)

We've Got Camera and Action; What About Lights?

Now that we've got our single triangle drawn and spinning around, what else could make it even better? Why, lights of course! We mentioned lights briefly before, when our triangle turned completely black after we first moved over to our nontransformed triangles. Actually, we completely turned the lighting off in our scene. The first thing we need to do is to turn that back on, so change the lighting member back to true:

```
device.RenderState.Lighting = true;
```

You could just delete the line entirely since the default behavior for the device is to have lighting on; we'll just leave it as true for the sake of clarity. Running the application now, we see we're back to the black triangle again; it's just spinning around now. Maybe we should go ahead and define a light, and turn it on. You'll notice the device class has a lights array attached to it, with each member of the array holding the various light properties. We want to define the first light in the scene and turn it on, so let's add the following code into our OnPaint method just after our triangle definition code:

```
device.Lights[0].Type = LightType.Point;
device.Lights[0].Position = new Vector3();
device.Lights[0].Diffuse = System.Drawing.Color.White;
device.Lights[0].Attenuation0 = 0.2f;
device.Lights[0].Range = 10000.0f;
device.Lights[0].Commit();
device.Lights[0].Enabled = true;
```

So, what exactly do all those lines of code do? First, we define what type of light we want to display. We've picked a point light, which is a light that radiates light in all directions equally, much like a light bulb would. There are also directional lights that travel in a given direction infinitely. You could think of the sun as a directional light (yes, in reality the sun would be a point light since it does give off light in all directions; however, from our perspective on Earth, it behaves more like a directional light). Directional lights are only affected by the direction and color of the light and ignore all other light factors (such as attenuation and range), so they are the least computationally expensive light to use. The last light type would be the spot light, which like its name is used to define a spot light, much like you would see at a concert, highlighting the person on stage. Given the large number of factors in making up the spot light (position, direction, cone angle, and so on), they are the most computationally expensive lights in the system.

With that brief discussion on light types out of the way, we'll continue. Next we want to set the position of our point light source. Since the center of our triangle is at 0, 0, 0, we may as well position our light there as well. The parameterless constructor of Vector3 does this. We set the diffuse component of the light to a white color, so it will light the surface normally. We set the first attenuation property, which governs how the light intensity changes over distance. The range of the light is the maximum distance the light has any effect. Our range far exceeds our needs in this case. You can see the DirectX SDK (included on the CD) for more information on the mathematics of lights.

Finally, we commit this light to the device, and enable it. If you look at the properties of a light, you will notice one of them is a Boolean value called "Deferred". By default, this value is false, and you are therefore required to call Commit on the light before it is ready to be used. Setting this value to true will eliminate the need to call Commit, but does so at a performance penalty. Always make sure your light is enabled and committed before expecting to see any results from it.

Well, if you've run your application once more, you'll notice that even though we've got our light defined in the scene now, the triangle is still black. If our light is on, yet we see no light, Direct3D must not be lighting our triangle, and in actuality, it isn't. Well, why not? Lighting calculations can only be done if each face in your geometry has a normal, which currently we don't have. What exactly is a normal, though? A normal is a perpendicular vector pointing away from the front side of a face; see [Figure 1.5](#).

Figure 1.5. Vertex normals.

[\[Team LiB\]](#)

Device States and Transforms

Two items we've used in our sample code thus far that haven't been delved into are the various device states and transforms. There are three different state variables on the device: the render states, the sampler states, and the texture states. We've only used some of the render states thus far; the latter two are used for texturing. Don't worry; we'll get to texturing soon enough. The render state class can modify how Direct3D will do its rasterization of the scene. There are many different attributes that can be changed with this class, including lighting and culling, which we've used in our application already. Other options you can set with these render states include fill mode (such as wire frame mode) and various fog parameters. We will discuss more of these options in subsequent chapters.

As mentioned before, transformations are matrices used to convert geometry from one coordinate space to another. The three major transformations used on a device are the world, view, and projection transforms; however, there are a few other transforms that can be used. There are transforms that are used to modify texture stages, as well as up to 255 world matrices.

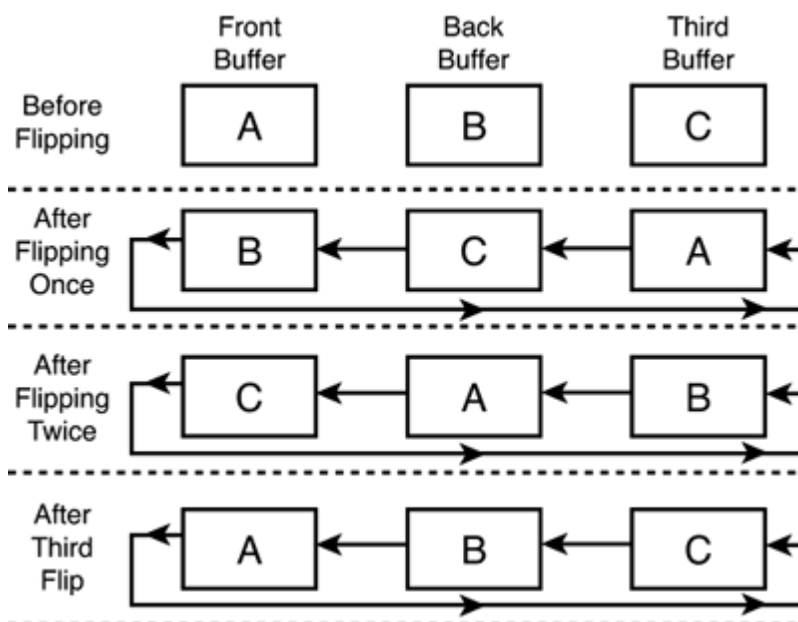
[\[Team LiB\]](#)

Swapchains and RenderTargets

So what exactly is going on with the device that allows it to draw these triangles? Well, there are a few things implicit on a device that handle where and how items are drawn. Each device has an implicit swap chain, as well as a render target.

A swap chain is essentially a series of buffers used to control rendering. There is a back buffer, which is where any drawing on this swap chain occurs. When a swap chain that has been created with `SwapEffect.Flip` is presented, the back buffer data is "flipped" to the front buffer, which is where your graphics card will actually read the data. At the same time, a third buffer becomes the new back buffer, while the previous front buffer moves to the unused third buffer. See [Figure 1.6](#) for details.

Figure 1.6. Back buffer chain during flips.



A true "flip" operation is implemented by changing the location of where the video card will read its data and swapping the old one with the current back buffer location. For DirectX 9, this term is used generically to indicate when a back buffer is being updated as the display. In windowed mode, these flip operations are actually a copy of the data, considering our device isn't controlling the entire display, but instead just a small portion of it. The end result is the same in either case, though. In full screen mode, using `SwapEffect.Flip`, the actual flip occurs; some drivers will also implement `SwapEffect.Discard` or `SwapEffect.Copy` with a flip operation in full screen mode.

If you create a swap chain using `SwapEffect.Copy` or `SwapEffect.Flip`, it is guaranteed that any present call will not affect the back buffer of the swap chain. The runtime will enforce this by creating extra hidden buffers if necessary. It is recommended that you use `SwapEffect.Discard` to avoid this potential penalty. This mode allows the driver to determine the most efficient way to present the back buffer. It is worth noting that when using `SwapEffect.Discard` you will want to ensure that you clear the entire back buffer before starting new drawing operations. The runtime will fill the back buffer with random data in the debug runtime so developers can see if they forget to call clear.

The back buffer of a swap chain can also be used as a render target. Implicitly, when your device is created, there is a swap chain created, and the render target of the device is set to that swap chain's back buffer. A render target is simply a surface that will hold the output of the drawing operations that you perform. If you create multiple swap chains to handle different rendering operations, you will want to ensure that you update the render target of your device beforehand. We will discuss this more in later chapters.

[\[Team LiB\]](#)

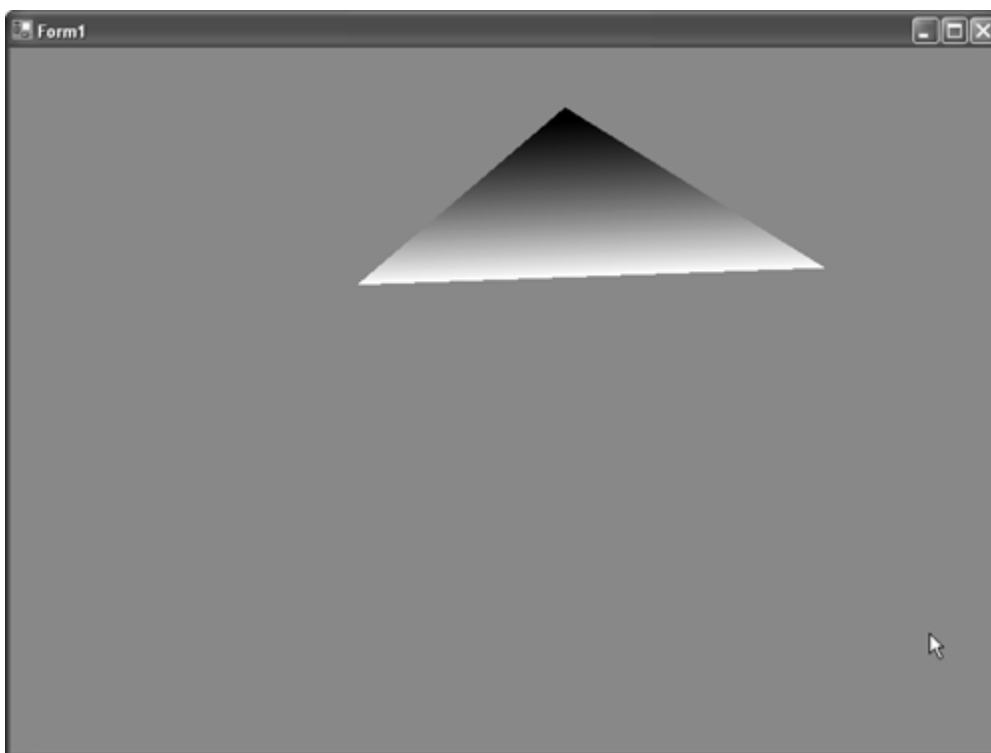
In Brief

We've covered the bare essentials for getting a triangle rendered, and then included code to make it spin around and get it lit inside our scene. In doing so, we've accomplished the following:

- Created a Direct3D device attached to our windows form
- Rendered a single colored triangle in our window
- Rotated our triangle around our scene
- Introduced a simple light to our scene

Compiling the code will show a window that looks much like [Figure 1.7](#).

Figure 1.7. Spinning 3D triangle.



In our next chapter we'll look at how we can find out which options our device can support, and create our devices accordingly.

Chapter 2. Choosing the Correct Device

IN THIS CHAPTER

-

- [Enumerating Your System's Adapters](#)

-

- [Determining Whether a Hardware Device Is Available](#)

-

- [Checking Device Capabilities](#)

-

- [In Brief](#)

The number of possible permutations when creating a device is quite staggering. With all of the varying types of graphics cards available in the market today, knowing which features each of them support is simply not feasible. You will need to query the device itself to find out the features it supports. During this chapter, we will discuss

-

- Enumerating all of the adapters on a system.

-

- Enumerating supported formats for each device.

-

- Capabilities of the devices we are enumerating.

[\[Team LiB\]](#)

Enumerating Your System's Adapters

Many systems today can support multiple monitors per machine. While it hasn't quite hit the "mainstream," having multiple monitors (multimon for short) can be quite useful, and is rapidly becoming more popular. With the higher-end graphics card supporting multimon natively, you can expect this feature to be used more exclusively. The latest cards from ATI, nVidia, and Matrox all have dual head support, which can allow multiple monitors to be attached to a single graphics card.

Direct3D devices are specified per adapter. In this case, you can think of an "adapter" as the actual graphics hardware that connects to the monitor. While the ATI Radeon 9700 is only one "physical" adapter, it has two monitor hookups (DVI and VGA), and thus has two adapters in Direct3D. You may not know which, or even how many, devices are on a system your game will be running on, so how can you detect them and pick the right one?

There is a static class in the Direct3D assemblies called "Manager" that can easily be used to enumerate adapters and device information, as well as retrieving the capabilities of the devices available on the system.

The very first property in the Manager class is the list of adapters in your system. This property can be used in multiple ways. It stores a "count" member that will tell you the total number of adapters on the system. Each adapter can be indexed directly (for example, Manager.Adapters[0]), and it can also be used to enumerate through each of the adapters on your system.

To demonstrate these features, a simple application will be written that will create a tree view of the current adapters on your system and the supported display modes that adapter can use. Load Visual Studio and follow these steps:

1.

Create a new C# Windows Forms Project. You can name it anything you want; the sample code provided was named "Enumeration".

2.

Add a reference to the Microsoft.DirectX.Direct3D and Microsoft.DirectX assemblies. Add using clauses for these assemblies as well.

3.

In design view for the created windows form, add a TreeView control to your application. You will find the TreeView control in the toolbox.

4.

Select the TreeView control on your form, and hit the F4 key (or right-click and choose Properties). In the properties of your TreeView, set the "Dock" parameter to "Fill". This will cause your TreeView to always fill up the entire window, even if it is resized.

Now, you should add a function that will scan through every adapter on the system, and provide a little information on the modes each one can support. The following function will do this work, so add it somewhere in your class:

```
/// <summary>
/// We will fill our tree view here
/// </summary>
public void LoadGraphics()
{
    foreach(AdapterInformation ai in Manager.Adapters)
    {
        TreeNode root = new TreeNode(ai.Information.Description);

        TreeNode driverInfo = new TreeNode(string.Format
            ("Driver information: {0} - {1}",
            ai.Information.DriverName,
            ai.Information.DriverVersion) );
```

[\[Team LiB\]](#)

Determining Whether a Hardware Device Is Available

The manager class has a multitude of functions that can be used to determine the support your adapter has for a particular feature. For example, if you wanted to determine whether or not your device supported a particular format, and didn't want to enumerate all possible adapters and formats, you could use the manager class to make this determination. The following function can be used:

```
public static System.Boolean CheckDeviceType ( System.Int32 adapter ,  
    Microsoft.DirectX.Direct3D.DeviceType checkType ,  
    Microsoft.DirectX.Direct3D.Format displayFormat ,  
    Microsoft.DirectX.Direct3D.Format backBufferFormat ,  
    System.Boolean windowed , System.Int32 result )
```

This can be used to determine quickly whether your device supports the type of format you wish to use. The first parameter is the adapter ordinal you are checking against. The second is the type of device you are checking, but this will invariably be DeviceType.Hardware the majority of the time. Finally, you specify the back buffer and display formats, and whether or not you want to run in windowed or full screen mode. The final parameter is optional, and if used, will return the integer code (HRESULT in COM) of the function. The method will return true if this is a valid device type, and false otherwise.

DETECTING FORMAT CONVERSION

It's important to note that in windowed mode, the back buffer format does not need to match the display mode format if the hardware can support the appropriate color conversion. While the CheckDeviceType method will return appropriate results regardless of whether or not this support is available, you can also use the CheckDeviceFormat Conversion method off of the Manager class to detect this ability directly. Full-screen applications cannot do color conversion at all. You may also use Format.Unknown in windowed mode.

This is quite useful if you know beforehand the only types of formats you will support. There isn't much of a need to enumerate through every possible permutation of device types and formats if you already know what you need.

[\[Team LiB\]](#)

Checking Device Capabilities

For every possible thing a device can do that may or may not be capable of happening purely in hardware, there is an item called a "capability," or "Cap" for short. There is a Caps structure in the Direct3D runtime that lists every possible capability a device can have. Once a device is created, you can use the "Caps" property of the device to determine the features it supports, but if you want to know the features the device can support before you've created it, then what? Naturally, there is a method in the Manager class that can help here as well.

Now to add a little code to our existing application that will get the capabilities of each adapter in our system. We can't add the list of capabilities to the current tree view because of the sheer number of capabilities included (there are hundreds of different capabilities that can be checked). The easiest way to show this data will be to use a text box.

Go ahead and go back to design view for our windows form, and switch the tree view's dock property from "Fill" to "Left". It's still the entire size of the window though, so cut the width value in half. Now, add a text box to the window, and set its dock member to "Fill". Also make sure "Multiline" is set to true and "Scrollbars" is set to "Both" for the text box.

Now you will want to add a hook to the application so that when one of the adapters is selected, it will update the text box with the capabilities of that adapter. You should hook the "AfterSelect" event from the tree view (I will assume you already know how to hook these events). Use the following code in the event handler function:

```
private void treeView1_AfterSelect(object sender, System.Windows.Forms.TreeViewEventArgs e)
{
    if (e.Node.Parent == null) // root node
    {
        // Get the device caps for this node
        textBox1.Text = e.Node.Text + " Capabilities: \r\n\r\n" + Manager.GetDeviceCaps
            (e.Node.Index, DeviceType.Hardware).ToString().Replace("\n", "\r\n");
    }
}
```

As you can see, this is relatively simple. For the root nodes (which happen to be our adapters), after they are selected we call the Manager.GetDeviceCaps function, passing in the adapter ordinal for the node (which happens to be the same as the index). The ToString member of this structure will return an extremely large list of all capabilities of this device. Running the application now will result in something like [Figure 2.2](#).

Figure 2.2. Device display mode and capabilities.



[\[Team LiB\]](#)

In Brief

In this chapter, we learned how to pick the right graphics device for our applications. The items covered included

- Enumerating each adapter on your system
- Enumerating formats supported for your device
- Detecting the capabilities of the devices we are enumerating

In our next chapter we'll begin covering more efficient ways of rendering geometry, plus texturing for even more realistic scenes.

Chapter 3. Rendering Using Simple Techniques

IN THIS CHAPTER

- [Using Vertex Buffers](#)
- [Texturing Our Objects](#)
-

[In Brief](#)

The rendering done up to this point hasn't been done very efficiently. New lists of vertices were allocated every time the scene was rendered, and everything was stored in system memory. With modern graphics cards having an abundance of memory built into the card, you can get vast performance improvements by storing your vertex data in the video memory of the card. Having the vertex data stored in system memory requires copying the data from the system to the video card every time the scene will be rendered, and this copying of data can be quite time consuming. Removing the allocation from every frame could only help as well.

[\[Team LiB\]](#)

Using Vertex Buffers

Direct3D has just the mechanism needed for this: vertex buffers. A vertex buffer, much like its name, is a memory store for vertices. The flexibility of vertex buffers makes them ideal for sharing transformed geometry in your scene. So how can the simple triangle application from [Chapter 1](#), "Introducing Direct3D," be modified to use vertex buffers?

Creating a vertex buffer is quite simple. There are three constructors that can be used to do so; we will look at each one.

```
public VertexBuffer ( Microsoft.DirectX.Direct3D.Device device , System.Int32
    sizeofBufferInBytes , Microsoft.DirectX.Direct3D.Usage usage ,
    Microsoft.DirectX.Direct3D.VertexFormats vertexFormat ,
    Microsoft.DirectX.Direct3D.Pool pool )
```

```
public VertexBuffer ( System.Type typeVertexType , System.Int32 numVerts ,
    Microsoft.DirectX.Direct3D.Device device ,
    Microsoft.DirectX.Direct3D.Usage usage ,
    Microsoft.DirectX.Direct3D.VertexFormats vertexFormat ,
    Microsoft.DirectX.Direct3D.Pool pool )
```

The various parameter values can be

- - device— The Direct3D device you are using to create this vertex buffer. The vertex buffer will only be valid on this device.
- - sizeofBufferInBytes— The size you want the vertex buffer to be, in bytes. If you are using the constructor with this parameter, the buffer will be able to hold any type of vertex.
- - typeVertexType— If you want your vertex buffer to only contain one type of vertex, you can specify that type here. This can be the type of one of the built-in vertex structures in the CustomVertex class, or it can be a user defined vertex type. This value cannot be null.
- - numVerts— When specifying the type of vertex you want to store in your buffer, you must also specify the maximum number of vertices the buffer will store of that type. This value must be greater than zero.
- - usage— Defines how this vertex buffer can be used. Not all members of the Usage type can be used when creating a vertex buffer. The following values are valid:
 - - DoNotClip— Used to indicate that this vertex buffer will never require clipping. You must set the clipping render state to false when rendering from a vertex buffer using this flag.
 - - Dynamic— Used to indicate that this vertex buffer requires dynamic memory use. If this flag isn't specified, the vertex buffer is static. Static vertex buffers are normally stored in video memory, while dynamic buffers are stored in AGP memory, so choosing this option is useful for drivers to determine where to store the memory. See the DirectX SDK documentation for more information on Dynamic usage.
 -

[\[Team LiB\]](#)

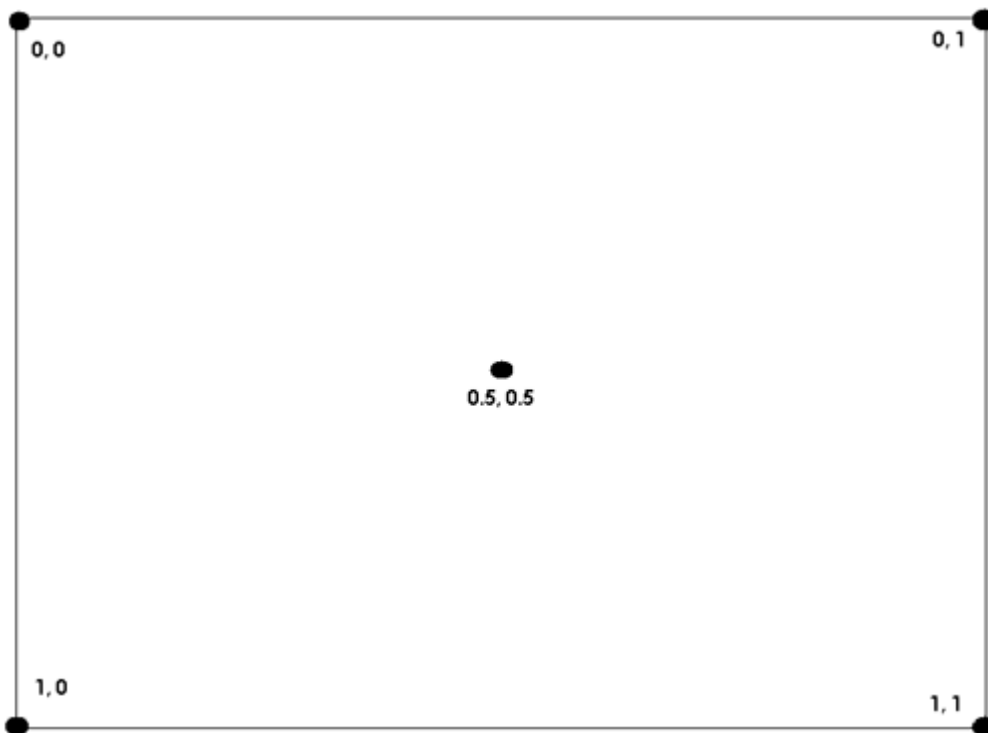
[\[Team LiB\]](#)

Texturing Our Objects

While rendering with colors and lights is quite exciting, objects don't look realistic enough if this is all we are using. The term "texture" when describing non-3D applications is usually in reference to the roughness of an object. Textures in a 3D scene are essentially flat 2D bitmaps that can be used to simulate a texture on a primitive. You might want to take a bitmap of grass to make a nice looking hill, or perhaps clouds to make a sky. Direct3D can render up to eight textures at a time for each primitive, but for now, let's just deal with a single texture per primitive.

Since Direct3D uses a generic bitmap as its texture format, any bitmap you load can be used to texture an object. How is the flat 2D bitmap converted into something that is drawn onto a 3D object, though? Each object that will be rendered in a scene requires texture coordinates, which are used to map each texel to a corresponding pixel on screen during rasterization. Texel is an abbreviation for texture element, or the corresponding color value for each address in a texture. The address can be thought of much like a row and column number, which are called "U" and "V" respectively. Normally, these values are scalar; that is the valid ranges for them go from 0.0 to 1.0. 0, 0 is located at the upper left corner of the texture, while 1, 1 would be located at the bottom right corner of the texture. The center of the texture would be located at 0.5, 0.5. See [Figure 3.2](#).

Figure 3.2. Visualizing texture coordinates.



In order to render our boxes with a texture, we must change the vertex format we are using to render our box, as well as the data we are passing down to the graphics card. We will replace the "color" component of our vertex data with texture coordinate data. While it's perfectly valid to have our object both colored and textured, for this exercise, we will simply use the texture to define the color of each primitive. Modify your vertex creation code as in [Listing 3.2](#):

Listing 3.2 Textured Box Data

```
CustomVertex.PositionTextured[] verts = new CustomVertex.PositionTextured[36];
```

```
// Front face
verts[0] = new CustomVertex.PositionTextured(-1.0f, 1.0f, 1.0f, 0.0f, 0.0f);
verts[1] = new CustomVertex.PositionTextured(-1.0f, -1.0f, 1.0f, 0.0f, 1.0f);
verts[2] = new CustomVertex.PositionTextured(1.0f, 1.0f, 1.0f, 1.0f, 0.0f);
verts[3] = new CustomVertex.PositionTextured(-1.0f, -1.0f, 1.0f, 0.0f, 1.0f);
verts[4] = new CustomVertex.PositionTextured(1.0f, -1.0f, 1.0f, 1.0f, 1.0f);
verts[5] = new CustomVertex.PositionTextured(1.0f, 1.0f, 1.0f, 1.0f, 0.0f);
```

```
// Back face (remember this is facing *away* from the camera, so vertices should be
// clockwise order)
```

[\[Team LiB\]](#)

In Brief

In this chapter, our applications started using much more efficient rendering techniques, for example

- Using vertex buffers to store our vertex data, and drawing multiple primitives from the same buffer
- Creating textures for more realistic objects
-

Including data in our vertex buffer to tell Direct3D where to render the texture on the primitives

In the next chapter we will begin discussing more advanced rendering techniques, including using index buffers, using other primitive types, and using depth buffers.

Chapter 4. More Rendering Techniques

IN THIS CHAPTER

-

[Rendering Other Primitive Types](#)

-

[Using Index Buffers](#)

-

[Using Depth Buffers](#)

-

[In Brief](#)

Now that we have the basic rendering concepts down, we can focus more on rendering techniques that can help improve performance and make our scenes look even better. This chapter will focus on rendering techniques such as

-

Using primitive types other than a list of triangles

-

Using index buffers for more controlled rendering

-

Using depth buffers for more realistic rendering

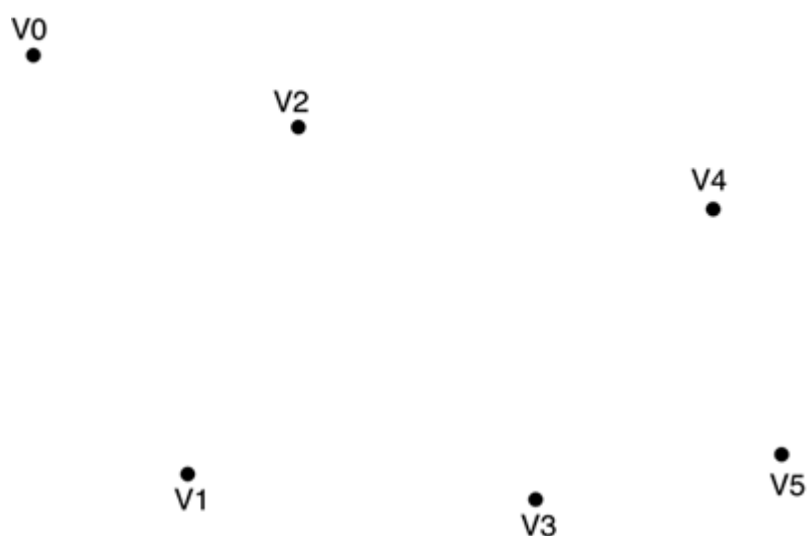
[\[Team LiB\]](#)

Rendering Other Primitive Types

All of the code we've written thus far concentrated on drawing a single type of primitive, namely a list of triangles. However, there are actually numerous different primitive types we can draw. The following list describes these primitive types:

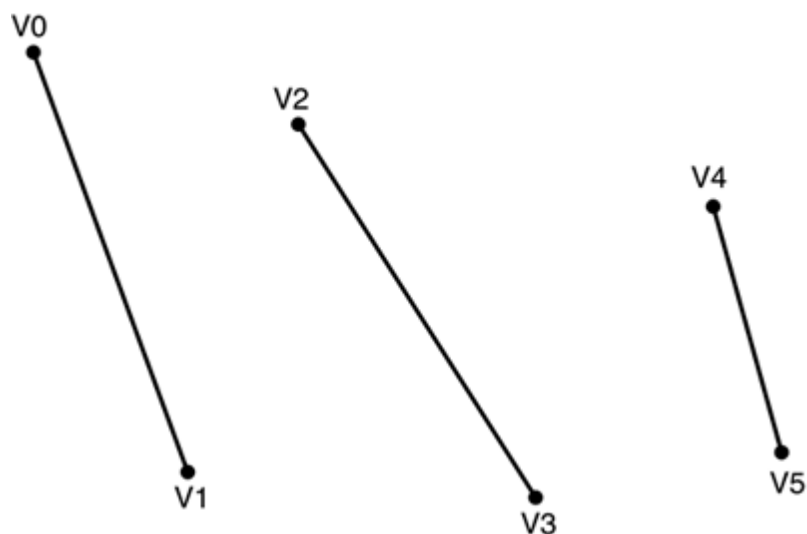
- PointList— This is a self-describing primitive type, as it will render the data as a list of points. You cannot use this primitive type if you are drawing indexed primitives (which we'll cover later in this chapter). See [Figure 4.1](#).

Figure 4.1. Point lists.



- LineList— Renders each pair of vertices as individual lines. You must pass in an even number of vertices (at least two) when using this primitive type. See [Figure 4.2](#).

Figure 4.2. Line lists.



- LineStrip— Renders the vertices as a single polyline. After the first line segment is drawn using the first two vertices, each subsequent line segment is drawn using the previous line's endpoint as its start point. You must pass in at least two vertices for this primitive type. See [Figure 4.3](#).

Figure 4.3. Line strips.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Index Buffers

If you remember our first application rendering cubes, we created data for a total of 36 vertices. There were 2 triangles for each face of the cube; 6 faces multiplied by 2 triangles equals 12 primitives. Since each primitive has 3 vertices, we get our 36. However, in reality, there were only 8 different vertices being used; one for each corner of the cube.

Storing the same vertex data multiple times in a small application such as this may not seem like that big of a deal, but in a large-scale application where you're storing mass amounts of vertex data, saving room by not duplicating your vertices is nothing but a good thing. Luckily, Direct3D has a mechanism for sharing vertex data inside a primitive called index buffers.

Like its name implies, an index buffer is a buffer that stores indices into vertex data. The indices stored in this buffer can be 32-bit (integer data) or 16-bit (short data). Unless you really need more indices than provided by the short data type, stick with that as it is half the size of the integer data.

When using an index buffer to render your primitives, each index in your index buffer corresponds directly to a vertex stored in your vertex data. For example, if you were rendering a single triangle with the indices of 0, 1, 6, it would render the triangle using the vertices mapped to those indices. Let's modify our cube drawing application to use indices. First, modify our vertex data creation function as shown in [Listing 4.3](#):

Listing 4.3 Creating Vertices for Our Cube

```
vb = new VertexBuffer(typeof(CustomVertex.PositionColored), 8, device, Usage.Dynamic |
    Usage.WriteOnly, CustomVertex.PositionColored.Format, Pool.Default);

CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[8];

// Vertices
verts[0] = new CustomVertex.PositionColored(-1.0f, 1.0f, 1.0f, Color.Purple.ToArgb());
verts[1] = new CustomVertex.PositionColored(-1.0f, -1.0f, 1.0f, Color.Red.ToArgb());
verts[2] = new CustomVertex.PositionColored(1.0f, 1.0f, 1.0f, Color.Blue.ToArgb());
verts[3] = new CustomVertex.PositionColored(1.0f, -1.0f, 1.0f, Color.Yellow.ToArgb());
verts[4] = new CustomVertex.PositionColored(-1.0f, 1.0f, -1.0f, Color.Gold.ToArgb());
verts[5] = new CustomVertex.PositionColored(1.0f, 1.0f, -1.0f, Color.Green.ToArgb());
verts[6] = new CustomVertex.PositionColored(-1.0f, -1.0f, -1.0f, Color.Black.ToArgb());
verts[7] = new CustomVertex.PositionColored(1.0f, -1.0f, -1.0f,
Color.WhiteSmoke.ToArgb());
buffer.SetData(verts, 0, LockFlags.None);
```

As you can see, we dramatically lowered the amount of vertex data, by only storing the 8 vertices that make up the corners of the cube. Now, we still want to draw 36 vertices, just with different orders of these 8 vertices. Since we know what our vertices are now, what are the 36 indices to these vertices that we would need to use to draw our cube? Looking at our previous application, we could compare each of the 36 vertices used, and find the appropriate index in our new list. Add our list of indices, shown in [Listing 4.4](#), to the declaration section:

Listing 4.4 The Box Index Buffer Data

```
private static readonly short[] indices = {
    0,1,2, // Front Face
    1,3,2, // Front Face
    4,5,6, // Back Face
    6,5,7, // Back Face
    0,5,4, // Top Face
    0,2,5, // Top Face
    1,6,7, // Bottom Face
    1,7,3, // Bottom Face
    0,6,1, // Left Face
    4,6,0, // Left Face
    2,3,7, // Right Face
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Depth Buffers

A depth buffer (often referred to as a z-buffer or a w-buffer) is how Direct3D stores "depth" information for rendering. This depth information is used during rasterization to determine how pixels block (occlude) each other. Currently, our application has no depth buffer associated with it, so no pixels are occluded during rasterization. However, we have no pixels that are occluded anyway, so let's try to draw a few more cubes that will actually overlap some of our existing cubes. Add the following DrawBox calls to the end of our existing DrawBox calls:

```
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle / (float)Math.PI /
    4.0f, 0.0f, (float)Math.Cos(angle), (float)Math.Sin(angle));
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle / (float)Math.PI *
    4.0f, 5.0f, (float)Math.Sin(angle), (float)Math.Cos(angle));
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle / (float)Math.PI /
    2.0f, -5.0f, (float)Math.Cos(angle), (float)Math.Sin(angle));
```

All we are really doing here is having three extra cubes spin around the three center horizontal cubes. Running this application, you can easily see the cubes overlapping, but you can't visually see where one cube ends and the other begins. It appears as if the cubes are simply a blob. This is where a depth buffer comes in handy.

Adding a depth buffer to our application is actually quite an easy task. Do you remember the presentation parameters we passed in to our device constructor? Well, this is where we include our depth buffer information. There are two new parameters we will need to fill out to include a depth buffer with our device:

```
public Microsoft.DirectX.Direct3D.DepthFormat AutoDepthStencilFormat [get, set]
public bool EnableAutoDepthStencil [get, set]
```

SHOP TALK: SAVING MEMORY WITH INDEX BUFFERS

So how exactly does using an index buffer save memory for our application? Let's compare the relatively simple cube applications, both with and without using the index buffer. In the first scenario, we created a vertex buffer holding 32 vertices of type CustomVertex.PositionColored. This structure happens to be 16 bytes (4 bytes each for x, y, z, and color). Multiply this size by the number of vertices, and you can see our vertex data consists of 576 bytes of data.

Now, compare this with the index buffer method. We only use 8 vertices (of the same type), so our vertex data size is 128 bytes. However, we are also storing our index data as well, so what is the size of that? We are using short indices (2 bytes each) and we are using 36 of them. So our index data alone is 72 bytes, combined with our vertex data we have a total size of 200 bytes.

Compare that to our original size of 576 bytes, and we have a remarkable 65% reduction in memory usage. Extrapolate these numbers out for very large scenes, and the possible memory saved using index buffers can be quite substantial.

Setting "EnableAutoDepthStencil" to true will turn on the depth buffer for your device, using the depth format specified in the AutoDepthStencilFormat member. Valid values of the DepthFormat enumeration for this member are listed in [Table 4.1](#):

Table 4.1. Possible Depth Formats

VALUE	DESCRIPTION
-------	-------------

[\[Team LiB\]](#)

In Brief

In this chapter we covered more advanced rendering techniques, including

- - Using index buffers to eliminate the need for duplicating vertices and lower our overall memory usage
- - The differences between each of the various primitive types and how to use each of them
- - Depth buffers, and how to use them

In the next chapter we will start rendering real objects loaded from files. It's time to move on to meshes.

Chapter 5. Rendering with Meshes

IN THIS CHAPTER

-

- [Defining the Mesh](#)

-

- [Using Materials and Lighting](#)

-

- [Using Meshes to Render Complex Models](#)

-

- [In Brief](#)

During this chapter we will

-

- Discuss and use the Mesh object.

-

- Get introduced to materials.

-

- Draw common objects as meshes.

-

- Use meshes to load and render external files.

[\[Team LiB\]](#)

Defining the Mesh

While there are many times when you may want to manually create vertex and index data, it's also common to want to load existing vertex data from an external source, such as a file. A common file format holding this data is an .X file. In the previous chapters, large portions of the code were there just to create simple objects that were to be rendered. While for the simple cubes and triangles, it wasn't all that bad, imagine trying to write similar code that was creating an object that had tens of thousands of vertices instead of the 36 that were used for the cube. The amount of time and effort to write this code would be substantial.

Luckily there is an object in Managed DirectX that can encapsulate storing and loading vertex and index data, namely the Mesh. Meshes can be used to store any type of graphical data, but are mainly used to encapsulate complex models. Meshes also contain several methods used to enhance performance of rendering these objects. All mesh objects will contain a vertex buffer and an index buffer like we've already used, plus will also contain an attribute buffer that we'll discuss later in this chapter.

The actual mesh object resides in the Direct3D Extensions library (D3DX). Currently, we've only had references to the main Direct3D assembly, so before we can use this mesh object, we'll first need to add a reference to the Microsoft.DirectX.D3DX.dll assembly to our project. Now, let's try to create our rotating box application using the mesh object.

After you make sure you have the correct references loaded in your solution, you should add a member variable for your mesh. You would place this in the same spot you put your vertex buffer and index buffer member variables before:

```
private Mesh mesh = null;
```

There are three constructors for creating a mesh; however, currently none of these will be needed. There are several static methods on the Mesh class that we can use to create or load various models. One of the first you'll notice is the "Box" method, which, as its name implies, creates a mesh that contains a box. Considering that's what we want to render right now, that looks perfect. Place the following code after your device creation method:

```
mesh = Mesh.Box(device, 2.0f, 2.0f, 2.0f);
```

This method will create a new mesh that contains the vertices and indices to render a cube with a height, width, and depth of 2.0f. This is the same size cube we created manually with our vertex buffer before. We've reduced all of that creation code we had down to one line; it doesn't get much easier than that.

Now that we've got our mesh created, though, do we render it the same way, or do we need to do something different? When rendering our cube before, we needed to call SetStreamSource to let Direct3D know which vertex buffer to read data from, as well as setting the indices and vertex format property. None of this is required when rendering a mesh.

DRAWING WITH A MESH

Saying the methods you used when drawing the cube before aren't necessary isn't entirely accurate. The mesh stores the vertex buffer, index buffer, and vertex format being used internally. When the mesh is rendered, it will automatically set the stream source, as well as the indices and vertex format properties for you.

Now that our mesh has been created, what do we need to do in order to render it onscreen? Meshes are broken up into a group of subsets (based on the attribute buffer, which we'll discuss shortly) and there is a method "DrawSubset" we can use for rendering. Let's modify our DrawBox function as follows:

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Materials and Lighting

So what's different now than when we were first using lights? The only major difference (other than the fact that we are using a mesh) is the lack of color in our vertex data. This is the cause of the light "failing" now.

In order for Direct3D to correctly calculate the color of a particular point on a 3D object, it must not only know the color of the light, but how that object will reflect the color of that light. In the real world if you shine a red light on a light blue surface, that surface will appear a soft purple color. You need to describe how our "surface" (our cube) reflects light.

In Direct3D, materials describe this property. You can specify how the object will reflect ambient and diffuse lighting, what the specular highlights (discussed later) will look like, and whether the object appears to emit light at all. Add the following code to your DrawBox call (before the DrawSubset call).

```
Material boxMaterial = new Material();
boxMaterial.Ambient = Color.White;
boxMaterial.Diffuse = Color.White;
device.Material = boxMaterial;
```

Here we create a new material, and set its ambient and diffuse color values to white. Using white means we will reflect all ambient and diffuse lighting that hits these objects. We then use the Material property on the device so that Direct3D knows which material to use while rendering its data.

Running the application now shows us the red cubes spinning around that we expected to see before. Modifying the ambient light color will change the color of every cube in the scene. Modifying the ambient color component of the material will change how the light is reflected off of the object. Changing the material to a color with absolutely no red in it (such as green) will cause the object to once again render black. Changing the material to a color with some red (for example, gray) will cause the object to appear a darker gray.

There are actually quite a few stock objects you can use when using mesh files. Use any of the following methods for creating these meshes (each of the stock mesh functions requires the device as the first parameter):

```
mesh = Mesh.Box(device, 2.0f, 2.0f, 2.0f);
```

SHOP TALK: USING MORE REALISTIC LIGHTING

As you can tell, the cubes don't look very realistic when rendered in this way. You can't even see where the "corners" of the cube are; it just looks like a blob of red in a cube-like shape. This is due to how ambient light affects the scene. If you remember, ambient light will calculate lighting the same for every vertex in a scene, regardless of normal, or any parameter of the light. See [Figure 5.1](#).

Figure 5.1. Ambient light with no shading.



[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Meshes to Render Complex Models

Rendering teapots can be quite exciting, but it's not often you make a game with nothing but teapots. Most meshes are created by artists using a modeling application. If your modeling application supports exporting to the X file format, you're in luck!

There are a few types of data stored in a normal x file that can be loaded while creating a mesh. There is naturally the vertex and index data that will be required to render the model. Each of the subsets of the mesh will have a material associated with it. Each material set can contain texture information as well. You can also get the High Level Shader Language (HLSL) file that should be used with this mesh while loading the file. HLSL is an advanced topic that will be discussed in depth later.

CONVERTING COMMON MODELING FORMATS TO THE X FILE FORMAT

The DirectX SDK (included on the CD) includes several conversion utilities for the most popular modeling applications out there. Using these conversion tools allows you to easily use your high-quality models in your applications.

Much like the static methods on the Mesh object that allowed us to create our generic "simple" primitive types, there are two main static methods on the mesh object that can be used to load external models. These two methods are Mesh.FromFile and Mesh.FromStream. The methods are essentially identical, with the stream method having more overloads for dealing with the size of the stream. The root overload for each method is as follows:

```
public static Microsoft.DirectX.Direct3D.Mesh FromFile ( System.String filename ,
    Microsoft.DirectX.Direct3D.MeshFlags options ,
    Microsoft.DirectX.Direct3D.Device device ,
    Microsoft.DirectX.Direct3D.GraphicsStream adjacency ,
    out Microsoft.DirectX.Direct3D.ExtendedMaterial[] materials ,
    Microsoft.DirectX.Direct3D.EffectInstance effects )

public static Microsoft.DirectX.Direct3D.Mesh FromStream ( System.IO.Stream stream ,
    System.Int32 readBytes ,
    Microsoft.DirectX.Direct3D.MeshFlags options ,
    Microsoft.DirectX.Direct3D.Device device ,
    Microsoft.DirectX.Direct3D.GraphicsStream adjacency ,
    out Microsoft.DirectX.Direct3D.ExtendedMaterial[] materials ,
    Microsoft.DirectX.Direct3D.EffectInstance effects )
```

The first parameter(s) are the source of the data we will be using to load this mesh. In the FromFile case, this is a string that is the filename of the mesh we wish to load. In the stream case, this is the stream, and the number of bytes we wish to read for the data. If you wish to read the entire stream, simply use the overload that does not include the readBytes member.

The MeshFlags parameter controls where and how the data is loaded. This parameter may be a bitwise combination of the values found in [Table 5.1](#).

Table 5.1. Mesh Flags Enumeration Values

PARAMETER

VALUE

MeshFlags_DeNotClip

Use the Usage_DeNotClip flag for vertex and index

[\[Team LiB\]](#)

In Brief

In this chapter we covered meshes, including

- - Using the Mesh object
- - Using materials
- - Using meshes to draw common objects
- - Using meshes to load and render external files

In our next chapter, we will explore optimizations that we can make to mesh data, and more advanced features of the mesh.

Chapter 6. Using Managed DirectX to Write a Game

IN THIS CHAPTER

-

- [Choosing the Game](#)

-

- [Writing the Game](#)

-

- [Adding a Movable Car into Your Scene](#)

-

- [Adding Obstacles](#)

-

- [Implementing the Finishing Touches](#)

-

- [In Brief](#)

Choosing the Game

While many of the more advanced features of 3D programming haven't been discussed yet, there has been enough basic ground covered to write a simple 3D game. You will use the knowledge gained thus far, plus add a few new things.

Before we actually begin writing a game, it would be a good idea to come up with a plan. We will need to know the type of game we will be writing, the basic features it will hold, and so on. Given the limited set of topics we've covered this far, we obviously can't create something overly complex.

Instead, a simple game will be written. One of the first demo games for MS-DOS was called "Donkey," in which the user controlled a car, and the goal was to avoid the donkeys in the road. Sounds simple enough; this chapter will make a 3D version, without the donkeys. We'll call this game "Dodger."

Before we begin actually writing some code, though, we need to spend a little bit of time planning and designing our game. What will we need for the game, and what do we want to do while playing it? Well, obviously, we'll need a car class to control our vehicle. Then it might be a good idea to have a class to control the obstacles we will be trying to avoid. Plus, we'll need our main game engine class that will do all of the rendering, and keep it all together.

If you were attempting to create a commercial game that you wanted to be published and sold in stores, you would spend a lot of time coming up with a game proposal. Game proposals are in-depth documents that detail the vast majority of the game concepts and features. Given that the scope of this book is intended to cover the actual development work, and not necessarily the steps that you would need to take in order to get it published, this exercise will be skipped.

Another recommended document that is normally required before development work starts is the technical specification (spec for short). This includes a mildly detailed list of all the classes, plus the methods and properties the classes will implement. It may also contain UML diagrams that show the relationships between the objects. The goal of the spec is to sit down and really think about the design of your application before you start writing any code. Again, since the focus of this book is on the code, the spec for this project will be omitted as well. I would recommend that you always spend some time coming up with a spec before writing any code.

[\[Team LiB\]](#)

Writing the Game

You can jump right into Visual Studio and get the project started. Create a new C# Windows Application project called "Dodger". The default name of the form that has been created for you is Form1. Replace each instance of Form1 with DodgerGame, which is the name of the class that the code in this chapter will represent. You'll want to add references to the three Managed DirectX assemblies you've been using in your projects thus far, and include a using statement for them in your code file. You should set this project up much like you did most of the others. You will have a private Direct3D device variable and should modify your constructor as follows:

```
public DodgerGame()
{
    this.Size = new Size(800,600);
    this.Text = "Dodger Game";

    this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque, true);
}
```

This will set the window size to 800x600 (which should be plenty good enough), and establish the window title and the style, so the rendering code will work normally. Then you should modify the entry point for the application by replacing the main method with the one found in [Listing 6.1](#).

Listing 6.1 Main Entry Point for Your Game

```
static void Main()
{
    using (DodgerGame frm = new DodgerGame())
    {
        // Show our form and initialize our graphics engine
        frm.Show();
        frm.InitializeGraphics();
        Application.Run(frm);
    }
}
```

This should be familiar; it's essentially the same code you've used to start all of the examples thus far. You create the windows form, show it, initialize the graphics engine, and then run the form, and thus the application. Inside the InitializeGraphics function, though, is where you will start to make some changes. Add the method found in [Listing 6.2](#) to your application.

Listing 6.2 Initializing Your Graphics Components

```
/// <summary>
/// We will initialize our graphics device here
/// </summary>
public void InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;

    // Store the default adapter
    int adapterOrdinal = Manager.Adapters.Default.Adapter;
    CreateFlags flags = CreateFlags.SoftwareVertexProcessing;

    // Check to see if we can use a pure hardware device
    Caps caps = Manager.GetDeviceCaps(adapterOrdinal, DeviceType.Hardware);
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Adding a Movable Car into Your Scene

Now that you have the road rendering, as well as moving through the scene, you should add the object the player will interact with: the car. You could simply put the car variables and constants and code into the main class where we have the road sections, but that just wouldn't be very modular. You will want to maintain the car code separately and in its own class. Create the car class now. From the project menu, select the Add New Item option, and choose to add a new class, named "Car". Make sure you include the using clauses for the Managed DirectX references in this new code file that has been created for you.

What will the car class need to do? Since the car will remain stationary while everything moves below it, the car will not need to move backward or forward. However, given that the car will need to dodge any obstacles in the road, it will need to be able to move left or right. It will also need to be able to render itself. Given this information, you should add the following set of variables in your class:

```
// Car constants
public const float Height = 2.5f;
public const float Depth = 3.0f;
public const float SpeedIncrement = 0.1f;
private const float Scale = 0.85f;

// Car data information
private float carLocation = DodgerGame.RoadLocationLeft;
private float carDiameter;
private float carSpeed = 10.0f;
private bool movingLeft = false;
private bool movingRight = false;

// Our car mesh information
private Mesh carMesh = null;
private Material[] carMaterials = null;
private Texture[] carTextures = null;
```

These will control everything you need to control for your car. You have constants for the height and depth of the car, which will remain static (since you are only moving left or right). The speed at which the car's lateral movement can change is also constant. The final constant is because the car's model is slightly larger than the side of the road, so you will need to scale the model down slightly.

The non-constant data in the car class is relatively self-explanatory. You have the current location of the car on the road, which we will default to the left side. You have the diameter of the car, which will be used when it is time to determine when it has collided with an obstacle. You have the current lateral speed of the car (since the road can increase in speed, the car movement speed should increase as well). Of course, you also have our two Boolean variables that determine which direction you are moving, if at all. Lastly, the mesh data is stored.

The constructor for the car class will need to create the mesh object (and its associated structures) and calculate the diameter of the car. Replace the default constructor created for you with the code found in [Listing 6.7](#).

Listing 6.7 Creating a Car Class

```
/// <summary>
/// Create a new car device, and load the mesh data for it
/// </summary>
/// <param name="device">D3D device to use</param>
public Car(Device device)
{
    // Create our car mesh
    carMesh = DodgerGame.LoadMesh(device, @"..\..\car.x", ref carMaterials,
    ref carTextures);

    // We need to calculate a bounding sphere for our car
    VertexBuffer vb = carMesh.VertexBuffer;
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Adding Obstacles

Congratulations. This is the first interactive 3D graphics application you've created. The effect of simulating car movement has been achieved. While you know that it's really the road that is moving below the car, it appears as if the car is speeding down the road. You've got half of the gameplay options implemented already. What you need now are the obstacles that you will be trying to avoid. Much like you added the car class, add a new class called "Obstacle". Make sure you add the using clauses for the Managed DirectX references for this code file as well.

Rather than using the same mesh for every obstacle, it would be nice to have a little variety in the obstacle list: in both type of obstacle as well as color. You can use the stock objects that can be created from the mesh object to vary the types of mesh, and you can use the materials to vary the colors. You should add the constants and variables needed for the obstacle class from [Listing 6.10](#).

Listing 6.10 Obstacle Constants

```
// Object constants
private const int NumberMeshTypes = 5;
private const float ObjectLength = 3.0f;
private const float ObjectRadius = ObjectLength / 2.0f;
private const int ObjectStacksSlices = 18;

// obstacle colors
private static readonly Color[] ObstacleColors = {
    Color.Red, Color.Blue, Color.Green,
    Color.Bisque, Color.Cyan, Color.DarkKhaki,
    Color.OldLace, Color.PowderBlue, Color.DarkTurquoise,
    Color.Azure, Color.Violet, Color.Tomato,
    Color.Yellow, Color.Purple, Color.AliceBlue,
    Color.Honeydew, Color.Crimson, Color.Firebrick };

// Mesh information
private Mesh obstacleMesh = null;
private Material obstacleMaterial;
private Vector3 position;
private bool isTeapot;
```

As you can see from the first constant, there will be five different mesh types (sphere, cube, torus, cylinder, and teapot). The majority of these mesh types will have either a length parameter or a radius parameter. Since you will want the obstacles to be similarly sized, you should use constants for these parameters. Many of the mesh types also have extra parameters that control the number of triangles included in the mesh (stacks, slices, rings, and so forth). The last constant will be used for these parameters. Increase this value for more triangles (and a higher level of detail), decrease it for less.

The list of colors that is next is where you will pick your color for the mesh. I've randomly selected these colors, but you can feel free to add, remove, or change any of the colors in this list. You'll notice that you aren't storing an array of materials, nor are you storing textures for our meshes in this class. You know the default mesh types only contain one subset with no textures or materials, so this extra information isn't necessary.

Since the obstacles should be "lying" on the road as the player drives by, and you know in reality that it's the road that is moving (not the player), you will also need to ensure that the obstacles move with the world. You therefore maintain a position for the obstacle that will be updated every frame as the road moves. Finally, since you can't control the size of a teapot during creation, you need to know if we're using a teapot obstacle so that you can scale it to the correct size.

Replace the class constructor of the Obstacle class with the following new one:

```
public Obstacle(Device device, float x, float y, float z)
{
    // Store our position
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Implementing the Finishing Touches

Currently, you don't do anything with these state variables. You should implement the game-over logic first. You will force the player to hit any key to start the game. After the game is over (you've crashed), there will be a slight pause (approximately one second), and then hitting any key will start the game once again. The first thing you'll need to make sure is that while the game is over, you don't update any other state. So the very first lines in `OnFrameUpdate` should be

```
// Nothing to update if the game is over
if ((isGameOver) || (!hasGameStarted))
    return;
```

Now you need to handle the keystrokes to restart the game. At the very end of your `OnKeyDown` override, you can add this logic:

```
if (isGameOver)
{
    LoadDefaultGameOptions();
}

// Always set isGameOver to false when a key is pressed
isGameOver = false;
hasGameStarted = true;
```

This is the behavior you want. Now, when the game is over, and the player presses a key, a new game is started with the default game options. You can remove the call to `LoadDefaultGameOptions` from `InitializeGraphics` now if you want to, since it will be called as soon as you press a key to start a new game. However, you still don't have the code that will cause the slight pause after you crash. You can add that to your `OnKeyDown` override as well; you should do it directly after checking for `Escape`:

```
// Ignore keystrokes for a second after the game is over
if ((System.Environment.TickCount - gameOverTick) < 1000)
{
    return;
}
```

This will ignore any keystrokes (except `Escape` to quit) for a period of one second after the game is over. Now, you can actually play the game! There's definitely something missing, though. You're maintaining a score, but there is nothing in the game that tells the user anything. You will need to rectify that. There is a `Font` class in the `Direct3D` namespace that can be used to draw some text; however, there is also a `Font` class in the `System.Drawing` namespace, and these two class names will collide if you attempt to use "Font" without fully qualifying it. Luckily, you can alias with the `using` statement as follows:

```
using Direct3D = Microsoft.DirectX.Direct3D;
```

Now each font you create can be a different color, but must be a unique size and family. You will want to draw two different types of text for this game, so you will need two different fonts. Add the following variables to the `DodgerGame` class:

```
// Fonts
private Direct3D.Font scoreFont = null;
private Direct3D.Font gameFont = null;
```

You will need to initialize these variables, but can't do so until the device has been created. You should do this after the device creation code. This does not need to happen in the `OnDeviceReset` event since these objects will automatically handle the device reset for you. Just add these lines to the end of the `InitializeGraphics` method:

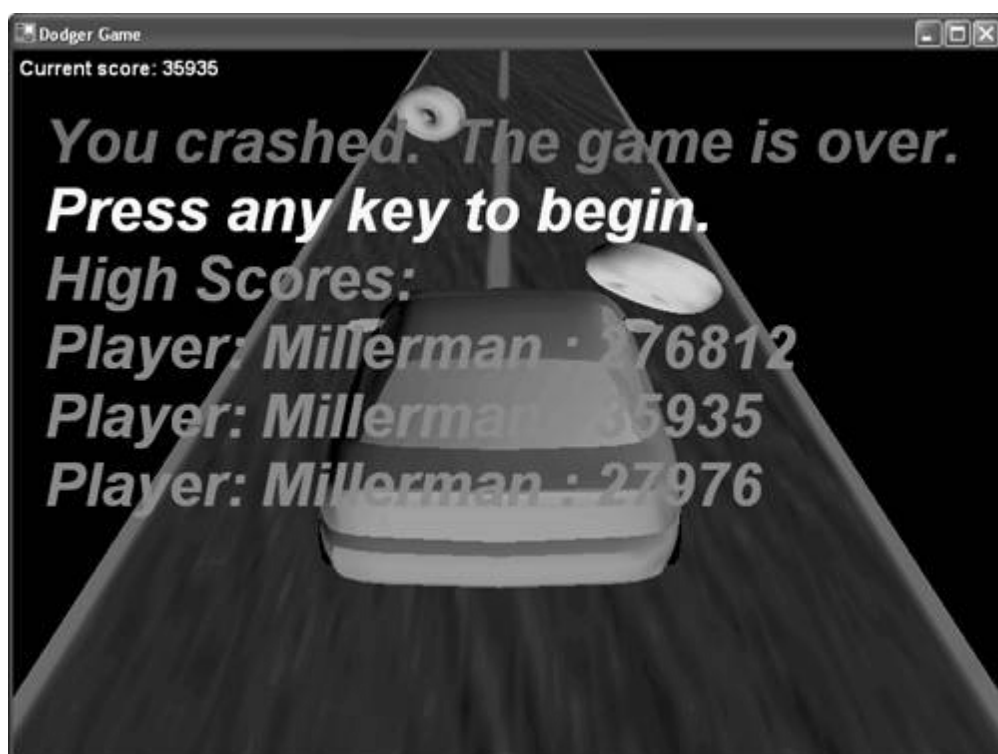
[\[Team LiB\]](#)

In Brief

- - Use the mesh classes to render the game objects.
- - Check certain device capabilities.
- - Basic user input.
- - Design a scoring system.
- - Putting it all together.

See if you can beat my best score in [Figure 6.1](#).

Figure 6.1. The completed game.



In the next chapter, we will discuss using advanced mesh features.

Part II: Intermediate Graphics Concepts

7 [Using Advanced Mesh Features](#)

8 [Understanding Resources](#)

9 [Using the Other Mesh Types](#)

10 [Using the Helper Classes](#)

Chapter 7. Using Advanced Mesh Features

IN THIS CHAPTER

- [Cloning Mesh Data](#)
- [Optimizing Mesh Data](#)
- [Simplifying Existing Meshes](#)
- [Welding Vertices in a Mesh](#)
- [Making Lots of Little Meshes Out of One Big One](#)
- [In Brief](#)

During the course of this chapter, we will discuss the more advanced features the mesh object contains, including

- Optimizing the mesh data
- Simplifying meshes
- Creating meshes with new vertex data components
- Welding vertices together

[\[Team LiB\]](#)

Cloning Mesh Data

So you've finally gotten your mesh loaded and rendered in your scene. The scene has a few lights, and yet your mesh just appears completely black. Glancing at the properties of your mesh, you notice the vertex format doesn't include normal data, which as we already know is required for lighting calculations.

What we need is a simple way to take all of the mesh data we already have, and simply add the normal data to it. If you guessed that there is already a mechanism for this, then you've guessed correctly. Look at [Listing 7.1](#):

Listing 7.1 Ensuring That Your Mesh Contains Normal Data

```
// Check if mesh doesn't include normal data
if ((mesh.VertexFormat & VertexFormats.Normal) != VertexFormats.Normal)
{
    Mesh tempMesh = mesh.Clone(mesh.Options.Value,
        mesh.VertexFormat | VertexFormats.Normal, device);
    tempMesh.ComputeNormals();
    // Replace existing mesh
    mesh.Dispose();
    mesh = tempMesh;
}
```

Here we take an existing mesh object we've created named "mesh", and check to see if it already includes normal data. The VertexFormat property returns a list of VertexFormats that are combined via a bitwise OR, so we use our bitwise AND operator to determine whether the normal bit is set. If it is not, we then create a second temporary mesh via the Clone method on our original mesh. The Clone method has three overloads:

```
public Microsoft.DirectX.Direct3D.Mesh Clone (
    Microsoft.DirectX.Direct3D.MeshFlags options ,
    Microsoft.DirectX.Direct3D.GraphicsStream declaration ,
    Microsoft.DirectX.Direct3D.Device device )

public Microsoft.DirectX.Direct3D.Mesh Clone (
    Microsoft.DirectX.Direct3D.MeshFlags options ,
    Microsoft.DirectX.Direct3D.VertexElement[] declaration ,
    Microsoft.DirectX.Direct3D.Device device )

public Microsoft.DirectX.Direct3D.Mesh Clone (
    Microsoft.DirectX.Direct3D.MeshFlags options ,
    Microsoft.DirectX.Direct3D.VertexFormats vertexFormat ,
    Microsoft.DirectX.Direct3D.Device device )
```

In our example, we used the final overload. In each of the overloads, the first and last parameters are the same. The options parameter allows the newly created mesh object to have a different set of options than the original mesh did. If you want to keep the same options in the new mesh, then you can do what we've done in our example code above, using the existing options from the original mesh; however, you can also change them. Say you wanted the new mesh to reside in system memory rather than managed memory. Any of the MeshFlags that are available during mesh creation are also available during cloning.

The last parameter of the clone method is the device the new mesh will be created on. In many instances, this device will be the same device you used to create the original mesh, but it is possible to clone a mesh to an entirely different device. For example, let's say you were writing an application that was multiple-monitor aware, and had each monitor running in full-screen mode. While your mesh object was on your first monitor, and being rendered there, it isn't necessary to have an "instance" of this mesh on the secondary monitor. If the mesh now needs to be rendered on the secondary monitor as well, you can easily clone it to the new device, and have it available there as well. Our preceding example simply used the existing device we already had.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Optimizing Mesh Data

Cloning a mesh isn't the only way to enhance an existing mesh. Meshes can also be optimized in multiple ways. The `Optimize` function on a mesh is similar to the `clone` method, in that it can generate a new mesh with different options; however it can also perform optimizations during the generation of this new mesh. You cannot use the `Optimize` function to add or remove vertex data or clone to a new device though. Look at the main overload for the `Optimize` method:

```
public Microsoft.DirectX.Direct3D.Mesh Optimize (
    Microsoft.DirectX.Direct3D.MeshFlags flags ,
    int[] adjacencyIn , out int[] adjacencyOut ,
    out int[] faceRemap ,
    out Microsoft.DirectX.Direct3D.GraphicsStream vertexRemap )
```

Each of the four overloads for this method takes this set of arguments, or simply the flags and adjacency in information. Note, that the `adjacencyIn` parameter can be used as an array of integers (as shown above), or as a `GraphicsStream`.

The flags parameter is used to determine how the new mesh can be created. This member can be one or more flags from the `MeshFlags` enumeration (excluding the `Use32Bit` or the `WriteOnly` flags). However, there are numerous `Optimize` flags in this enumeration that can be used specifically for this function. See [Table 7.1](#):

Table 7.1. Mesh Optimization Options

<code>MeshFlags.OptimizeCompact</code>	Reorders the faces in the mesh to remove unused vertices and faces.
<code>MeshFlags.OptimizeAttrSort</code>	Reorders the faces of the mesh so that there are fewer attribute state changes (i.e., materials), which can enhance <code>DrawSubset</code> performance.
<code>MeshFlags.OptimizeDeviceIndependent</code>	Using this option affects the vertex cache size by specifying a default cache size that works well on legacy hardware.
<code>MeshFlags.OptimizeDoNotSplit</code>	Using this flag specifies that vertices should not be split if they are shared between attribute groups.
<code>MeshFlags.OptimizeIgnoreVerts</code>	Optimize faces only; ignore vertices.
<code>MeshFlags.OptimizeStripeReorder</code>	Using this flag will reorder the faces to maximize the length of adjacent triangles.
<code>MeshFlags.OptimizeVertexCache</code>	Using this flag will reorder the faces to increase the cache hit rate of the vertex caches.

The `adjacencyIn` parameter is also required, and can either be an integer array containing three integers per face that specify the three neighbors for each face in the mesh, or a graphics stream containing this same data.

OPTIMIZING MESHES IN PLACE

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Simplifying Existing Meshes

Now let's say that your artist has just given you a mesh for "prop" that resides in differing locations in your scene, depending on which level you are on. In some levels, the mesh is way in the background, and doesn't need to be nearly as detailed as in other levels where it is right up close to the character. You could ask your artist to make you two different models, a high detailed one, and a low detailed one, but if the artist is already swamped with work, why not just use some of the built-in simplification our mesh class can provide.

Simplifying a mesh is the act of taking an existing mesh, using a set of provided weights, and trying to remove as many faces or vertices as possible to achieve a lower-detail mesh. However, before a mesh can be simplified, it must first be cleaned.

Cleaning a mesh is performed by adding another vertex where two fans of triangles share the same vertex. Let's look at the overloads for the clean method:

```
public static Microsoft.DirectX.Direct3D.Mesh Clean (
    Microsoft.DirectX.Direct3D.Mesh mesh ,
    Microsoft.DirectX.Direct3D.GraphicsStream adjacency ,
    Microsoft.DirectX.Direct3D.GraphicsStream adjacencyOut ,
    out System.String errorsAndWarnings )
```

You'll notice that much like the earlier functions, this one takes as a parameter in the mesh we are going to clean, as well as the adjacency information. However, you'll also notice that the adjacencyOut parameter is always required as well. The most common technique is to use the adjacency in graphics stream you get from the mesh creation as both the adjacencyIn and adjacencyOut parameter for the cleaning. There is also a string that can be returned to notify you of any errors or warnings encountered when cleaning the mesh. You should also note that the adjacency parameters can be either the graphics streams as shown above, or arrays of integers.

To show the dramatic simplification that can be achieved using these techniques, I took the MeshFile example that we wrote in [Chapter 5](#), "Rendering with Meshes," and decided to use this as the starting point for our simplification. First, I turned on wire-frame mode so you could see the effect the simplification has on the vertices more easily. I added the following line in the SetupCamera function:

```
device.RenderState.FillMode = FillMode.WireFrame;
```

Next, as we've just discussed, the mesh needs to be cleaned. Since we need to know the mesh's adjacency information for the cleaning, we need to modify the mesh creation code in LoadMesh as follows:

```
ExtendedMaterial[] mtrl;
GraphicsStream adj;

// Load our mesh
mesh = Mesh.FromFile(file, MeshFlags.Managed, device, out adj, out mtrl);
```

The only real change was the addition of the adj variable, which will hold our adjacency information. We then called the overload for FromFile that returned this data. Now we just need to make the actual call to Clean, so at the end of the LoadMesh function, we can add the following code to clean our mesh:

```
// Clean our main mesh
Mesh tempMesh = Mesh.Clean(mesh, adj, adj);
// Replace our existing mesh with this one
mesh.Dispose();
mesh = tempMesh;
```

Before we update our code to do the actual simplification of this mesh, we should first look at the actual Simplify

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Welding Vertices in a Mesh

Another way to simplify vertices with much less control, but at a much higher speed, is to weld similar vertices together. There is a method on the mesh class, "WeldVertices", that can be used to weld together vertices that are replicated and have equal attribute values. The method prototypes contain the following:

```
public void WeldVertices ( Microsoft.DirectX.Direct3D.WeldEpsilonsFlags flags ,
    Microsoft.DirectX.Direct3D.WeldEpsilons epsilons ,
    Microsoft.DirectX.Direct3D.GraphicsStream adjacencyIn ,
    Microsoft.DirectX.Direct3D.GraphicsStream adjacencyOut ,
    out int[] faceRemap ,
    Microsoft.DirectX.Direct3D.GraphicsStream vertexRemap )
```

The last four parameters have already been discussed at length, so we won't delve into those here once more. They are the same as they were in the clean function we've discussed previously. The first two parameters, however, control how the various vertices are welded together. The first parameter can be one or more of the values found in [Table 7.2](#):

Table 7.2. Flags for Welding Vertices

WeldEpsilonsFlags.WeldAll	Welds all vertices marked by adjacency as being overlapping.
WeldEpsilonsFlags.WeldPartialMatches	If the given vertex is within the epsilon value given by the WeldEpsilons structure, modify the partially matched vertices to be identical. If all components are equal, then remove one of the vertices.
WeldEpsilonsFlags.DoNotRemoveVertices	Can only be used if WeldPartialMatches is specified. Only allows modification of vertices, but not removal.
WeldEpsilonsFlags.DoNotSplit	Can only be used if WeldPartialMatches is specified. Does not allow the vertices to be split.

The WeldEpsilons structure itself is strikingly similar to the AttributeWeights structure used for simplifying a mesh. The only difference is there is a new member for this structure that controls the tessellation factor. Tessellating a mesh is the act of changing the level of detail in the model by either removing triangles, or splitting a single triangle into two or more smaller triangles. All other members remain the same.

To show the effect of using this method, we'll once again modify the existing MeshFile example to weld its vertices together. Since this is really only one call that operates in place, there isn't much code we will need to add. The first time a key is pressed we will do the weld, so replace the OnKeyPress method with the following:

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
    Console.WriteLine("Before: {0}", mesh.NumberVertices);
    mesh.WeldVertices(WeldEpsilonsFlags.WeldAll, new WeldEpsilons(), null, null);
    Console.WriteLine("After: {0}", mesh.NumberVertices);
}
```

Running the example now and pressing the space bar will result in the following text output:

```
Before: 4432
After: 3422
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Making Lots of Little Meshes Out of One Big One

Rather than just removing vertices from your mesh, what if you actually wanted to split your one large mesh into many different pieces? Let's once again take our MeshFile example and modify it slightly to split the one large mesh into smaller, more manageable meshes.

When you split a mesh up, you will take the existing single mesh and turn it into an array of meshes that can be used to do the rendering. For this example, you will maintain both the original mesh as well as the split mesh so that you can see them both rendered and compare them.

First, you will need to declare an array of meshes to hold the split mesh. You will also need some Boolean variables to control how and which meshes are being rendered. Add the following variables to your declaration section:

```
// Split information
private Mesh[] meshes = null;
private bool drawSplit = false;
private bool drawAllSplit = false;
private int index = 0;
private int lastIndexTick = System.Environment.TickCount;
```

These variables will hold the entire list of newly created meshes after the split, as well as the flag on whether or not you are drawing the original or split mesh (you should default to the original). If you are drawing the split mesh, there is a second flag to see whether you are drawing all of the split meshes in the array, or a single one at a time; again, you should default to a single one at a time. You will need to maintain the current index of the mesh you are drawing (in case you are drawing one at a time), plus a "mini-timer" to determine when to switch to the next mesh (if you are drawing one at a time).

With the variable declarations out of the way, we can create our array of meshes. Directly after the LoadMesh call in InitializeGraphics, add the following section of code:

```
meshes = Mesh.Split(mesh, null, 1000, mesh.Options.Value);
```

We should take a look at the various parameters the split function takes. There are two overloads, so we will look at the most complete one:

```
public static Microsoft.DirectX.Direct3D.Mesh[] Split (Mesh mesh , int[] adjacencyIn ,
    System.Int32 maxSize , MeshFlags options ,
    out GraphicsStream adjacencyArrayOut ,
    out GraphicsStream faceRemapArrayOut ,
    out GraphicsStream vertRemapArrayOut )
```

As you can see, the call takes in the mesh we want to split as the first parameter. The adjacency is the next parameter (you can pass null as we've done above if you don't care about this member). The third parameter is the maximum number of vertices you want each of the newly created meshes to have. In our previous example, we want a maximum of 1,000 vertices per new mesh. The options parameter is used to specify the flags for each of the newly created meshes. The final three out parameters all return the information about each of the new meshes in a stream. We will use the overloads without this information.

With our newly created array of meshes, we now need a way to specify which meshes we will be drawing: the original or the array. Before we can do that, though, we need a way to toggle between them. We will use the same keys we used earlier: the space bar and "m". Replace the OnKeyPress method with the following code:

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
    if (e.KeyChar == ' ')
    {
        // Toggle between original and split mesh
    }
}
```

[\[Team LiB\]](#)

In Brief

In this chapter, we learned just about everything there is to know about the standard mesh class, including

- - Optimizing mesh data
- - Simplifying meshes
- - Creating meshes with new vertex data components
- - Welding vertices together

In the next chapter, we will explore the resources available in Direct3D in depth, and the more advanced features of them.

Chapter 8. Understanding Resources

IN THIS CHAPTER

- [Starting with the Resource Class](#)
- [Using the Vertex and Index Buffers](#)
- [Locking Our Buffers](#)
- [Controlling How Buffers Are Locked](#)
- [Using Texture Resources](#)
- [Locking Textures and Getting Descriptions](#)
- [In Brief](#)

Resources are the backbone of rendering complex objects in Direct3D. During the course of this chapter, we will discuss the more advanced features of these resources, including

- Static and dynamic resources
- Updating the buffers included in meshes
- Uses of the various locking flags
- Using unsafe mode for maximum performance

[\[Team LiB\]](#)

Starting with the Resource Class

The Resource class is relatively small, and is the base class for all resources in Direct3D, so it may be a good idea to start this section with a list of each method, and what it's intended to be used for. You will never actually create or use the Resource class itself, since it is an abstract class, but you may find yourself using these methods on your other resource classes. See [Table 8.1](#) for the methods on the resource class:

Table 8.1. Resource Class Methods and Properties

PROPERTY OR METHOD	DESCRIPTION
Device	Read-only property that will return the device this resource was created with.
Type	Read-only property that returns the type of resource this is. Valid return values for this property are in the ResourceType enumeration.
Priority	Read-write property that returns the priority of this resource. Priorities only matter when dealing with resources that reside in the managed memory pool. They are used to determine when a resource can be removed from memory. The lower the priority, the faster a resource can be removed. By default, all managed resources have a priority of 0, and all non-managed resources will always have a priority of 0. There is also a SetPriority method, which returns the old priority after setting the new one.
PreLoad	Use this method to indicate that a managed resource will be needed shortly. This will allow Direct3D to move your resource into video memory before it is actually needed. It's important to note, that if you are already using more video memory than you have, this method will silently do nothing.
PrivateData members	There are three members that allow you to get and set private data for each of your resources. This data is never used by Direct3D and can be used by your application as you see fit.

As you can see, the resource class deals mainly with enhancing managed resources performance. While not listed in the chart above, or the resource class, it's important to note that each resource in Direct3D will also have a usage, as well as a memory pool (defined in the Usage and Pool enumerations). These properties define how the resource is used, and where it is located in memory (be it system memory, video memory, or AGP memory).

Resources all have a way to allow the CPU to access the data it stores, normally by a mechanism called Locking. In our examples thus far, we've filled our buffers with single "SetData" calls, and created our textures from files; however, it is possible to retrieve the data, or only set small subsections of that data, via locking these resources. We will discuss locking in this chapter briefly. Let's look at the first resource we were introduced to.

[\[Team LiB \]](#)

Using the Vertex and Index Buffers

Vertex buffers are the main data structure used by Direct3D to store vertices, while the Index buffers are the same for indices. These classes derive from the Resource class, and thus inherit those methods, plus add a few of their own.

The first new property is a description of the buffer. The structure returned from this property will tell you everything you need to know about how the buffer was created, including the format, the usage, the memory pool, the size, and the vertex format. While it may be true that you would already know this information if you were creating the buffers yourself, this is very useful information if you are simply receiving the buffers from an external source that you know nothing about.

SHOP TALK: STATIC AND DYNAMIC BUFFERS

Before getting into the last two methods that reside on the buffers, now seems like the perfect time to discuss the differences between static and dynamic buffers. The concepts here apply equally well to both vertex and index buffers.

So, obviously the first question would be, "What is the difference between a static and a dynamic buffer?" As luck would have it, I happen to have an answer for that. If the buffer was created with the usage flag Usage.Dynamic, the buffer is a dynamic buffer; any other buffer is a static buffer. Static buffers are designed for items that do not change very often, while dynamic buffers are more easily used for buffers that have constantly changing data. You can see where the names come from.

Just because a buffer is static, though, doesn't mean it cannot be modified. However, locking a static buffer while that buffer is already in use can have a very significant performance penalty. Before the data can be modified, the GPU needs to finish reading the current set of data, which can be very time consuming. Doing this several times in the same frame is even worse, because now you've eliminated the buffering the driver can do, and you are forcing your GPU to sit around idling waiting for you to finish modifying your buffer's data. Graphics cards are extremely powerful today; you do not want them idling when they could be busy drawing triangles.

If you expect your buffer's data to be modified frequently, it should be created with Usage.Dynamic. This flag allows Direct3D to optimize the pipeline for frequent data modification. You will receive no speed enhancements by specifying a buffer as dynamic, and then never modifying it; as a matter of fact, it will be slower than if you simply made a static buffer to begin with.

Always know the amount of "churn" you expect on your buffers per frame and create accordingly. For maximum performance, it is recommended that you have one large static vertex buffer for each vertex format you wish to render. It's sometimes not feasible to have only these static buffers, though, and in those cases, the dynamic buffers are there for you.

[\[Team LiB\]](#)

Locking Our Buffers

The locking mechanism seems to be one of the most misunderstood calls in Direct3D, particularly in Managed DirectX. How to do the locks, and do them fast seems to be a particularly tough question for people to grasp.

Before we get to that, though, what exactly is "locking" the buffer? It is simply the act of allowing your CPU to access a range of data in your resource. Since you cannot talk directly to the graphics hardware, there needs to be a way to allow you to manipulate vertex data from your application, and the locking mechanism is it. Look at the various lock overloads that exist for our vertex and index buffers (they take the same parameters):

```
public System.Array Lock ( System.Int32 offsetToLock ,
    Microsoft.DirectX.Direct3D.LockFlags flags )

public System.Array Lock ( System.Int32 offsetToLock , System.Type typeVertex ,
    Microsoft.DirectX.Direct3D.LockFlags flags , params int[] ranks )

public Microsoft.DirectX.Direct3D.GraphicsStream Lock ( System.Int32 offsetToLock ,
    System.Int32 sizeToLock , Microsoft.DirectX.Direct3D.LockFlags flags )
```

As you can see, there are three overloads available when attempting to lock our buffers. We should start with the first one, since it is the simplest. This overload is only available if you created your buffer via the constructor that takes a `System.Type` and a number of vertices or indices. In actuality, this overload really just calls the next overload with the data you passed in from the constructor.

The next two overloads are where things get interesting. The first parameter of each is the offset (in bytes) at which you want to start the lock. If you wish to lock the entire buffer, you will naturally want to start this member at zero. You'll notice the first two overloads each return an array, which can be extremely useful since it will return the data to you in an easy-to-use form. In the second overload, the second parameter is the base type of the array you wish to be returned. The last parameter is used to determine the size of the array that is returned.

This "ranks" parameter seems to throw everyone for a loop for some reason, so let's look at it in depth for a moment. Let's assume you had a vertex buffer that had nothing but position data in it (`Vector3`), and it had 1,200 vertices in it. If you wanted to lock this buffer into a `Vector3` array of 1,200 items, the lock call would look like this:

```
Vector3[] data = (Vector3[])vb.Lock(0, typeof(Vector3), LockFlags.None, 1200);
```

This call will result in a single dimension array of 1,200 `Vector3`s. Now let's say for some reason you wanted to have a two-dimensional array of 600 vertices each. That lock call would look like

```
Vector3[,] data = (Vector3[,])vb.Lock(0, typeof(Vector3), LockFlags.None, 600, 600);
```

Notice the extra parameter? The ranks parameter is actually a parameter array. It can and will create up to a three-dimensional array of data to return back to you. You should use this parameter to specify exactly how large of an array you wish to get back.

SHOP TALK: LOCKING BUFFERS WITH HIGH PERFORMANCE

LockFlags will be covered with the last overload, but first I want to point out some of the drawbacks of using either of the overloads that return an array. First and foremost, we should talk about performance. Assuming you have created a vertex buffer with the "default" options, and no lock flags, when this

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Controlling How Buffers Are Locked

Last, we need to talk about the flags you can use when locking the buffer. The only valid flags when locking a vertex buffer are

- LockFlags.None
- LockFlags.Discard
- LockFlags.NoOverwrite
- LockFlags.NoDirtyUpdate
- LockFlags.NoSystemLock
- LockFlags.ReadOnly

Obviously, when using no lock flags, the default locking mechanism is used. However, if you want more control over your locking, the other flags can be used for a variety of options.

The "Discard" flag can only be used on dynamic buffers. For Vertex and Index buffers, the entire buffer will be discarded, and a new chunk of memory will be returned to you in case there is anything else still accessing the old memory. This flag is useful if you are filling a vertex buffer dynamically. Once the vertex buffer has been filled completely, you lock with discard. It is often used in combination with the next flag.

The "NoOverwrite" flag (which is also only valid on dynamic buffers) tells Direct3D that you will not overwrite any data in the vertex or index buffer. Using the flag will allow the call to return immediately, and continue to use the buffer. If you do not use this flag, the lock call will not return until any rendering is currently finished. Since you may not overwrite any of the data currently in this buffer, it is only useful for appending vertices to a buffer.

A common use of these two flags comes from a large buffer you are filling dynamically. As you keep adding new data into your buffer, you continue to use the NoOverwrite flag until the buffer is full. At that time, your next lock will use the Discard flag to flush the buffer and start over. We can look at [Listing 8.1](#), an excerpt of code from the Point Sprites sample that ships with the DirectX SDK.

Listing 8.1 Portion of PointSprites Example

```
if (++numParticlesToRender == flush)
{
    // Done filling this chunk of the vertex buffer.  Lets unlock and
    // draw this portion so we can begin filling the next chunk.

    vertexBuffer.Unlock();

    dev.DrawPrimitives(PrimitiveType.PointList, baseParticle,
                      numParticlesToRender);

    // Lock the next chunk of the vertex buffer.  If we are at the
    // end of the vertex buffer, LockFlags.Discard the vertex buffer and start
    // at the beginning.  Otherwise, specify LockFlags.NoOverWrite, so we can
    // continue filling the VB while the previous chunk is drawing.
    baseParticle += flush;
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Texture Resources

All of the texture resources in Managed DirectX derive from a single `BaseTexture` class, which itself derives from the `Resource` class. There are a few new methods on the `BaseTexture` object that are covered in [Table 8.2](#).

Table 8.2. Base Texture Methods

PROPERTY OR METHOD	DESCRIPTION
<code>LevelCount</code>	This read-only property specifies the number of levels this texture has. For example, a mipmapped texture (described in the sidebar " Dissecting Mipmaps ") would have the number of mipmaps as its level count.
<code>GenerateMipSubLevels</code>	This method will automatically generate the mipmap sublevels we've just mentioned for you. It will use the next property to determine the filtering mode to use when creating these levels.
<code>AutoGenerateFilterType</code>	This read-write property describes how the mipmap sublevels are generated automatically. It takes a member of the <code>TextureFilter</code> enumeration, and will automatically re-create any mipmap sublevels if the filter type is changed. The default filter type for mipmap level generation is <code>TextureFilter.Linear</code> unless the driver does not support that mode, in which case it will use <code>TextureFilter.Point</code> . If the device does not support the selected filter type, this property set will throw an exception. All filter values are valid with the exception of "None". You can see the <code>TextureFilterCaps</code> member of the <code>Caps</code> structure to determine which filter types are supported by your device.
<code>LevelOfDetail</code>	Another read-write property that determines the most detailed level of detail for managed textures.

SHOP TALK: DISSECTING MIPMAPS

So what is this mipmap we keep talking about? Simply put, a mipmap is a chain of textures, with the first member of that chain being the most highly detailed texture. For each subsequent member of that chain, the size is reduced by one power of 2. For example, assuming that you have an original "high resolution" texture of 256x256, the next level in that mipmap would be 128x128, followed then by 64x64, and so on. Mipmap chains are used by Direct3D to control the quality of the rendered textures, at the expense of needing more memory to store the mipmaps. When objects are up close, the high-quality versions of the textures are used; when objects are far away, the lower-resolution textures are used instead.

When creating a texture, one of the parameters used specifies the number of levels you wish to have in this texture. This number of levels corresponds directly to the mipmap chain. Specifying zero for this parameter will cause Direct3D to automatically create a set of mipmaps from your original texture resolution all the way down to a final 1x1 texture in your mipmap chain. In our example above, using 0 for this parameter on a texture that has an original size of 256x256 will create a chain of nine textures: 256x256, 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, and 1x1.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Locking Textures and Getting Descriptions

Just like the vertex and index buffers earlier, each of the texture types also has a locking mechanism, and a way to determine the features it uses. In addition to these methods, textures also have two new features that the geometry resources do not: namely, dirty regions and a "backing" object. Before we get into those, let's explore the locking mechanism for textures.

The locking mechanism for textures is remarkably similar to locking our geometry buffers, with two differences. The first is the lock calls take in a "level" parameter, which is naturally the mipmap level you wish to perform this lock operation on. The second is the rectangle (or box for volume textures) you wish to perform the lock on. You can use the overloads that do not specify this member to lock the entire texture surface.

When dealing with cube textures, there is one extra parameter that is needed: the face of the cube map you wish to lock. You can use a value from the CubeMapFace enumeration for this parameter.

You may also notice that some of the overloads for LockRectangle can return a pitch parameter as an out. If you need to know the amount of data in each row, this is what the pitch will provide you. If you specify this parameter, it will return the number of bytes in a row of blows.

While the texture locks can take in a type parameter for locking specific types of data, the textures themselves only store color information. If you wish to use one of the overloads that use a type, you should ensure that type is the same size as the pixel format of the texture; for example, a 32-bit pixel format should be locked with an integer, while a 16-bit pixel format should be locked with a short.

We should look at an example of locking a texture now. We will use the MeshFile example we've already looked at as the basis for this example. We'll remove the texture that is currently used for rendering the model, and replace it with one we generate dynamically for our application. We will want to have two modes for this texture, one that is "cycling" the textures, and one that is purely random. We will need the following variables to be declared for this:

```
private uint texColor = 0;
private bool randomTextureColor = false;
```

We plan on having a 32-bit pixel format for our texture, so we store our cycling color in a 32-bit format (uint) as well. We also want to specify whether we are creating random texture colors, or using the cycling texture, thus the need for the Boolean. Now, we should create the method that will do the actual texture locking and updating. In order to show the differences between the array methods, and the unsafe methods, we will do both. The array method is shown in [Listing 8.2](#).

Listing 8.2 Filling a Texture with an Array

```
private void FillTexture(Texture t, bool random)
{
    SurfaceDescription s = t.GetLevelDescription(0);
    Random r = new Random();
    uint[,] data = (uint[,])t.LockRectangle(typeof(uint), 0,
        LockFlags.None, s.Width, s.Height);

    for (int i = 0; i < s.Width; i++)
    {
        for (int j = 0; j < s.Height; j++)
        {
            if (random)
            {
                data[i,j] = (uint)Color.FromArgb(
                    r.Next(byte.MaxValue),
                    r.Next(byte.MaxValue),
                    r.Next(byte.MaxValue)).ToArgb();
            }
            else
            {
                data[i,j] = texColor;
            }
        }
    }
}
```

[\[Team LiB\]](#)

In Brief

In this chapter we talked in depth about the resources available in a Direct3D application and how they can be used, including

- Static and dynamic resources
- Updating the buffers included in meshes
- Uses of the various locking flags
- Using unsafe mode for maximum performance

In our next chapter, we will begin looking at the more complex mesh types available to us.

Chapter 9. Using the Other Mesh Types

IN THIS CHAPTER

- [Simplifying Meshes](#)
- [Controlling the Level of Detail with Progressive Meshes](#)
- [Rendering Patch Meshes](#)
- [Seeing the Tessellation Levels](#)
-

[In Brief](#)

[\[Team LiB\]](#)

Simplifying Meshes

We spent a good amount of time understanding the basic Mesh class that is included in the extensions library; however, there are actually three other mesh types we haven't looked at yet. During the course of this chapter we will explore these remaining objects.

You've already looked at creating a simplified mesh by using the Simplify method on the mesh class. Since the common case for using a simplified mesh is to show a less-detailed version of a model, in any given scene, it's probably not a great idea to store multiple versions of a mesh, particularly if you're not even going to be using some of them.

The SimplificationMesh object can be used to encapsulate the process of simplifying a mesh. However, you can't actually use the object to do the rendering; you must first get an actual mesh from the simplification mesh. If you remember the simplified example before, there were two "modes" for the example. One mode was an up-close mode that displayed the model before it was simplified. There was also a "far away" mode that displayed the same model in the distance with many fewer faces and vertices. You will do something similar now, except allowing the model to gradually go into the distance.

Rather than starting with the simplify mesh sample we did in [Chapter 7](#), "Using Advanced Mesh Features," it will be easier if we simply start with the example we did in [Chapter 5](#), "Rendering with Meshes," for loading a mesh from a file. We will simply add our code to this example instead. The first thing we will need is a declaration for our SimplificationMesh object, as well as a variable to control our camera's position. Add the following variables:

```
private SimplificationMesh simplifiedMesh = null;
private float cameraPos = 580.0f;
```

Storing the camera's depth position as a variable will allow us to easily change where the camera is located. You will also need to update the view transform to actually use the correct position as well. In the SetupCamera method, update the view transform as follows:

```
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, cameraPos),
    new Vector3(), new Vector3(0,1,0));
```

Obviously since the SetupCamera method is called every frame, any update we make to this variable will update the camera as well, which is what we want. This simplification mesh variable is obviously where we store the data to simplify our mesh. If you notice, we don't have any extra mesh objects. Anytime we simplify our mesh, we will simply replace it with the "new" one.

We will need to modify the LoadMesh method in order to make sure our mesh is cleaned and our simplification mesh object is created correctly. We will need to replace this method with the one in [Listing 9.1](#):

Listing 9.1 Creating a Simplification Mesh

```
private void LoadMesh(string file)
{
    ExtendedMaterial[] mtrl;
    GraphicsStream adj;

    // Load our mesh
    mesh = Mesh.FromFile(file, MeshFlags.Managed, device, out adj, out mtrl);

    // If we have any materials, store them
    if ((mtrl != null) && (mtrl.Length > 0))
    {
        meshMaterials = new Material[mtrl.Length];
        meshTextures = new Texture[mtrl.Length];

        // Store each material and texture
```


[\[Team LiB\]](#)

[\[Team LiB\]](#)

Controlling the Level of Detail with Progressive Meshes

There are cases where you'd only want to simplify a mesh (for example, imagine a model for a rocket you've fired). However, if you are only simplifying, it means the object's detail never needs to be increased, which isn't as common as the cases where you may need to progressively lower or raise the level of detail. I'm sure you can see where the name of the progressive mesh came from.

To show the behavior of the progressive meshes, we will write a similar application to the one we wrote for the simplification mesh a moment ago. However, instead of just simplifying, we will also allow you to raise the level of detail by bringing the model closer to the camera as well. Once again we will start with the example to load a mesh from a file.

The `ProgressiveMesh` class derives from the `BaseMesh` class just like the `Mesh` class does. You can use the progressive mesh class to draw your objects, unlike the `SimplificationMesh` object. Knowing this, we can replace the declaration for your mesh object with this one:

```
private ProgressiveMesh progressiveMesh = null;
```

You will also need to replace each instance of the old mesh variable with the new `progressiveMesh` variable. With the new variable name and type, we obviously need to update the `LoadMesh` method, since our code won't even compile right now. We will use a similar method; we will just generate our progressive mesh at the end. Use the code shown in [Listing 9.3](#).

Listing 9.3 Loading a Progressive Mesh

```
private void LoadMesh(string file)
{
    ExtendedMaterial[] mtrl;
    GraphicsStream adj;

    // Load our mesh
    using(Mesh mesh = Mesh.FromFile(file, MeshFlags.Managed, device,
        out adj, out mtrl))
    {

        // If we have any materials, store them
        if ((mtrl != null) && (mtrl.Length > 0))
        {
            meshMaterials = new Material[mtrl.Length];
            meshTextures = new Texture[mtrl.Length];

            // Store each material and texture
            for (int i = 0; i < mtrl.Length; i++)
            {
                meshMaterials[i] = mtrl[i].Material3D;
                if ((mtrl[i].TextureFilename != null) &&
                    (mtrl[i].TextureFilename != string.Empty))
                {
                    // We have a texture, try to load it
                    meshTextures[i] = TextureLoader.FromFile(device,
                        @"..\..\\" + mtrl[i].TextureFilename);
                }
            }
        }

        // Clean our main mesh
        using(Mesh tempMesh = Mesh.Clean(mesh, adj, adj))
        {
            // Create our progressive mesh
            progressiveMesh = new ProgressiveMesh(tempMesh, adj,
                null, 1, MeshFlags.SimplifyVertex);
        }
    }
}
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering Patch Meshes

Simplifying meshes is a common practice, but what about the times when the meshes you've got are already too simple? You have enough triangle bandwidth to display even more vertices in your mesh. While it may not be common to want to increase the detail of a model, it is possible, and patch meshes are the way. Most modern 3D graphics modeling programs have a concept of a type of patch mesh, or some type of high order primitives such as NURBS or subdivided surfaces.

The objective of this book is not to explain the theory behind understanding patch meshes, so we will not get into that. What we want to do is to understand how to use them, and how they can increase a model's level of detail. To do this, we will create a small application that loads a model and allows us to subdivide the vertices to make a more complex model.

Once again, we will start with the file load mesh example from [Chapter 5](#). We keep doing this because it sets up the majority of the "state" for us to get started. You will need the models from the included CD for this example as well. The code on the included CD uses two of the models that are included in the DirectX SDK (tiger.x and cube.x), plus a small sphere model you will find on the CD. Make sure you copy the texture for the tiger model (tiger.bmp) as well. You will need the following variables added:

```
private float tessLevel = 1.0f;
private const float tessIncrement = 1.0f;
private string filename = @"..\..\sphere.x";
private Microsoft.DirectX.Direct3D.Font font = null;
```

You will have our current tessellation level (which is initially set to 1.0f), as well as how much we will increment this level each time. Feel free to manipulate this constant, but for now, the default value will do nicely. We will also need to store the name of the current model we are loading, as well as a font to draw the text during rendering.

You will also want to change the SetupCamera method since the last one was set up for a very large model, while the ones being used for this example are quite a bit smaller. Update the method as follows:

```
private void SetupCamera()
{
    device.Transform.Projection = Matrix.PerspectiveFovLH(
        (float)Math.PI / 4, this.Width / this.Height, 1.0f, 100.0f);

    device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 5.0f),
        new Vector3(), new Vector3(0,1,0));

    device.Lights[0].Type = LightType.Directional;
    device.Lights[0].Diffuse = Color.DarkKhaki;
    device.Lights[0].Direction = new Vector3(0, 0, -1);
    device.Lights[0].Commit();
    device.Lights[0].Enabled = true;
}
```

This simply sets the camera and our light up. It's not very efficient to call this every single frame, but for clarity, we will do so for this example. Of course, we should be checking to see whether our graphics card supports directional lighting as well, but for this simple example, we will assume it exists as well.

Instead of using a single LoadMesh call (like we've done before) to get our mesh ready, replace the call to that method with the following few calls:

```
// Create our patch mesh
CreatePatchMesh(filename, tessLevel);

// Create our font
```

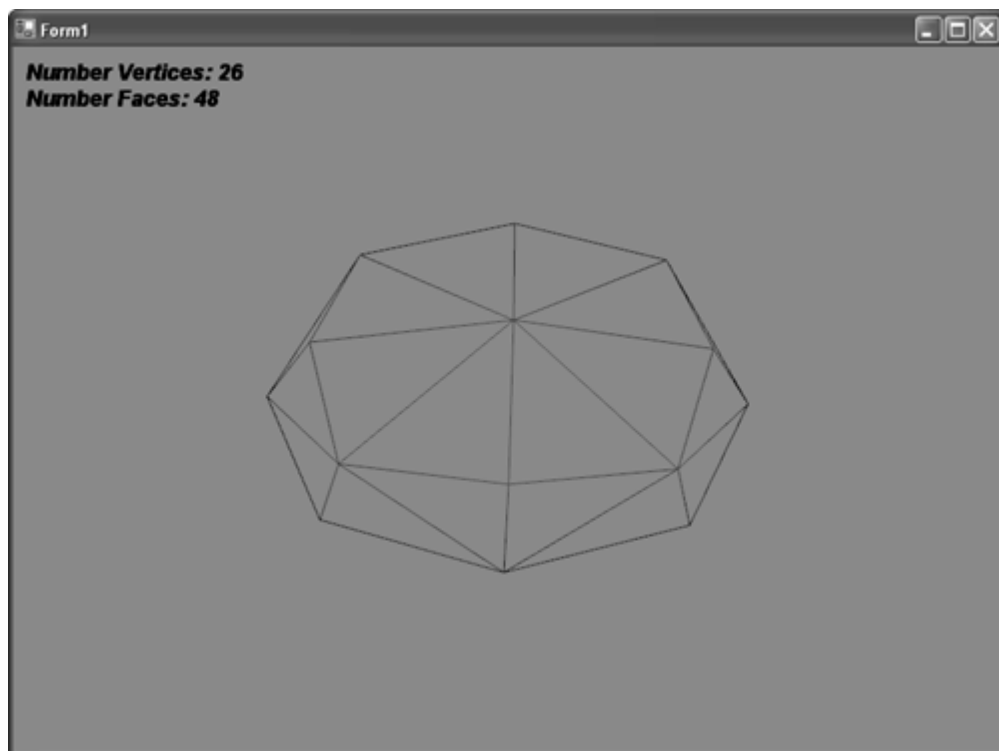
[\[Team LiB\]](#)

[\[Team LiB\]](#)

Seeing the Tessellation Levels

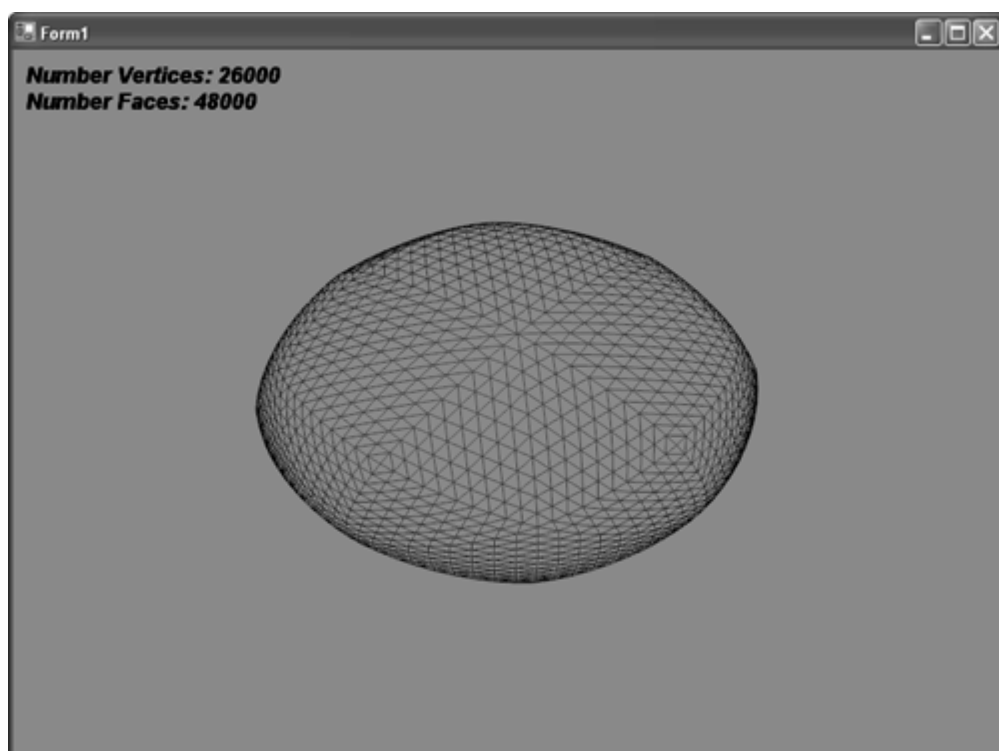
You can see the difference in the following figures:

Figure 9.2. A very low-detail sphere.



As you can see here, this "sphere" doesn't look much like a sphere. Hardly any triangles are being drawn, it looks all blocky in solid fill modes, and it's just not very realistic. However, if we just split each triangle into multiple triangles (subdivide) a few times, it looks much better.

Figure 9.3. Adding detail to the sphere.



The sphere looks much more like a sphere now than it did before. Each triangle has been tessellated quite a few times, and the effect makes the sphere look much more realistic.

[\[Team LiB\]](#)

In Brief

We covered the rest of the mesh types in this chapter, including

- Simplification meshes
- Progressive meshes
- Patch meshes

In our next chapter, we will explore some of the other classes in the extensions library, including lines, environment maps, and rendering to textures.

Chapter 10. Using the Helper Classes

IN THIS CHAPTER

-

- [Drawing Lines](#)

-

- [Drawing Text](#)

-

- [Rendering to Surfaces](#)

-

- [Rendering Environment Maps](#)

-

- [In Brief](#)

[\[Team LiB\]](#)

Drawing Lines

Drawing simple line types was actually covered back in [Chapter 4](#), "More Rendering Techniques," when dealing with primitive types, as there is a primitive type for a line list or a line strip that can be easily used. However, when drawing these types of lines, you cannot have lines with varying widths, or lines that are anti-aliased (unless the whole scene is).

Depending on the type of application you are trying to write, drawing lines can be a very common occurrence, or something that you simply never have to do. However, with the ease of the Line class, it's a simple matter to draw them anytime they're necessary. To show the simplicity of drawing some lines, we will write a quick example to draw a random number of lines on the screen continuously.

Go ahead and create a new project, and get it ready for writing some Direct3D code. This includes adding the references to the assemblies, as well as including the using clauses for the namespaces. Go ahead and create a private variable for the device, and set the window style for our rendering as well. No need to put the code necessary for these items here, as we've done them quite a few times by now.

Lines can be drawn in either pre-transformed coordinates (that is, screen coordinates) or in world space. For this example, we will only care about the lines drawn in screen space, so our device will not need to have a depth buffer. Use the method found in [Listing 10.1](#) to initialize the graphics.

Listing 10.1 Initializing Your Device

```
public void InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;

    // Create our device
    device = new Device(0, DeviceType.Hardware, this,
        CreateFlags.SoftwareVertexProcessing, presentParams);
}
```

This code is just like we've seen before, only without the depth buffer. Now, our rendering function will be the painting override just like we've done for each example thus far:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    device.Clear(ClearFlags.Target, Color.Black, 1.0f, 0);

    device.BeginScene();
    //Draw some lines
    DrawRandomLines();
    device.EndScene();
    device.Present();

    // Let them show for a few seconds
    System.Threading.Thread.Sleep(250);
    this.Invalidate();
}
```

Once again, there is nothing overly spectacular here. We clear our device, begin the scene, make our call to draw some lines, end the scene, and present it. Finally, we sleep for a quarter of a second so that we can actually see the lines being drawn, and start the whole loop over again. Naturally, our draw lines method hasn't been defined yet. Use the one found in [Listing 10.2](#).

Listing 10.2 Drawing Random Lines

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Drawing Text

Again, drawing text is something that has already been done. However you just did the bare minimum before; we will delve a little deeper into the methods now. In the previous examples, there has been a Font object both in the Microsoft.DirectX.Direct3D namespace, as well as the System.Drawing namespace. You must distinguish between them, so in order to help facilitate this, you should use the following using clause:

```
using Direct3D = Microsoft.DirectX.Direct3D;
```

This will allow us to use the Direct3D alias instead of having to type the entire namespace out. With that, let's go ahead and create a new project for our text drawing example. Make sure you add the references and the using clauses to the project (including the previous one). Also, ensure that you have a private variable for our device, and the window style set correctly for our rendering. With those items done, add the following variables:

```
private Direct3D.Font font = null;
private Mesh mesh = null;
private Material meshMaterial;
private float angle = 0.0f;
```

Here we are declaring a font that will allow us to draw text on the screen, as well as a mesh and that mesh's material. The mesh will actually be simple text that has been extruded and is a 3D mesh rather than the simple 2D text we'll be drawing as well. The angle parameter we've used before will allow the 3D text to rotate around based on frame rate. We now need to initialize the graphics with the method found in [Listing 10.3](#).

Listing 10.3 Initializing Your Graphics for Font Drawing

```
public void InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;

    // Create our device
    device = new Device(0, DeviceType.Hardware, this,
        CreateFlags.SoftwareVertexProcessing, presentParams);
    device.DeviceReset += new System.EventHandler(this.OnDeviceReset);
    OnDeviceReset(device, null);

    // What font do we want to use?
    System.Drawing.Font localFont = new System.Drawing.Font
        ("Arial", 14.0f, FontStyle.Italic);

    // Create an extruded version of this font
    mesh = Mesh.TextFromFont(device, localFont, "Managed DirectX",
        0.001f, 0.4f);

    // Create a material for our text mesh
    meshMaterial = new Material();
    meshMaterial.Diffuse = Color.Peru;

    // Create a font we can draw with
    font = new Direct3D.Font(device, localFont);
}
```

We create a hardware device with a depth buffer here, and hook the DeviceReset event. Since we will only need to

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering to Surfaces

Have you ever been playing a game, such as a racing game, and seen how you can turn on a "rear-view" mirror? Possibly you've played one of the kart racing games where there is a billboard showing the race in the background? These effects are normally achieved by rendering the same scene (most times with a different camera) into a texture. Actually, despite sounding somewhat difficult, this is in reality quite simple to do. You can once again take the example for loading a mesh from a file from [Chapter 5](#), "Rendering with Meshes," and update it for this example.

We will naturally need to declare a new texture that we will be rendering to, among other things, so add the following declarations:

```
private Texture renderTexture = null;
private Surface renderSurface = null;
private RenderToSurface rts = null;
private const int RenderSurfaceSize = 128;
```

Here we have declared the texture we will be rendering to, the actual surface we will be rendering to (which we will actually just get from the texture), and the helper class for rendering to a surface. We've also declared the size of the surfaces we'll be creating. In the InitializeGraphics method, we will want to hook the device reset event to give us a place to create our textures and surfaces, so replace the device constructor call with this:

```
device = new Device(0, DeviceType.Hardware, this,
    CreateFlags.SoftwareVertexProcessing, presentParams);
device.DeviceReset += new EventHandler(OnDeviceReset);
OnDeviceReset(device, null);
```

Now, add the actual event handler found in [Listing 10.5](#).

Listing 10.5 Setting Default Options on Your Device

```
private void OnDeviceReset(object sender, EventArgs e)
{
    Device dev = (Device)sender;

    if (dev.DeviceCaps.VertexProcessingCaps.SupportsDirectionalLights)
    {
        uint maxLights = (uint)dev.DeviceCaps.MaxActiveLights;
        if (maxLights > 0)
        {
            dev.Lights[0].Type = LightType.Directional;
            dev.Lights[0].Diffuse = Color.White;
            dev.Lights[0].Direction = new Vector3(0, -1, -1);
            dev.Lights[0].Commit();
            dev.Lights[0].Enabled = true;
        }

        if (maxLights > 1)
        {
            dev.Lights[1].Type = LightType.Directional;
            dev.Lights[1].Diffuse = Color.White;
            dev.Lights[1].Direction = new Vector3(0, -1, 1);
            dev.Lights[1].Commit();
            dev.Lights[1].Enabled = true;
        }
    }

    rts = new RenderToSurface(dev, RenderSurfaceSize, RenderSurfaceSize,
        Format.X8R8G8B8, true, DepthFormat.D16);

    renderTexture = new Texture(dev, RenderSurfaceSize, RenderSurfaceSize, 1,
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering Environment Maps

Environment mapping is a technique that can be used to simulate a highly reflective surface. You may see this effect in racing games, where a bright shiny car "reflects" the clouds above it, or where a "new" ice hockey rink seems to reflect the players as they skate on it. A common way to implement these environment maps is to use a cube texture (a six-sided texture), and we will show this technique now.

Before we begin looking at this code specifically, you will once again need to create a new project and get it ready for coding (including adding the references, the device variable, and setting the window style). We will also need two of the meshes that ship with the DirectX SDK, the car model and the skybox2 model (and associated textures). Once that has been done, add the following variable declarations:

```
private Mesh skybox = null;
private Material[] skyboxMaterials;
private Texture[] skyboxTextures;

private Mesh car = null;
private Material[] carMaterials;
private Texture[] carTextures;

private CubeTexture environment = null;
private RenderToEnvironmentMap rte = null;
private const int CubeMapSize = 128;
private readonly Matrix ViewMatrix = Matrix.Translation(0.0f, 0.0f, 13.0f);
```

Here we've declared the two meshes we will need to draw, the sky box (our "environment"), and the object we want to reflect this environment on[md]in our case, the car. We also need to have the cube texture that will store the environment and the helper class for rendering the environment map.

Since not all cards support cube textures, we will need to detect whether your card supports these before we can continue with the example. You will need a new graphics initialization method to handle this. Use the one found in [Listing 10.7](#).

Listing 10.7 Initializing Your Device for Environment Mapping

```
public bool InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;

    // Create our device
    device = new Device(0, DeviceType.Hardware, this,
        CreateFlags.SoftwareVertexProcessing, presentParams);
    device.DeviceReset += new EventHandler(OnDeviceReset);
    OnDeviceReset(device, null);

    // Do we support cube maps?
    if (!device.DeviceCaps.TextureCaps.SupportsCubeMap)
        return false;

    // Load our meshes
    skybox = LoadMesh(@"..\..\skybox2.x", ref skyboxMaterials,
        ref skyboxTextures);
    car = LoadMesh(@"..\..\car.x", ref carMaterials, ref carTextures);

    return true;
}
```

[\[Team LiB\]](#)

In Brief

- - Drawing lines
- - Drawing text
- - Rendering to textures
- - Rendering environment maps

In our next chapter, we will get into advanced graphics concepts and begin to talk about vertex and pixel shaders.

Part III: Advanced Graphics Concepts

11 [Introducing the Programmable Pipeline with the High Level Shader Language](#)

12 [Using the High Level Shader Language](#)

13 [Rendering Skeletal Animation](#)

Chapter 11. Introducing the Programmable Pipeline with the High Level Shader Language

IN THIS CHAPTER

-

- [Rendering a Single Triangle Without Using the Fixed-function Pipeline](#)

-

- [Rendering Shader Programs with Techniques](#)

-

- [Rendering Meshes Using the Programmable Pipeline](#)

-

- [Using HLSL to Write a Pixel Shader](#)

-

- [In Brief](#)

Up to this point, you've been using the fixed-function pipeline to do all of your rendering. Back in the "old days" (before the release of DirectX 8.0) this was the only way to render anything. The fixed-function pipeline is essentially a set of rules and behaviors that govern how particular types of data will be rendered. While sufficient for many things, it lacks the control some developers need to fully take advantage of the features they plan on using. For example, all lighting in the fixed-function pipeline is done per vertex, rather than per pixel. With the rendering pipeline "fixed," there was no way to change the behavior of many rendering options, such as lights.

Enter DirectX 8.0 and the programmable pipeline. A revolutionary feature that first shipped with this release allowed the developers to control most every aspect of the pipeline. They could control the vertex processing with a new feature called "Vertex Shaders," and they could control the pixel processing with the aptly named "Pixel Shaders." These shader programs were quite powerful, but they were also not very easy to use. The syntax of the language resembled assembly, and it became apparent that something easier was needed.

In DirectX 9 the High Level Shader Language (HLSL) was released. HLSL was a simple C-like language that could be compiled into real shader code, but was much easier for the developers to read, maintain, and author. The full power of the programmable pipeline was still available in a much easier to use package. In this chapter the basic features of the High Level Shading Language will be covered, including

-

- Using the programmable pipeline.

-

- Transforming vertices.

-

- Using pixel shaders.

[\[Team LiB\]](#)

Rendering a Single Triangle Without Using the Fixed-function Pipeline

What exactly is the "fixed-function pipeline" that has been mentioned several times so far in this chapter? The fixed-function pipeline controls exactly how vertices are rendered, how they are transformed, how they are lit, everything. When you set the vertex format property on the device, you are telling the device that you want to render these vertices in a certain way, based on what the vertex format is.

One of the major drawbacks of this design is that for every single feature that a graphics card needs to have exposed, a corresponding fixed-function API must be designed and implemented. With the power of graphics hardware increasing quite rapidly (quicker than CPU chips for example), the number of these APIs would quickly get out of hand. Even if it were ideal to have this large number of hard-to-understand APIs, there is still the underlying issue of the developer not really knowing what's going on under the covers. If there's one thing that most developers (particularly game developers) want, it's complete control.

One of the first applications this book discussed was a single triangle spinning around the screen. This was quite simple to do using the fixed-function pipeline, but now you will see what it would take to use the programmable pipeline instead. Go ahead and create a new windows form application and get it ready for this graphics program by declaring the device variable, as well as the window style. You will then need the following variables added:

```
private VertexBuffer vb = null;
private Effect effect = null;
private VertexDeclaration decl = null;

// Our matrices
private Matrix worldMatrix;
private Matrix viewMatrix;
private Matrix projMatrix;
private float angle = 0.0f;
```

Naturally, you will store the vertex data in the vertex buffer. Since you will not be using the transforms from the device, you will also need to store each of these transformation matrices to use in the programmable pipeline. The second and third variables here are completely new, though. The Effect object is the main object you will use when dealing with HLSL. The vertex declaration class is similar in use to the vertex format enumeration in the fixed-function pipeline. It informs the Direct3D runtime about the size and types of data it will be reading from the vertex buffer.

Since this application will be using a relatively new feature of graphics cards (namely the programmable pipeline), it's entirely possible that your graphics card will not support it. If that is the case, you will want to switch to the reference device that ships with the DirectX SDK. The reference device implements the entire API in software, albeit very slowly. You will need a more robust initialization routine for this, so use the one found in [Listing 11.1](#).

Listing 11.1 Initializing Graphics with a Fallback

```
public bool InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;

    bool canDoShaders = true;
    // Does a hardware device support shaders?
    Caps hardware = Manager.GetDeviceCaps(0, DeviceType.Hardware);
    if (hardware.VertexShaderVersion >= new Version(1, 1))
    {
        // Default to software processing
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering Shader Programs with Techniques

Now that you have the vertex program written, how can you actually use it? There isn't an "entry point" into the vertex program, just a method somewhere. How do you get this method called, though? Quite simply, you can use something called a "technique." A technique consists of one to many passes, where each pass can set device state and set the vertex and pixel shaders to use your HLSL code. Look at the technique you can use for this application. Add this to your simple.fx file:

```
technique TransformDiffuse
{
    pass P0
    {
        CullMode = None;

        // shaders
        VertexShader = compile vs_1_1 Transform();
        PixelShader   = NULL;
    }
}
```

Here, you've declared a technique called TransformDiffuse, since this method will transform the vertices and add a diffuse color. This name is purely decorative and has no real purpose. In this technique, one pass has been defined. The cull mode is set to none, so you can see both the front and back of the single triangle. If you remember back to the original sample, you had to do this in the actual C# code. However, passes can be thought of as an excellent place to store (and update) device state, so it makes plenty of sense to add this here.

Next, you will want to use the vertex program for the vertices that will be rendered here, so the method is compiled using the vs_1_1 target. This target string should be used based on what you know your device supports. Since you already know your device supports at least vertex shader 1.1 (based on the capabilities check), you can use this target. If you were using vertex shader 2.0, the target would be vs_2_0. Pixel shader targets are defined such as ps_2_0.

Since there are no pixel programs, you can set the pixel shader member to null here, and that's the end of the pass. Nothing much going on, but now you will need to update the actual C# code to use this technique and pass.

First, you will use the Effect object that has been declared, since this is the main object used when dealing with your HLSL code. Add the following to the InitializeGraphics method (after device creation):

```
// Create our effect
effect = Effect.FromFile(device, @"..\..\simple.fx", null, ShaderFlags.None,
    null);
effect.Technique = "TransformDiffuse";
```

As you see, you are creating the effect object from a file (the one you've just created). You also set the technique here since only one technique will be used for the entire program. This effect object will be used for all of your dealings with the HLSL code you've written thus far. If you remember, you had two variables that needed to be updated each frame. Add the following code to the end of the UpdateWorld method:

```
Matrix worldViewProj = worldMatrix * viewMatrix * projMatrix;
effect.SetValue("Time", (float)Math.Sin(angle / 5.0f));
effect.SetValue("WorldViewProj", worldViewProj);
```

Here, you take the stored matrices and combine them, and update the variables in the HLSL code. This method is called every frame, so the variables will be updated this often as well. The only thing you have left to do is actually update the rendering to draw using the vertex program. Update your DrawPrimitives call as follows:

```
int numPasses = effect.Begin(0);
```


[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering Meshes Using the Programmable Pipeline

The simple triangle example is just that: simple. It's rare that you would use a vertex program on a single triangle in a real-world example, so instead of using this example, you should switch to rendering an entire mesh. Go ahead and replace the variables for the vertex declaration and the vertex buffer with this one for the mesh:

```
private Mesh mesh = null;
```

One nice thing about the mesh objects is that when they are rendered, they will automatically set the vertex declaration, so you don't need to worry about it. Naturally, you'll need to remove all instances in the code that dealt with either the vertex buffer or vertex declaration, as well. You will also need to update the initialize method to create the mesh. After the device and effect objects have been created, add the following code:

```
// Create our cylinder
mesh = Mesh.Cylinder(device, 1.5f, 1.5f, 1.5f, 36, 36);
```

Finally, you'll need to change the drawing method. Replace the DrawPrimitives call with the DrawSubset method on the mesh:

```
mesh.DrawSubset(0);
```

As you can see, there's nothing really special you need to do. Running this application now will cause a (very flat-looking) cylinder to be rendered with the same cycling colors our triangle had. The cylinder doesn't look much like a cylinder, though, since it's shaded with the same color everywhere. If you remember when this was discussed before, you simply added a directional light, and everything looked wonderful.

Using the lighting properties that you used before would defeat the purpose of this chapter, since those lights are implemented as part of the fixed-function pipeline. Since the goal is to show off the programmable pipeline, you can just update the HLSL code to render a single white directional light. First, you'll need a variable to describe the direction of the light. Add the following variable to your HLSL code:

```
// The direction of the light in world space
float3 LightDir = {0.0f, 0.0f, -1.0f};
```

Now you can simply replace the existing method with the following:

```
// Transform our coordinates into world space
VS_OUTPUT Transform(
    float4 Pos : POSITION,
    float3 Normal : NORMAL )
{
    // Declare our return variable
    VS_OUTPUT Out = (VS_OUTPUT)0;

    // Transform the normal into the same coord system
    float4 transformedNormal = mul(Normal, WorldViewProj);

    // Set our color
    Out.diff.rgba = 1.0f;
    Out.diff *= dot(transformedNormal, LightDir);

    // Transform our position
    Out.pos = mul(Pos, WorldViewProj);

    // Return
    return Out;
}
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using HLSL to Write a Pixel Shader

So far you've manipulated the vertices and written vertex programs, but that's only half the fun when dealing with the programmable pipeline. What about the color of each individual pixel? For this example, we will take the MeshFile example you wrote back in [Chapter 5](#), "Rendering with Meshes," and update it to use the programmable pipeline instead. This one is being used because not only does it render vertices, but they are colored from a texture where a pixel shader could be used.

After loading this project, you will need to make a few changes to the source to begin using the programmable pipeline. First, you will need the variables to store the transformations and effect object, so add those:

```
private Effect effect = null;
// Matrices
private Matrix worldMatrix;
private Matrix viewMatrix;
private Matrix projMatrix;
```

You will also need to update the initialization method to handle picking the reference device if shader support isn't available. Replace the method with the method found in [Listing 11.2](#).

Listing 11.2 Initializing Graphics with Fallback

```
public bool InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;

    bool canDoShaders = true;
    // Does a hardware device support shaders?
    Caps hardware = Manager.GetDeviceCaps(0, DeviceType.Hardware);
    if ((hardware.VertexShaderVersion >= new Version(1, 1)) &&
        (hardware.PixelShaderVersion >= new Version(1, 1)))
    {
        // Default to software processing
        CreateFlags flags = CreateFlags.SoftwareVertexProcessing;

        // Use hardware if it's available
        if (hardware.DeviceCaps.SupportsHardwareTransformAndLight)
            flags = CreateFlags.HardwareVertexProcessing;

        // Use pure if it's available
        if (hardware.DeviceCaps.SupportsPureDevice)
            flags |= CreateFlags.PureDevice;

        // Yes, Create our device
        device = new Device(0, DeviceType.Hardware, this, flags, presentParams);
    }
    else
    {
        // No shader support
        canDoShaders = false;

        // Create a reference device
        device = new Device(0, DeviceType.Reference, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
    }
}
```

[\[Team LiB\]](#)

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

In Brief

- - Using the programmable pipeline.
- - Transforming vertices.
- - Using pixel shaders.

In our next chapter, we will get into more advanced HLSL concepts.

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Chapter 12. Using the High Level Shader Language

IN THIS CHAPTER

- [Using Simple Formulas to Simulate Animation](#)
- [Determining Color by Blending Textures](#)
- [Lighting Textures](#)
- [Adding Specular Highlights](#)
- [In Brief](#)

In dealing with the programmable pipeline and the High Level Shader Language thus far, the possibilities of what can be accomplished have barely been touched. The items completed up to this point include basic functionality that exists in the fixed-function pipeline, but nothing more advanced. With the open-ended nature of the programmable pipeline, these advanced features that don't exist in the fixed-function pipeline should be the tasks you try to accomplish.

During this chapter you will look a little more deeply into HLSL, and how you can use this technology to implement features that are impossible (or difficult) in the fixed-function pipeline. Advanced features that will be discussed in this chapter include

- Simple vertex animation
- Simple color animation
- Texture blending
- Lighting models with textures

The difference between per-vertex lighting and per-pixel lighting

[\[Team LiB\]](#)

Using Simple Formulas to Simulate Animation

Rather than wasting space here duplicating the same code that has already been covered, you should copy the code that was used to render the cylinder in the previous chapter. This code will be modified slightly to show how easily simple animation can be added once you've begun using the programmable pipeline. This example will be modified to allow the mesh to "wobble" as it's being rendered.

Once the code has been loaded from the previous example, the first thing you'll want to do is to switch to using a sphere rather than the cylinder. It's not that this update wouldn't work with the cylinder; it would. However, the sphere happens to just look better. Replace the cylinder creation code with this:

```
// Create our sphere
mesh = Mesh.Sphere(device, 3.0f, 36, 36);
```

Since the sphere's size is larger than the cylinder that was being rendered before, you will also need to update the view matrix. You'll need to move the camera position back some, as in the following:

```
viewMatrix = Matrix.LookAtLH(new Vector3(0,0, 9.0f), new Vector3(),
    new Vector3(0,1,0));
```

With that, you should be able to run the application and see a (quite boring) spinning sphere being lit by a single directional light. What do you think could be done to make the sphere perform some simple animation? In your shader code, replace the single line that does transformation with the following code:

```
// Store our local position
float4 tempPos = Pos;
// Make the sphere 'wobble' some
tempPos.y += cos(Pos + (Time * 2.0f));
// Transform our position
Out.pos = mul(tempPos, WorldViewProj);
```

You'll notice that the Time variable is used in this example once more. Since that variable wasn't used in the example this was based on, you will need to add it to your shader code:

```
float Time = 0.0f;
```

In addition, you will need to update the variable every frame, so in your main code at the end of the UpdateWorld method, add the following code to update this variables value:

```
effect.SetValue("Time", angle);
```

So what exactly does the shader code accomplish? First, the input position of the vertex is stored in a temporary variable. Then the y component of this variable is modulated by adding the cosine of the current position of the vertex added with the constantly updating Time variable. Since the cosine method will always be within the range of -1 to 1, the effect will repeat indefinitely. With the vertex now being updated to simulate our animation, the temporary variable is used to perform the transformation rather than the original input position.

While this updated code does look a lot nicer than the boring old sphere, there's yet another easy thing you can do to make it look even better. Why not add some "animating" color as well? Replace the Out.diff initialization code with the following:

```
// Set our color
Out.diff = sin(Pos + Time);
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Determining Color by Blending Textures

In the real world, not very many applications will use simple pseudo-randomly generated colors when rendering models. Models are normally generated with highly detailed textures, and occasionally quite a few of them. What if there were a scenario where you needed to blend two or more textures for your model? Sure, you could do that with the fixed-function pipeline, but what if the model was something like "tiny.x", which most of the code in this book uses? This model only uses a single texture, so how will you use two textures blended together?

UNDERSTANDING THE PIXEL SHADER INSTRUCTION SET LIMIT

Since you will be manipulating a pixel shader in this code, one thing you may notice is that the amount of available "instructions" you can use in a 1.1 pixel shader code segment is extremely limited. For pixel shaders before version 1.4, you are limited to 12 instructions for the entire shader program, which obviously isn't very much. For version 1.4, the limit is raised to 28 (in two phases) but that's still not much. You are also limited to 8 constants in these shader programs. Pixel shader version 2.0 and higher has the capability to perform much more complex operations and have much higher instruction and constant counts. You can check the capabilities for your card to find out how many of each are supported.

Once again, for brevity, you should use the code example written in the previous chapter that rendered the textured mesh. This will focus on adding a second texture and writing the code to blend between the two textures to calculate a final color.

Knowing this limitation, the code for the texture blending here will be shown in two separate ways: one with an extremely simple pixel shader that can be used on cards that support versions 1.1, and another more complex version that can be used on cards that support version 2.0 and above. You will need a relatively new graphics card in order to support this feature set.

Naturally, you should start with the 1.1 version since it's obviously simpler. Before you get to the actual shader code, though, you will need to make a few updates to the actual code. For one thing, you need a second texture to do the blending. Add the following texture variable for your second texture into your class:

```
private Texture skyTexture;
```

Also, since you know that the model you will be loading only has one material and one texture, you may want to remove the need for using arrays of materials and textures. The code included on the CD has done this, and the remaining code for this example will assume it has been completed.

You will now need to create a second texture that will be used to blend into your model. This example will assume that you've copied the skybox_top.JPG texture that ships with the DirectX SDK into your application folder. Feel free to use a different texture that suits your mood if you want. Once you have decided on the second texture, you will need to create it. The perfect place would be where the original texture is created in the LoadMesh method. Replace the single texture creation line with this:

```
// We have a texture, try to load it
meshTexture = TextureLoader.FromFile(device, @"..\..\\" +
    mtrl[i].TextureFilename);
skyTexture = TextureLoader.FromFile(device, @"..\..\skybox_top.JPG");
```

If you've chosen a different texture file, you will want to use that filename instead. With the second texture created, the code to do the blending between the two can now be written. Add the following variables to your shader code:

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Lighting Textures

In the previous chapter, simple directional lighting was discussed for a cylinder. The cylinder that was used for that example wasn't textured, and so far the lighting calculations have only been performed in the vertex programs. You can just as easily perform these lighting operations in your pixel shader code as well.

Like the previous example, we will use the simple mesh rendering example from the previous chapter as our starting point. In this section you will take that existing sample and add a colored light to the scene in our pixel shader.

The first step to making this update will be to add the following declarations to your shader code:

```
float4x4 WorldMatrix : WORLD;
float4 DiffuseDirection;

Texture meshTexture;
sampler TextureSampler = sampler_state { texture = <meshTexture>; mipfilter = LINEAR; };

struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float2 TexCoord : TEXCOORD0;
    float3 Light : TEXCOORD1;
    float3 Normal : TEXCOORD2;
};
```

You'll notice that not only are you storing the combined world, view, and projection matrices, but there is another variable to store the world matrix itself. You'll understand why this is necessary when the shader code itself is discussed. There is also a variable to hold the direction of the light. Much like the previous example, there is a texture and a sampler state stored for reading the texture.

The output structure is quite different than anything seen yet. Not only are the position and texture coordinates passed on, but there are two new texture coordinate members that will be passed on as well. In reality, these members will not really be texture coordinates at all; they are just what will be used to transfer the necessary data from the vertex shader to the pixel shader.

You'll also need to actually update the values of these members. In your mesh drawing routine, you are already setting the WorldViewProj variable, so that is the perfect place for the following:

```
effect.SetValue("WorldMatrix", worldMatrix);
effect.SetValue("DiffuseDirection", new Vector4(0, 0, 1, 1));
```

The texture variable will also need to be set. In your initialization routine after your mesh (and texture) has been loaded, add the following:

```
effect.SetValue("meshTexture", meshTexture);
```

With those simple items out of the way, you're ready to update the shader code to perform the lighting. You should pick a color other than white for this light so that it's obvious the work is being done. Before looking at the pixel shader code, you should examine the vertex shader:

```
// Transform our coordinates into world space
VS_OUTPUT Transform(
    float4 inputPosition : POSITION,
    float3 inputNormal : NORMAL,
    float2 inputTexCoord : TEXCOORD0
)
{
    //Declare our output structure
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Adding Specular Highlights

Another type of lighting that hasn't been discussed yet is specular highlighting. Specular highlighting makes an object appear "shiny" and much more realistic when being rendered. It's possible to use the lighting objects in the fixed-function pipeline to enable specular highlights; however, these highlights will be calculated per-vertex. This example will show how to accomplish this with the programmable pipeline instead.

Like the other examples in this chapter, this one will be based on an earlier section of code; in this case, the lit cylinder example you wrote in the previous chapter. This code already handles the diffuse lighting case, which should stay in this updated example to show the differences between the two lighting models.

While the cylinder object will show off the features that this example is trying to, a better object to use is the teapot. Change your main code from loading a cylinder to loading a teapot:

```
// Load our mesh
mesh = Mesh.Teapot(device);
```

Since this example will also be showing multiple different effects and allowing you to switch between them, you will want to have something to notify the user of the current state. You should declare a font variable to allow the user to be notified during the rendered scene:

```
// Out font
private Direct3D.Font font = null;
```

You may as well initialize the font variable directly after your teapot has been created as well:

```
//Create our font
font = new Direct3D.Font(device, new System.Drawing.Font("Arial", 12.0f));
```

With these things out of the way, you should now add the constants and variables you will need into your shader code:

```
float4x4 WorldViewProj : WORLDVIEWPROJECTION;
float4x4 WorldMatrix : WORLD;
float4 DiffuseDirection;
float4 EyeLocation;

// Color constants
const float4 MetallicColor = { 0.8f, 0.8f, 0.8f, 1.0f };
const float4 AmbientColor = { 0.05f, 0.05f, 0.05f, 1.0f };
```

The combined world, view, and projection matrix variable has been used in each example for transforming the vertices. The single world matrix will once again be used to transform the normals of each vertex. The diffuse direction member will allow the application to define the direction of the light, rather than having it hard coded in the shader like before. The last variable is the location of the eye. Specular highlights are calculated by determining a reflection between the normal and the eye.

You'll also notice there are two constants declared. Since metal is normally shiny, the example will use a color that resembles a metallic material for our diffuse light. An ambient color is also declared, although for this example it is essentially a non-factor. It's only been included for completeness of the lighting mathematic formulas.

For this example, the per-vertex lighting output structure only needs to be concerned with the position and color of each vertex, so declare the structure as follows:

```
struct VS_OUTPUT_PER_VERTEX
{
```

[\[Team LiB\]](#)

In Brief

- - Simple vertex and color animation
- - Texture blending
- - Advanced lighting mechanisms

In the next chapter, the animation system of Managed DirectX will be discussed.

Chapter 13. Rendering Skeletal Animation

IN THIS CHAPTER

- [Creating the Frame Hierarchy](#)
- [Loading Meshes with Animation](#)
- [Rendering Animated Meshes](#)
- [In Brief](#)

You don't see modern games render a bunch of static items; instead you see rich environments that are fluid. Characters are fully animated, with much of the animation coming from a motion capture studio. Models are designed with "bones" that can be positioned and animated to simulate motion. Aside from these complex animations, there are also other simple animation types, dealing mainly with scaling, rotation, and translation. This chapter will cover rendering meshes with animation data, including

- Loading frame hierarchy
- Generating any skeletal information
- Rendering each frame
- Advancing the animation time
- Using indexed animation for better performance

[\[Team LiB\]](#)

Creating the Frame Hierarchy

Most meshes (even those not created with animation built in) have some type of hierarchy; an arm attached to the chest, a finger attached to the arm. In the Direct3D extensions library, you can load these hierarchies and maintain them yourself.

In order to store this information, there are two abstract classes that will be used to maintain the hierarchy: `Frame` and `MeshContainer`. Each `Frame` can contain zero to many sibling frames, as well as a potential child frame. Each frame can also contain zero to many mesh containers.

Before we get into all that, though, a new project will need to be created. Go ahead and get it created with the default options, the device variable declared, and the window style set, just like you've done with all of the other Direct3D applications.

Declare the following variables in your application:

```
private AnimationRootFrame rootFrame;
private Vector3 objectCenter;
private float objectRadius;
private float elapsedTime;
```

The `AnimationRootFrame` structure will hold the root frame of the hierarchy tree, as well as the animation controller class. The next two variables will hold the center of the loaded mesh, with the other storing the radius of the bounding sphere that can encase the full mesh. These will be used to position the camera so that the entire model will be seen. The last variable will store the elapsed time of the application, which will be used to update the animation.

Since each animated mesh is different, Direct3D will not force a particular way to load the hierarchy on you. Instead, you will need to derive a new class from the `AllocateHierarchy` object. Add the following class to your application:

```
public class AllocateHierarchyDerived : AllocateHierarchy
{
    Form1 app = null;
    /// <summary>
    /// Create new instance of this class
    /// </summary>
    /// <param name="parent">Parent of this class</param>
    public AllocateHierarchyDerived(Form1 parent)
    {
        app = parent;
    }
}
```

So far, the class doesn't do anything. The constructor stores an instance of the main form so that it can be used to call methods later, but right now the application won't even compile. There are two methods that must be implemented in this class before it can be compiled, since they are marked abstract. The two methods are used to create the frames and mesh containers that will be used to store the hierarchy. Since both of these classes are marked abstract as well, they cannot be created, so you will need to add new derived versions of the classes to your application. There will be extra information stored for each class to help our rendering later as well. Add the following class to your application:

```
public class FrameDerived : Frame
{
    // Store the combined transformation matrix
    private Matrix combined = Matrix.Identity;
    public Matrix CombinedTransformationMatrix
    {
        get { return combined; } set { combined = value; }
    }
}
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Loading Meshes with Animation

The basic hierarchy of the mesh has been established. Now, all you need to do is write some code to actually use it. Before you do that, though, you will need to initialize your graphics device. You should use a method similar to the programmable pipeline examples earlier, so use the method found in [Listing 13.4](#).

Listing 13.4 Initializing Your Graphics

```
public bool InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();

    presentParams.Windowed = true;
    presentParams.SwapEffect = SwapEffect.Discard;
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
    presentParams.EnableAutoDepthStencil = true;

    bool canDoHardwareSkinning = true;
    // Does a hardware device support shaders?
    Caps hardware = Manager.GetDeviceCaps(0, DeviceType.Hardware);
    // We will need at least four blend matrices
    if (hardware.MaxVertexBlendMatrices >= 4)
    {
        // Default to software processing
        CreateFlags flags = CreateFlags.SoftwareVertexProcessing;

        // Use hardware if it's available
        if (hardware.DeviceCaps.SupportsHardwareTransformAndLight)
            flags = CreateFlags.HardwareVertexProcessing;

        // Use pure if it's available
        if (hardware.DeviceCaps.SupportsPureDevice)
            flags |= CreateFlags.PureDevice;

        // Yes, Create our device
        device = new Device(0, DeviceType.Hardware, this, flags, presentParams);
    }
    else
    {
        // No shader support
        canDoHardwareSkinning = false;

        // Create a reference device
        device = new Device(0, DeviceType.Reference, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
    }

    // Create the animation
    CreateAnimation(@"..\..\tiny.x", presentParams);

    // Hook the device reset event
    device.DeviceReset += new EventHandler(OnDeviceReset);
    OnDeviceReset(device, null);

    return canDoHardwareSkinning;
}
```

Once again, you want to try to run in hardware first, and then fallback to the reference device if you must. The animation file used for this example will require at least four vertex blend matrices in order to render completely in hardware, so that's what the code will check for. Assuming it can, the best possible device will be created; otherwise, the reference device is created.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering Animated Meshes

What you want to do now is to render your animated character. The rendering method itself appears deceptively simple, so add the code from [Listing 13.8](#) to your class now.

Listing 13.8 Rendering Your Animated Character

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    ProcessNextFrame();

    device.Clear(ClearFlags.Target | ClearFlags.ZBuffer,
        Color.CornflowerBlue, 1.0f, 0);

    device.BeginScene();

    // Draw our root frame
    DrawFrame((FrameDerived)rootFrame.FrameHierarchy);

    device.EndScene();

    device.Present();

    this.Invalidate();
}
```

As you can see, the method is quite simple. Process the next frame, clear the device, draw the root frame, and you're done. A lot of things need to happen in these steps, though. First, look at the things that need to be done to process the next frame:

```
private void ProcessNextFrame()
{
    // Get the current elapsed time
    elapsedTime = DXUtil.Timer(DirectXTimer.GetElapsedTime);

    // Set the world matrix
    Matrix worldMatrix = Matrix.Translation(objectCenter);
    device.Transform.World = worldMatrix;

    if (rootFrame.AnimationController != null)
        rootFrame.AnimationController.AdvanceTime(elapsedTime, null);

    UpdateFrameMatrices((FrameDerived)rootFrame.FrameHierarchy, worldMatrix);
}
```

First, the current elapsed time is stored. Next, the world matrix for the root frame is created. Simply translate to the object's center, and update the device. Assuming this mesh has animation, you should advance the time, using the stored elapsed time. Finally, each of combined transformation matrices needs to be updated. Look at the following method:

```
private void UpdateFrameMatrices(FrameDerived frame, Matrix parentMatrix)
{
    frame.CombinedTransformationMatrix = frame.TransformationMatrix *
        parentMatrix;

    if (frame.FrameSibling != null)
    {
        UpdateFrameMatrices((FrameDerived)frame.FrameSibling, parentMatrix);
    }

    if (frame.FrameFirstChild != null)
    {

```

[\[Team LiB\]](#)

In Brief

In this chapter we covered the basics of animation, specifically skeletal animation. Topics included

- - Creating and loading a frame hierarchy
- - Walking this hierarchy to update and render the meshes
- - Using the animation controller
- - Converting our animated character to an indexed mesh

In the next chapter, we will discuss how to use sound in your applications.

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Part IV: Sound and Input

14. [Discovering the Wonders of Sound](#)

15. [Controlling User Input](#)

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Chapter 14. Discovering the Wonders of Sound

IN THIS CHAPTER

-

- [Including the Sound Namespace](#)

-

- [Loading and Playing a Static Sound](#)

-

- [Using Sounds in 3D](#)

-

- [Manipulating the Listener](#)

-

- [Using Effects with Your Sounds](#)

-

- [In Brief](#)

All of the 3D graphics in the world won't make a great game. Aside from the obvious game play issues, you also need to create an atmosphere in order to be compelling. Fancy graphics and pretty environments help, but the world doesn't really come alive until you've added sound. Could you imagine playing Pac-Man without hearing the trademark "wakka wakka wakka" while you're running around? What fun would Undying be without the amazing use of creepy sounds? During the course of this chapter you'll learn how to use rich sound and include it in your applications. The topics that will be covered include

-

- Loading and playing static sounds

-

- Playing sounds in a 3D world

-

- Playing sounds with special effects

[\[Team LiB\]](#)

◀ PREVIOUS

NEXT ▶

Including the Sound Namespace

Well, it's been a while since you've needed to add any new references to one of the applications. Naturally the sound resources that you will need to use do not reside in the Direct3D namespace; you will instead need to use the Microsoft.DirectX.DirectSound namespace. For each of the examples in this chapter, you will need to add a reference to this assembly for the project, as well as add a using clause for this namespace.

[\[Team LiB\]](#)

◀ PREVIOUS

NEXT ▶

[\[Team LiB\]](#)

Loading and Playing a Static Sound

In the simplest case, what do you want to do for a sound? You want to have some type of audio data, and you want to output this audio data to all of the speakers attached to the system. Sounds easy enough, but how do you do that in a DirectX application? For the simple case just described, it is exceedingly simple.

Much like Direct3D has a Device class that controls the actual graphics hardware, DirectSound has a Device class that controls the actual sound hardware. Since both of these classes share the same name (in different namespaces), it's important to remember that if you are referencing both Direct3D and DirectSound in the same code file, you may need to fully qualify any references to a "Device" variable.

DirectSound also has buffers that are used to hold the audio data it needs to play the sounds, much like Direct3D used buffers to hold geometry or pixel data. There are only two types of buffers in DirectSound, either the stock "Buffer" object, or the more robust "SecondaryBuffer" object (which also derives from Buffer).

So since the simple case is so simple, you can just write it now, and take a look at it. First, create a new project and get the DirectSound references and namespaces in, and then add the following variable declarations:

```
private Device device = null;
private SecondaryBuffer sound = null;
```

You will have one device that is used to talk to the sound hardware, and you will have one buffer you can use to play the actual audio data that will be loaded. Now, you can write the "InitializeSound" method, which will load and play a sound. Since this method will reference an audio file, you will need to provide one. The file used in this method was copied from the DirectX SDK. You can copy this file from the included source on the CD. Add the following method:

```
public void InitializeSound()
{
    device = new Device();
    device.SetCooperativeLevel(this, CooperativeLevel.Normal);
    sound = new SecondaryBuffer(@"..\..\drumpad-crash.wav", device);
    sound.Play(0, BufferPlayFlags.Default);
}
```

As you can see, the code is quite simple. You create a new device (by using the parameter-less constructor; the other overloads for this object will be discussed later in this chapter), and then set the cooperative level on the device. I know what you're thinking: What exactly is a cooperative level?

The sound hardware in your machine is shared among many different applications. Windows may beep at you when an error has occurred. Messenger may beep when a friend has logged in, and all of these things can happen while you're listening to your favorite songs. The cooperative level is used to determine how your application behaves with others, hence its name. The choice selected here is the default option. It has the best multitasking and resource sharing behavior. You may look in the DirectX SDK documentation for details on the other priority levels; for the needs of this application, you can stick with normal.

After you have set the cooperative level, and are prepared to play nice with the rest of the system, you can create the buffer. You will load the buffer from a file (the code on the included CD uses a simple sound that ships with the DirectX SDK), and finally just play it. The first parameter of the play call is the priority, which is only valid if you've created your buffer deferred (which you haven't). You play the buffer using the default flags, and you're off. Now, you just need to actually call this method from somewhere, so how about replacing the main code with this:

```
static void Main()
{
    using (Form1 frm = new Form1())
    {
        frm.InitializeSound();
        Application.Run(frm);
    }
}
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Sounds in 3D

Many multimedia computer systems nowadays have high-end speaker systems attached to high-end audio cards. It isn't uncommon for a system to have the ability to use full surround sound, and it won't be long before 5.1 systems are the norm for computers, particularly for the gamers.

The DirectSound API already has the ability to play sounds in 3D space, and can use these features if the computer supports them today. The options are controlled with the Buffer3D object (for controlling the actual source of the sound) and the Listener3D object (for controlling the position and orientation of the listener). You'll notice that the constructors for these objects each take in a buffer that is used to create the 3D version of the audio. These buffers must be created with the Control3D flag set to true.

So what benefits do you gain from implementing 3D sound into your application? Naturally you get a richer experience. When you go to the movie theater nowadays, the sound is normally played digitally, and they always have a surround sound system. Imagine if you were playing a game, and a door opened behind you, if there were speakers behind you, you would expect the sound to come from those speakers, not the two in front of you. Games today require the full experience, and this includes 3D sound.

There are two items you can deal with when you want to manipulate the sounds in 3D. You can manipulate the actual source of the sound, or you can manipulate the "ears" that are hearing the sound. First, you can look at moving around the sound itself, by using the Buffer3D object.

This object is only designed to control the 3D settings for the buffer. All of the methods for playing the buffer, and so on, still reside on the original buffer object. The properties you can update with this object are located in [Table 14.2](#).

Table 14.2. Buffer3D Properties

PROPERTY	DESCRIPTION
ConeAngles	A read-write property that is used to set or get the angles for the inside or outside of the projection cone. Both angles are specified in degrees. Sounds within the inside cone angle are at normal volume, while sounds outside of the outside cone angle are at the outside volume.
ConeOrientation	A read-write property that controls the orientation of the sound's projection cone. DirectSound will automatically normalize the vector. The vector information used should be the center of the sound cone.
ConeOutsideVolume	A read-write property that controls the volume of sounds that lie outside the angle of the sound cone.
Deferred	A read-write Boolean property that determines if property changes will happen instantly or if they will be deferred until the listener updates settings. The default is false, and the properties are updated immediately.
MaxDistance	A read-write property that determines the maximum distance from the listener before the sound is no longer attenuated.
MinDistance	A read-write property that determines the minimum

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Manipulating the Listener

The Listener3D object is created and used much like the Buffer3D object is. However, instead of manipulating the actual sound source (of which there can be many), you are dealing directly with the listener, of which there is only one per device. For this reason, you cannot use a SecondaryBuffer when creating a listener object; you can only use a Buffer object, and then only if it is a primary buffer.

Before you write the code using the listener object, you should take a look at the properties and settings you can modify on it, much like you did with the 3D buffer object. You will find these properties in [Table 14.3](#).

Table 14.3. Listener3D Properties

PROPERTY	DESCRIPTION
CommitDeferredSettings	If any of the settings were deferred on any 3D sound buffers, or the listener, this method will commit all new values at the same time. This method will have no effect if there were no deferred changes made.
DistanceFactor	A read-write property that controls the number of meters in a vector unit.
DopplerFactor	A read-write property that controls the multiplier for the Doppler effect. Any buffer that has velocity will automatically get Doppler shift effects, and these effects are cumulative.
Deferred	A read-write Boolean property that determines if property changes will happen instantly or if they will be deferred until the listener updates settings. The default is false, and the properties are updated immediately.
Orientation	A read-write property that controls the orientation of the listener. DirectSound will automatically normalize the vector.
RolloffFactor	A read-write property that determines the rolloff factor. This factor determines the rate of attenuation over distance.
Position	A read-write property that determines the current position of the listener in world space.
Velocity	A read-write property that determines the velocity of the listener, in meters per second by default.

You should take the existing example where the sound buffer is moving, and update it to move the listener instead. You can get rid of the 3D buffer object you were using, and replace that variable with the following two:

```
private Listener3D listener = null;
private Microsoft.DirectX.DirectSound.Buffer primary = null;
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Effects with Your Sounds

Aside from 3D manipulation of sounds, another technique used to bring out the realism of the games you play would be the effects added to already existing sounds. For example, walking across a wood floor would sound totally different in a small room from the way it would sound if you were in a large empty concert hall. Yelling in a cavern may cause your voice to echo among the cavern walls. These effects are an integral part of the immersion you are trying to create when developing a game, or any good interactive multimedia application.

BUFFERS THAT SUPPORT EFFECTS

Effects can only be used on a secondary buffer. The required method to apply the effects to the buffer only resides on this object.

Luckily, the act of using these special effects is pretty straightforward and there are quite a few of these effects already built into the DirectSound API. Revisiting the old example you created that was controlling the pan, update that to control some effects. Remove the timer since you won't be using that anymore. Then change the InitializeSound method as follows:

```
public void InitializeSound()
{
    device = new Device();
    device.SetCooperativeLevel(this, CooperativeLevel.Normal);

    BufferDescription desc = new BufferDescription();
    desc.ControlEffects = true;
    desc.GlobalFocus = true;

    sound = new SecondaryBuffer(@"..\..\drumpad-crash.wav", desc, device);

    EffectDescription[] effects = new EffectDescription[1];
    effects[0].GuidEffectClass = DSoundHelper.StandardEchoGuid;
    sound.SetEffects(effects);

    sound.Play(0, BufferPlayFlags.Looping);
}
```

The big changes here are the ability to control the pan being removed, and change it to controlling the effects. Then you create an array of effect descriptions (currently with just one member), and assign that member to be the standard echo effect. You then pass in the effects that you have just created and start playing the buffer.

SETTING EFFECTS

Effects can only be set on a buffer that is stopped. If the buffer is playing when SetEffects is called, an exception will be thrown. You can use the Status property on the buffer to detect if it is playing. If it is, you can call the Stop method to ensure that it has stopped before applying the effects.

That's really all there is to setting the basic effects. If you run the application now you will be able to hear the echo that wasn't there before. It's also possible to add "stacks" of these effects on top of each other. If we wanted to have the sound played with an echo effect along with a flanger effect, followed by a distortion effect, you could update the method as follows:

```
EffectDescription[] effects = new EffectDescription[3];
effects[0].GuidEffectClass = DSoundHelper.StandardEchoGuid;
effects[1].GuidEffectClass = DSoundHelper.StandardFlangerGuid;
```

[\[Team LiB\]](#)

In Brief

- Loading and playing static sounds
- Playing sounds in a 3D world
- Playing sounds with special effects

In our next chapter, we will finally begin to look at controlling user input with the keyboard, mouse, or joystick.

Chapter 15. Controlling User Input

IN THIS CHAPTER

- [Detecting the Devices You Can Use](#)
- [Using the Keyboard Device](#)
- [Using the Mouse Device](#)
- [Using Game Pads and Joysticks for User Input](#)
- [Using Force Feedback](#)
- [In Brief](#)

Thus far, the book has covered 3D graphics in depth, and it has now covered sound as well. Aside from briefly in [Chapter 6](#), "Using Managed DirectX to Write a Game," user control hasn't been discussed at all. Even that control only dealt with the keyboard, while ignoring the mouse, or even joysticks and game pads. Not to mention the force feedback possibilities. This chapter will cover the DirectInput API, and how to use it to get the data the user input devices are giving you. The topics being covered include

- Controlling the keyboard
- Controlling the mouse
- Controlling joysticks and game pads
- Force feedback

[\[Team LiB\]](#)

Detecting the Devices You Can Use

Just like last chapter, you will need to ensure that you add references to DirectInput in order to use the code that will be discussed in this chapter. You will need to add a reference to Microsoft.DirectX.DirectInput as well as add a using clause for this namespace. Once this is done for the project(s), you are ready to begin.

Unless your computer is a server system, mounted inside a "rack" (a headless system), chances are your computer has at a minimum two input devices: a keyboard and a mouse.

With the multitude of USB devices on the market today, though, you may very well have quite a few input devices. It's not uncommon for a laptop, for example, to have multiple mice attached. You could theoretically have upward of 100 different user input devices attached to your system at any given time.

With all of these different input devices available to you, what can you do to detect the various devices available on your system? Do you remember way back early in the book when you talked about the Manager class that exists in Direct3D? DirectInput has a similar class that you can use to find out just about anything you will need to about the devices attached (or even not attached) to your system.

The easiest thing you can do is to detect all of the devices available on the system. You can use the devices property on the manager object to do this enumeration. First though, you'll need to create a tree view on our form, and set its dock parameter to fill so that it covers the entire form. You will also need some constants for the "root" key names you will use for the tree view. Add these constants to your application:

```
private const string AllItemsNode = "All Items";
private const string KeyboardsNode = "All Keyboard Items";
private const string MiceNode = "All Mice Items";
private const string GamePadNode = "All Joysticks and Gamepad Items";
private const string FeedbackNode = "All ForceFeedback Items";
```

Based on these constants, you can see that you will need to create five different nodes in the tree: one for all the items on your system, and then one section each for mice, keyboards, joysticks, and force feedback items. Everything in the "All Items" node should be duplicated in other nodes as well.

You should fill this node first. You will create a single function to do them all called "LoadDevices". The only object of this function will be to fill the tree view with the items currently on the system. You should add the method found in [Listing 15.1](#) now.

Listing 15.1 Adding Your Devices to the Tree View

```
public void LoadDevices()
{
    TreeNode allNodes = new TreeNode(AllItemsNode);
    // First get all devices
    foreach(DeviceInstance di in Manager.Devices)
    {
        TreeNode newNode = new TreeNode(string.Format("{0} - {1} ({2})",
            di.InstanceName, Manager.GetDeviceAttached(di.InstanceGuid)
            ? "Attached" : "Detached", di.InstanceGuid));

        allNodes.Nodes.Add(newNode);
    }
    treeView1.Nodes.Add(allNodes);
}
```

As you can see, the device enumerator class (DeviceList) that is returned from the Devices property returns a list of DeviceInstance structures. This structure contains all of the useful information on the devices, including the instance GUID (to use when creating the device), the product name, and the type of device.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using the Keyboard Device

Each of the devices you can use in DirectInput have their own Device object, much like each graphics card and sound card have their own device objects. There are normally more DirectInput devices than the others though.

Creating a device requires a GUID to be passed in that is the same as the instance GUID you receive when enumerating your devices. It can also be a member of the SystemGuid object if you only want to create the default keyboard or mouse device. The devices that are created are generic enough to retrieve data from any device support by DirectInput.

Our first look at the device will start with the most common device, the keyboard. You will obviously need a device variable to use, so go ahead and declare one now:

```
private Device device = null;
```

Now, there are two ways you can use DirectInput. You can use a loop (the normal "game loop" mechanism) and get the current state of the device each frame. You can also set DirectInput up to notify you when the device state has changed so that you can update the game's state. This code will investigate the former mechanism first. Add the following initialization function:

```
private bool running = true;
public void InitializeInput()
{
    // Create our keyboard device
    device = new Device(SystemGuid.Keyboard);
    device.SetCooperativeLevel(this, CooperativeLevelFlags.Background |
        CooperativeLevelFlags.NonExclusive);
    device.Acquire();

    while(running)
    {
        UpdateInputState();
        Application.DoEvents();
    }
}
```

As you can see here, this is the entire "input" loop. You first create the device, using the standard keyboard GUID. If you remember from the DirectSound section, you must set the cooperative level of the devices, since they are shared throughout the system. The various flags that can be used (and combined) for the cooperative levels can be seen in [Table 15.1](#).

Table 15.1. DirectInput Cooperative Levels

FLAG	DESCRIPTION
Background	The device can be used in the background, and can be acquired at any time, even if the associated window isn't the active window.
Foreground	The device can only be used if the window is in the foreground. When this window loses focus, any acquired device will be unacquired.
Exclusive	The device requires exclusive access. While acquired exclusively, no other application can acquire the device exclusively. Nonexclusive acquires are still allowed. For

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using the Mouse Device

All of the DirectInput devices use this same device class, so the differences between using the mouse and keyboard are quite simple. You will need to update the device creation method in `InitializeInput()` to use a mouse GUID rather than the keyboard:

```
device = new Device(System.Guid.Mouse);
```

ACQUIRING THE MOUSE EXCLUSIVELY

Acquiring the mouse in exclusive mode on certain operating systems will hide the mouse cursor completely. While this behavior is sometimes desired, in many instances it is not. For example, if you are writing a menu system for your game that is in full screen mode, the user will expect to be able to use that menu via the mouse. Many times you will need to provide your own cursors in this case.

In order to get the application working, the only other thing you will really need to change is the piece of code where you actually get the data, since the keyboard data doesn't make much sense for the mouse. Update the `UpdateInputState` method as follows:

```
private void UpdateInputState()
{
    // Check the mouse state
    MouseState state = device.CurrentMouseState;
    string mouseState = "Using CurrentMouseState: \r\n";

    // Current location of the mouse
    mouseState += string.Format("{0}x{1}x{2}\r\n", state.X,
        state.Y, state.Z);

    // Buttons
    byte[] buttons = state.GetMouseButtons();
    for(int i = 0; i < buttons.Length; i++)
        mouseState += string.Format("Button {0} {1}\r\n",
            i, buttons[i] != 0 ? "Pressed" : "Not Pressed");

    textBox1.Text = mouseState;
}
```

Here, instead of using a "helper" function to get the current data, you just take a complete snapshot of the mouse data currently. You then output the current X, Y, and Z of the mouse (the Z axis normally refers to the mouse wheel), as well as any of the buttons.

Running the application now, you'll notice that the majority of the time, the "position" of the mouse is reported as 0x0x0, even while you move it around. It's obvious you are really moving the mouse, so why isn't the UI being updated? In actuality, it is; however, the mouse data is being reported in relative units, as opposed to absolute values. That means the values reported for the different axes of the mouse are relative to the last frame. If you want to have the mouse data reported in absolute values, you can update your device's properties as follows (do this immediately after device creation):

```
device.Properties.AxisModeAbsolute = true;
```

While the application is running, it's quite obvious to see it being updated as the mouse moves around; however, the data returned for the axes initially is some crazy number. It is up to the application to keep track of this data to ensure that the mouse moves correctly in the application. The majority of applications will use the relative axis mode that is the default

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Game Pads and Joysticks for User Input

While the basic idea of the joystick device is quite similar to the mouse and keyboard (since they do share the same device class), there are a few other things that need to happen for these, since they are much less standardized. You can pretty much guarantee that there will be two buttons on a mouse, and the keyboard will have at least 36 keys. Joysticks and game pads, on the other hand, come in all shapes and sizes. You may have one with two axes and two buttons, and then another with 3 axes, 10 buttons, two throttles, a pedal, and who knows what else.

There's also no default `System.Guid` member for creating a joystick. You will need to enumerate the joysticks first. Update the device creation function using [Listing 15.4](#).

Listing 15.4 Initializing DirectInput for Joysticks

```
public bool InitializeInput()
{
    // Create our joystick device
    foreach(DeviceInstance di in Manager.GetDevices(DeviceClass.GameControl,
        EnumDevicesFlags.AttachedOnly))
    {
        // Pick the first attached joystick we see
        device = new Device(di.InstanceGuid);
        break;
    }
    if (device == null) // We couldn't find a joystick
        return false;

    device.SetDataFormat(DeviceDataFormat.Joystick);
    device.SetCooperativeLevel(this, CooperativeLevelFlags.Background |
        CooperativeLevelFlags.NonExclusive);
    device.Properties.AxisModeAbsolute = true;
    device.Acquire();

    while(running)
    {
        UpdateInputState();
        Application.DoEvents();
    }

    return true;
}
```

Here you do the enumeration discussed at the beginning of this chapter to try to find an attached joystick or game pad. Since you didn't create the device with a system GUID, you will need to inform `DirectInput` about the type of device this is so that it knows how to interpret the data. You do this with the `SetDataFormat` method. A return value for the method has also been added so that the caller can know if the device creation has failed (because there is no device to create). With this, you will need to update the main function to handle this case:

```
static void Main()
{
    using (Form1 frm = new Form1())
    {
        frm.Show();
        if (!frm.InitializeInput())
            MessageBox.Show("Couldn't find a joystick.");
    }
}
```

Now, joysticks normally have a few axes that can be manipulated, and the range of these axes is unknown normally. You should treat all of the axes the same, with a "resolution" of 10,000 units. It's entirely possible that the device doesn't support this resolution, but `DirectInput` will fake it for you. You need to find the axes on the joystick and

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using Force Feedback

Using force feedback in your application adds to the immersion a player experiences while using your application. Driving a car around at over 100 miles per hour, suddenly ramming into a tree, and having the joystick you're using jerk out of your hand from the force is an experience you just can't match. So, how can you add this to your applications?

One of the first things that needs mentioning is that force feedback effects will require you to have exclusive access to the device. Luckily, these joysticks can be acquired with exclusive mode and in the background. You will use a similar method to create the device for this example as you did for the joystick. Replace the existing `InitializeInput` method with the one found in [Listing 15.5](#).

Listing 15.5 Initializing a Device for Force Feedback

```
private ArrayList effectList = new ArrayList();
public bool InitializeInput()
{
    // Create our joystick device
    foreach(DeviceInstance di in Manager.GetDevices(DeviceClass.GameControl,
        EnumDevicesFlags.AttachedOnly | EnumDevicesFlags.ForceFeedback))
    {
        // Pick the first attached joystick we see
        device = new Device(di.InstanceGuid);
        break;
    }
    if (device == null) // We couldn't find a joystick
        return false;

    device.SetDataFormat(DeviceDataFormat.Joystick);
    device.SetCooperativeLevel(this, CooperativeLevelFlags.Background |
        CooperativeLevelFlags.Exclusive);
    device.Properties.AxisModeAbsolute = true;
    device.Acquire();

    // Enumerate any axes
    foreach(DeviceObjectInstance doi in device.Objects)
    {
        if ((doi.ObjectId & (int)DeviceObjectTypeFlags.Axis) != 0)
        {
            // We found an axis, set the range to a max of 10,000
            device.Properties.SetRange(ParameterHow.ById,
                doi.ObjectId, new InputRange(-5000, 5000));
        }
    }

    // Load our feedback file
    EffectList effects = null;
    effects = device.GetEffects(@"..\..\idling.ffe",
        FileEffectsFlags.ModifyIfNeeded);
    foreach(FileEffect fe in effects)
    {
        EffectObject myEffect = new EffectObject(fe.EffectGuid, fe.EffectStruct,
            device);
        myEffect.Download();
        effectList.Add(myEffect);
    }

    while(running)
    {
        UpdateInputState();
        Application.DoEvents();
    }

    return true;
}
```


[\[Team LiB\]](#)

In Brief

- - Controlling the keyboard
- - Controlling the mouse
- - Controlling joysticks and game pads
- - Force feedback

In our next chapter, we will look at multiplayer opportunities by introducing peer-to-peer networking.

[\[Team LiB\]](#)

◀ PREVIOUS

NEXT ▶

Part V: 2D Graphics

16 [Using Direct3D for 2D Graphics](#)

17 [Using DirectDraw for 2D Rendering](#)

[\[Team LiB\]](#)

◀ PREVIOUS

NEXT ▶

Chapter 16. Using Direct3D for 2D Graphics

IN THIS CHAPTER

- [Creating a Full Screen Rendering Device](#)
- [Rendering Sprites](#)
- [Animating Your Sprites](#)
- [In Brief](#)

This chapter will cover rendering 2D graphics in Direct3D, including

- Creating a full screen device.
- Using the Sprite class.
- Rendering sprites.

[\[Team LiB\]](#)

Creating a Full Screen Rendering Device

Rendering complex 3D scenes is by definition "complex." While it is true that the vast majority of games being released currently use 3D graphics, there are still plenty of game types out there where the "fancy" graphics aren't necessary. It seems that more often than not these days, games that should be written in 2D are still written in 3D, just because you "have to."

Given that Direct3D can render complex 3D scenes, it should be no surprise that it is also quite capable of rendering a fully two-dimensional scene.

Creating a full screen device is not that much different from creating a windowed device. The same basic principles apply; there are just a few extra properties that will need to be filled out. Create a windows forms application that can be used to render the 2D graphics. You will need to add the using clauses, declare the device variable, and set the windows style before the application, like has been done in previous chapters.

Once the project has been created, you will need to add two constants for the full screen size:

```
public const int ScreenWidth = 800;
public const int ScreenHeight = 600;
```

These constants will represent the width and height of the full screen window that will be used for rendering. With the constants declared, you can now add the updated initialization method found in [Listing 16.1](#).

Listing 16.1 Initializing Your Full Screen Device

```
public void InitializeGraphics()
{
    // Set our presentation parameters
    PresentParameters presentParams = new PresentParameters();
    presentParams.SwapEffect = SwapEffect.Discard;

    // Start up full screen
    Format current = Manager.Adapters[0].CurrentDisplayMode.Format;

    if (Manager.CheckDeviceType(0, DeviceType.Hardware, current, current, false))
    {
        // Perfect, this is valid
        presentParams.Windowed = false;
        presentParams.BackBufferFormat = current;
        presentParams.BackBufferCount = 1;
        presentParams.BackBufferWidth = ScreenWidth;
        presentParams.BackBufferHeight = ScreenHeight;
    }
    else
    {
        presentParams.Windowed = true;
    }
    // Create our device
    device = new Device(0, DeviceType.Hardware, this,
        CreateFlags.SoftwareVertexProcessing, presentParams);
}
```

The beginning of this method should appear familiar. You'll notice that there is no depth buffer being declared for this device. Since the only objects being rendered will be 2D, and thus at the same depth, keeping the depth buffer provides no benefit. In actuality it's a performance hit, since the depth check will be performed for each sprite.

It's normally a safe assumption that the current display mode format of the default adapter is a valid format for a full screen device. Before blindly assuming this is the case, you should perform the check done here by calling the `CheckDeviceType` method. If this device type is supported, update the presentation parameters to specify a full

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Rendering Sprites

Direct3D already has an object that can be thought of as a sprite, namely the texture. However, textures must be rendered onto some type of primitives in order to be seen by the user. The Sprite class handles this work for you. Add the following variables for your application:

```
private Texture spriteTexture;
private Sprite sprite;
private Rectangle textureSize;
```

These obviously store the texture that will be used to render the sprite, as well as the sprite itself. You will also need to know the size of the texture in order to render it correctly, so that will be stored as well. At the end of your initialization method, directly after the device has been created, you will need to initialize these variables as follows:

```
// Create our texture
spriteTexture = TextureLoader.FromFile(device, @"..\..\logo.tga");

using (Surface s = spriteTexture.GetSurfaceLevel(0))
{
    SurfaceDescription desc = s.Description;
    textureSize = new Rectangle(0, 0, desc.Width, desc.Height);
}
sprite = new Sprite(device);
```

You will find the logo.tga file located on the enclosed CD. After the texture has been created, we will need to determine its size. We retrieve the underlying surface and use the description of this surface to store our texture's size. Finally, we create an instance of the sprite class.

Each instance of the sprite class can draw zero to many sprites, by using various textures. Rather than duplicate our sprite class and textures, we will create a new class to handle our multiple sprites. Add the class found in [Listing 16.2](#).

Listing 16.2 The Custom Sprite Class

```
public class GraphicsSprite
{
    // static data for our sprites
    private static readonly Vector3 Center = new Vector3(0, 0, 0);
    private static readonly Random rnd = new Random();

    // Instance data for our sprites
    private Vector3 position;
    private float xUpdate = 1.4f;
    private float yUpdate = 1.4f;

    public GraphicsSprite(int posX, int posY)
    {
        position = new Vector3(posx, posY, 1);
        xUpdate += (float)rnd.NextDouble();
        yUpdate += (float)rnd.NextDouble();
    }

    public void Draw(Sprite sprite, Texture t, Rectangle r)
    {
        sprite.Draw(t, r, Center, position, Color.White);
    }
}
```

Sprites can be rotated (even though this code won't perform this operation). When you draw the sprites, one of the parameters that must be passed in is the center of the sprite for rotation calculation. Rather than creating a new vector

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Animating Your Sprites

Most applications that render 2D sprites will not simply render a static image continuously. The majority of times, they will render a series of consecutive images that form what appears to be animation. Since the basic principle of this rendering is the same as the static image written earlier, that code will be used as the starting point. Go ahead and make a copy of that code for this new application.

The major differences between this application and the last will be in the helper class being used to render the sprites. The initialization will be mostly the same, although the filename that the source code included with the CD uses has been changed to sprites.tga to emphasize the fact that there are multiple sprites stored in this one file.

First, since these will be small animated sprites flying around the screen, rather than the large static one, it makes more sense to have the position of these sprites initialized randomly. Update the constructor for the sprite class as follows:

```
public GraphicsSprite()
{
    position = new Vector3(rnd.Next(Form1.ScreenWidth-SpriteSize),
        rnd.Next(Form1.ScreenHeight-SpriteSize), 1);

    xUpdate += (float)rnd.NextDouble();
    yUpdate += (float)rnd.NextDouble();

    column = rnd.Next(NumberSpritesCol);
    row = rnd.Next(NumberSpritesRow);

    if ((column % 3) == 0)
        xUpdate *= -1;

    if ((row % 2) == 0)
        yUpdate *= -1;
}
```

The position of the sprite onscreen is determined by taking the width and height of the screen, subtracting the size of the sprite, and getting a random integer between zero and that value. However, the sprite size hasn't been defined yet. For the sprite that is stored on the included CD, the size of each frame of animation is 50x45. Define these constants now:

```
private const int SpriteSizeWidth = 50;
private const int SpriteSizeHeight = 45;
```

Once the position has been determined, the velocity is set to a random value between zero and one. You could multiply this value by two to simulate "quicker" sprites if need be. The sprites in this example are formed by a series of 50x45 images, each located in the file sequentially. The file is composed of five of these sprites per column, and a total of six rows. Store this information using constants:

```
private const int NumberSpritesRow = 6;
private const int NumberSpritesCol = 5;
```

You will also need to declare the variables that will be used to store the current animation frame:

```
private int column = 0;
private int row = 0;
```

Since the code is randomly picking the position, it may as well randomly pick a starting point for the animation as well. Once that is done, you don't want every sprite to be moving in the same direction (at random speeds), so some of the sprites' initial velocity will be reversed

[\[Team LiB\]](#)

In Brief

- - Switching to full-screen mode.
- - Rendering sprites.
- - Rendering animated sprites.

In the next chapter we will look at each of the items covered in this chapter, only DirectDraw will be used instead.

Chapter 17. Using DirectDraw for 2D Rendering

IN THIS CHAPTER

- [Creating a Full Screen DirectDraw Device](#)
- [Animating Your Sprites](#)
- [In Brief](#)

[In Brief](#)

One of the downsides of using Direct3D to render simple 2D applications is the hardware requirements. Direct3D is by its nature more complex than DirectDraw, and the minimum hardware requirements just for creating a Direct3D device can be too high for some systems. For example, laptops that are only a little over a year old at times don't have the capabilities of creating a Direct3D device, yet they can easily create a DirectDraw Device.

Even with the pervasive support Direct3D has in the graphics cards today, DirectDraw has even more. If you need to render 2D graphics with the minimum hardware requirements, DirectDraw will meet your needs quite nicely. In this chapter, we will cover the same types of rendering we did in the previous chapter, just using DirectDraw instead. These actions will include

- Creating a full screen device.
- Rendering sprites.
- Animating sprites.

[\[Team LiB\]](#)

Creating a Full Screen DirectDraw Device

Before you can create your DirectDraw device, a new project will need to be created. Since this will still be an application where items will need to be rendered, windows forms should still be used. Use these steps to get started:

1.
Create a new Windows Forms project named whatever you like.
2.
Add a reference to Microsoft.DirectX.dll and Microsoft.DirectDraw.dll.
3.
Include a new using clause for Microsoft.DirectX.DirectDraw into your main code file.
4.
Set the window style to Opaque and AllPaintingInWmPaint, just like you did for 3D applications.
5.
Add a private variable for your DirectDraw device.

You'll notice that most of these steps are similar to what was done for the 3D applications that were built in earlier chapters, with the major difference being the references that were added.

In DirectDraw you don't have textures that are used to represent your 2D images; instead you have "surfaces." A surface is essentially a data store for a rectangular image. When rendering a single sprite with a full-screen device, you will need a total of three surfaces. Add these to your application now:

```
private Surface primary = null;
private Surface backBuffer = null;
private Surface sprite = null;
```

Obviously the sprite surface will be where the actual image being rendered is stored. The first two parameters are something new, though. Since the screen itself is essentially a rectangular surface that you will be rendering to, you will use a surface to access this as well. Rendering directly to the video screen is never a good idea, because then you can have problems with what is called "tearing," which is when part of two (or more) different rendered scenes end up on the screen at the same time. To ensure this behavior doesn't happen, all of the rendering should happen in a stored back buffer, and then be copied to the primary buffer in one chunk. This is handled for you automatically in Direct3D (see [Chapter 1](#), "Introducing Direct3D"), but you must handle it yourself in DirectDraw.

Before you go on, you should once again ensure that you handle the Escape key to quit. Since the application will be running in full screen mode, you want an easy way to quit. Add the following code to ensure your application exits on escape:

```
protected override void OnKeyUp(KeyEventArgs e)
{
    if (e.KeyCode == Keys.Escape) // Quit
        this.Close();

    base.OnKeyUp (e);
}
```

Now that an easy method to quit has been established, you can create your initialization method. Add the method found in [Listing 17.1](#) to your application.

Listing 17.1 Initialize a DirectDraw Device

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Animating Your Sprites

To follow the same pattern that was used in the previous chapter, you should now make a copy of this code so that it can be updated to show some animated sprites.

USING NON-SQUARE SURFACES

In Direct3D textures are normally required not only to be square, but to also have each side have a length that is a power of two. Most of the modern cards out today have support for textures that don't fit these parameters, but even those that do can suffer a performance loss for using "irregularly" shaped textures.

In the previous chapter, the animated sprites file was "enlarged" to be a square with a power-of-two length, with the remaining area being filled with "unused" data just to ensure that the texture could be loaded on most cards. DirectDraw does not enforce these rules on the surfaces it creates, thus the sprite in this example is of the exact size needed to hold the data.

The major changes will once again happen in the sprite class. The same constants that were used before can be used now:

```
private const int NumberSpritesRow = 6;
private const int NumberSpritesCol = 5;
private const int SpriteSizeWidth = 50;
private const int SpriteSizeHeight = 45;
```

The source code included on the CD uses the file sprites.bmp for the sprites to be rendered, which is the same file used in the previous chapter, just with different dimensions and no alpha channel.

You will need to store the animation frame as well, which is a combination of a row and column. Add those variables to your sprite class:

```
private int column = 0;
private int row = 0;
```

Now, you will need to update the constructor, to randomly place the sprite somewhere on the screen, with a random animation frame:

```
public GraphicsSprite()
{
    xPosition = rnd.Next(Form1.ScreenWidth-SpriteSizeWidth);
    yPosition = rnd.Next(Form1.ScreenHeight-SpriteSizeHeight);

    xUpdate += (float)rnd.NextDouble();
    yUpdate += (float)rnd.NextDouble();

    column = rnd.Next(NumberSpritesCol);
    row = rnd.Next(NumberSpritesRow);

    if ((column % 3) == 0)
        xUpdate *= -1;

    if ((row % 2) == 0)
        yUpdate *= -1;
}
```

[\[Team LiB\]](#)

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

In Brief

- - Switching to full screen exclusive mode.
- - Rendering a single image as a sprite onscreen.
- - Rendering animated sprites.

In the next chapter we will look at peer-to-peer networking.

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Part VI: Adding Networking

18 [Implementing Peer-to-Peer Networking Using DirectPlay](#)

19 [Creating a Client/Server Session](#)

20 [Understanding Advanced Networking Features](#)

21 [Achieving Maximum Performance](#)

Chapter 18. Implementing Peer-to-Peer Networking Using DirectPlay

IN THIS CHAPTER

- [Understanding DirectPlay Addresses](#)
- [Creating a Peer Connection](#)
- [Getting into a Session](#)
- [Using the Event Model](#)
- [Performing Actions in Our Session](#)
- [Handling Lost Sessions](#)
- [In Brief](#)

In this chapter, we will look at the peer-to-peer networking aspects of DirectPlay, including the following:

- DirectPlay addresses
- The peer object
- The event model of DirectPlay
- Sending data to other members of your session

[\[Team LiB\]](#)

Understanding DirectPlay Addresses

Playing single-player games can be fun and exciting, but it's nothing compared to playing against real live people. Even game consoles that were notoriously single player have begun offering online play and matchmaking services. It's rare to see top-of-the-line games coming out nowadays without some form of multiplayer features. There is also an entire class of quite popular games that have no single-player options at all; they are only multiplayer.

Managed DirectX includes the DirectPlay API that can be used for implementing a networking layer. Before we begin the code for this chapter, we will once again need to ensure that we have the right references for DirectPlay. Ensure that you have the Microsoft.DirectX.DirectPlay reference included in your projects, as well as the using clause. All of the code in this chapter assumes this has already been done.

If you have a computer that can connect to the Internet, your computer already has a TCP/IP address. Each computer needs to have a unique "address" that distinguishes it from other computers on a network. While it's true that TCP/IP addresses are not unique, they are unique within a single network. TCP/IP addresses are the standard address of the Internet.

DirectPlay also needs a way to distinguish one computer from another, and each networked computer must have a unique address. The act of creating a unique DirectPlay address isn't all that difficult. Let's look at the code:

```
Address address = new Address();
address.ServiceProvider = Address.ServiceProviderTcpIp;
```

As the preceding code implies, there is more than one service provider that DirectPlay can use. We've chosen in this instance to use the TCP/IP provider, which is by far the most common service provider available in computers today. The service providers available to DirectPlay by default are

- TCP/IP
- IPX
- BlueTooth
- Serial connections
- Direct modem-modem connections

USING URLS AS ADDRESSES

Addresses can also be specified in URL (Uniform Resource Locator) form, much like Web addresses. If you've seen an address like <http://www.mycompany.com>, you've seen an URL. The first "section" of the URL is the protocol type, in this case http (or Hypertext Transfer Protocol). The Web pages you see are normally HTML (Hypertext Markup Language), thus the name of the protocol. In DirectPlay, these URLs take on a construct; for example, the address we just created could also be specified as

[\[View full width\]](#)

```
x-directplay:/provider=%7BEBFE7BA0-628D-11D2-AE0F-006097B01411
%7D;hostname=www.mygameserver.com;port=9798
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Creating a Peer Connection

With the basics of addressing out of the way, we can move on to actually getting a connection made, and really start to network. The types of networking that DirectPlay supports would be peer-to-peer and client/server. In this chapter, we will discuss the peer-to-peer connections.

So what is a peer-to-peer connection? As the name implies, each member (or peer) of a session is connected to every other member (or peer) in the session. For small sessions, this method works quite well. There are many examples of peer-to-peer networks that are quite popular today. Many of the file-sharing programs out there today (Kazaa, Napster, and so on) are peer-to-peer networks.

Before you can begin a peer-to-peer session in DirectPlay, you must first decide a few things. What service provider do you want the connection to use? Will you be "hosting" this connection, or will you connect to an existing session? You can think of the first peer in a session as the "master" peer, or host of the session.

The main class that we will use for our session is (I'm sure you've guessed it) the Peer object. It's always easier to see things visually while looking at code, so let's start writing some now. Declare a variable for our peer object, and our addresses:

```
private Peer connection = null;
private Address deviceAddress = null;
```

You'll notice that we have our address declared. We will maintain this address to hold the connection information for our local machine. We also maintain a peer object for our connection.

In the design view for your form, create two buttons in the upper right corner. The first will be to "Host" a session, while the second will be to "Connect" to an existing session. Add a third button to the upper right corner of your form; this will be to send any data throughout the session. This third button should be disabled at first (we don't want to send any data before we are in a session). Finally, create a large label to fill up the bottom portion of the UI. This will be our status screen. With the plumbing work out of the way, we're ready to start networking.

We will create one initialization function to set up our peer object and any default state. Let's examine the function in [Listing 18.1](#) now.

Listing 18.1 Initializing a Peer Object

```
private void InitializeDirectPlay(bool host)
{
    // Create our peer object
    connection = new Peer();

    // Check to see if we can create a TCP/IP connection
    if (!IsServiceProviderValid(Address.ServiceProviderTcpIp))
    {
        // Nope, can't, quit this application
        MessageBox.Show("Could not create a TCP/IP service provider.", "Exiting",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        this.Close();
    }
    // Create a new address for our local machine
    deviceAddress = new Address();
    deviceAddress.ServiceProvider = Address.ServiceProviderTcpIp;
}
```

This method will get much larger before we're done. For now though, what is it doing? Well, first we create our peer object using the default constructor. There is one other constructor available that takes in initialization flags that we didn't use. These flags can be one or more of the values found in [Table 18.1](#).

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Getting into a Session

We have two buttons on our form for either hosting or connecting to an existing session. Once we try to do either of these actions, we will need to disable these buttons so that we can't try to do more than one action at a time. The event handlers for these two buttons would be

```
private void button1_Click(object sender, System.EventArgs e)
{
    button1.Enabled = false;
    button2.Enabled = false;
    InitializeDirectPlay(true);
}

private void button2_Click(object sender, System.EventArgs e)
{
    button1.Enabled = false;
    button2.Enabled = false;
    InitializeDirectPlay(false);
}
```

As you see, they are both remarkably similar. The only difference between the two is the parameter they pass in to the `InitializeDirectPlay` method; one uses `true` to host, the other uses `false` to connect. We could simplify this by having each button use the same event handler:

```
private void button_Click(object sender, System.EventArgs e)
{
    button1.Enabled = false;
    button2.Enabled = false;
    InitializeDirectPlay((sender == button1));
}
```

Much nicer; this eliminates the duplicated code and achieves the exact same behavior. If the sender of the event happens to be the first button (our hosting button), it will call `InitializeDirectPlay` with `true`; otherwise, it will call it with `false`. Hook each button's event in this way. We never actually did anything with the parameter for this method, though. We should look at that now.

Before we can either host a session or connect to an existing one, we will need a way to "recognize" a session. `DirectPlay` includes an application description structure that will contain all of the information we need to describe an application. Let's look at the code we'll use for our application description; add this to the end of the `InitializeDirectPlay` method:

```
// Set up an application description
ApplicationDescription desc = new ApplicationDescription();
desc.SessionName = "MDXBookPeerSession";
desc.GuidApplication = new Guid(41039, 1702, 1503, 178, 101,
    32, 13, 121, 230, 109, 59);
```

The most important member of this structure is the `GuidApplication` member. This GUID uniquely identifies your application, and all instances of your application must use the same GUID for this member. The other member we set is the session name. This is useful for when there are many sessions available to help identify yours. You will find all of the members of the `ApplicationDescription` structure in [Table 18.2](#).

Table 18.2. ApplicationDescription Members

MEMBER	DESCRIPTION
<code>GuidApplication</code>	Uniquely identify your application.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Using the Event Model

The majority of actions you perform in a DirectPlay session happen asynchronously. The method will return almost instantly, and then DirectPlay will go to work doing the action you requested. After this action has been completed, or when there is something DirectPlay needs to inform the application about, an event will be fired on a separate thread. These events are how we find out everything we need to know.

In our preceding call to, the method will return right away, but behind the scenes, DirectPlay is busy looking for any available sessions. Whenever one is found the event is fired. Since we want to know about the hosts we find, we should hook this event. Add the following line directly after you've created your new peer object in the InitializeDirectPlay method:

```
connection.FindHostResponse += new
    FindHostResponseEventHandler(OnFindHost);
```

You'll also need to add the actual method that will be used to handle this event. You will find this in [Listing 18.2](#).

Listing 18.2 Event Handler for a Found Host

```
private void OnFindHost(object sender, FindHostResponseEventArgs e)
{
    lock(this)
    {
        // Do nothing if we're connected already
        if (connected)
            return;

        connected = true;
        string foundSession = string.Format
            ("Found session ({0}), trying to connect.",
            e.Message.ApplicationDescription.SessionName);

        this.BeginInvoke(new AddTextCallback(AddText),
            new object[] { foundSession });

        // Connect to the first one
        ((Peer)sender).Connect(e.Message.ApplicationDescription,
            e.Message.AddressSender,
            e.Message.AddressDevice, null, ConnectFlags.OkToQueryForAddressing);
    }
}
```

As you can see, our event handler includes a FindHostsResponseEventArgs object that will include the data about the host that was found. Examining the rest of this method tells us a lot about how multi-threaded programming should happen.

First, it's entirely possible for two different hosts to be found by DirectPlay. It's also possible that your application could be notified about each on two separate threads at the exact same time. Having two separate threads manipulating the same data at the same time is a big no-no, so we want to avoid that. We do this by using the lock keyword from C# (which is really just Monitor.Enter and Monitor.Exit calls wrapped in a try/finally block). Once the lock block has been entered by one thread, no other thread can enter that block until the first one has left the block.

We can only connect to one host at a time, so we will use a Boolean variable to check if we are already connected (you will need to declare this variable in your class; feel free to declare it publicly). If we are, there's nothing more to do in this method; we can just return. If we aren't, however, the first thing we will do is update this variable so that new enumerations won't interrupt us while we really do try to connect.

Now we want to update our UI to let the user know that we have found a session, and we are going to try to

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Performing Actions in Our Session

We've made our connect call, but much like after our find hosts call, we don't really know what the outcome of that call was. Naturally, there is an event to handle this as well. Add the following handler to our InitializeDirectPlay method:

```
connection.ConnectComplete += new
    ConnectCompleteEventHandler(OnConnectComplete);
```

We also need the event handler method implementation, so add this as well:

```
private void OnConnectComplete(object sender, ConnectCompleteEventArgs e)
{
    // Check to see if we connected properly
    if (e.Message.ResultCode == ResultCode.Success)
    {
        this.BeginInvoke(new AddTextCallback(AddText),
            new object[] { "Connect Success." });

        connected = true;

        this.BeginInvoke(new EnableCallback(EnableSendDataButton),
            new object[] { true });
    }
    else
    {
        this.BeginInvoke(new AddTextCallback(AddText), new object[] {
            string.Format("Connect Failure: {0}", e.Message.ResultCode) });

        connected = false;

        this.BeginInvoke(new EnableCallback(EnableSendDataButton),
            new object[] { false });
    }
}
```

So this is where we can check to see if our connection call actually succeeded. The result code will tell us the result of the operation, and if that was a success, we've been connected to the session. In this case, we want to update our UI, and enable the send data button. If the connect call failed, we still update our UI; however, instead we disable the send data button, and set our connected variable to false. Setting this to false allows the FindHosts handler to try to connect to other sessions that it finds. You'll also notice that we don't have the delegate to enable or disable our send data button declared. You can declare it as follows:

```
private delegate void EnableCallback(bool enable);
```

You can run two instances of the application now and get them to connect to one another and be joined in the same session. On the first instance, click the Host button, and it should inform you that you are currently hosting the session. You can click the Connect button on the second instance, and after the dialog asking you about the remote host, you should see that it found your session and connected to it.

What we need now is a way to actually send some type of data to the session. We already have our send data button that currently doesn't do anything, so let's add this event handler for it:

```
private void button3_Click(object sender, System.EventArgs e)
{
    NetworkPacket packet = new NetworkPacket();

    packet.Write(byte.MaxValue);
    connection.SendTo((int)PlayerID.AllPlayers, packet,
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Handling Lost Sessions

When the host of the session quit, the connection was then dropped. There was no longer any host, so the connection couldn't be maintained any longer. DirectPlay sent a message to all connected clients informing them of this; however, we never included a handler for this event. Let's do this now:

```
connection.SessionTerminated += new
    SessionTerminatedEventHandler(OnSessionTerminate);
```

The session-terminated event can be fired when the host has quit (like we did previously) or for just about any other reason why you can no longer be in the session. For example, if you unplugged your network cable, shortly later you would receive this event because no one in the session could be contacted. We should add the code for this handler now:

```
private void OnSessionTerminate(object sender, SessionTerminatedEventArgs e)
{
    // The session was terminated, close our peer object, re-enable our buttons
    this.BeginInvoke(new DisconnectCallback(OnDisconnect), null);
}
```

Hmmm, we've got a new delegate we're calling, plus an entirely new method. Why exactly did we do it this way? First, we'll declare the delegate and the method used:

```
private delegate void DisconnectCallback();
private void OnDisconnect()
{
    // Re-enable the UI
    button1.Enabled = true;
    button2.Enabled = true;
    button3.Enabled = false;
    // Notify the user
    AddText("The host has quit, you have been disconnected.");

    // Dispose of our connection, and set it to null
    connection.Dispose();
    connection = null;
}
```

Now you can see the reason for the new delegate and method. We need to do quite a bit here. First, we switch the button states back to the defaults so that we can be allowed to host a new session or join a different session. We then update the UI to inform the user, and finally, since this connection is no longer valid, we dispose it and set it to null.

SHOP TALK: HANDLING HOST MIGRATION

I know what you must be thinking. Just because the first person in the session quit, why does the entire session need to be disbanded? That seems pretty stupid. Luckily, this isn't always the case. DirectPlay has a feature called "Host Migration" that will automatically pick a new host out of the remaining peers if the original host happens to quit. In the design view of our form, add a check box between the Connect and Send Data buttons. The text for this button should be "Migrate the host".

Now we'll need to update a few different places in our application to enable host migration. First, we'll need to see whether our box has been checked before we host, since the peer that calls Host must include the host migration flag if the option is going to be used. Add the following line before the call to

[\[Team LiB\]](#)

In Brief

- - DirectPlay addresses
- - The peer object
- - The event model of DirectPlay
- - Sending data to other members of your session
- - Host migration

In our next chapter, we will look at multiplayer opportunities using the client/server topology and start sending much more interesting data.

Chapter 19. Creating a Client/Server Session

IN THIS CHAPTER

- [Creating a Dedicated Server Session](#)
- [Connecting a Client](#)
- [Detecting Users Joining and Leaving Your Session](#)
- [Sending Data Packets](#)
- [Making the Client React](#)
- [Handling the Server Lost Event](#)
- [In Brief](#)

Peer-to-peer sessions can work quite well when you're dealing with a small subset of users. However, when you're looking to maintain a network session with hundreds or even thousands of simultaneous players, you really need a dedicated server. In a peer-to-peer session, all data sent from one peer to the other peers can be duplicated for every peer in the session. For small groups, this isn't prohibitive, but even a small chunk of data can be quite large when it's sent 10,000 times.

DirectPlay includes two other networking classes much like the peer object: the client and the server objects. You can think of each of these as a subset of the functionality that exists in the peer object, considering that the peer object is essentially both a client and a server. In this chapter you will create a client/server application that delves a little deeper into the workings of DirectPlay, including

- Creating dedicated servers running on particular ports
- Connecting to these servers using the client interface
- Tracking player joining and disconnecting
- Sending game-specific data

[\[Team LiB\]](#)

Creating a Dedicated Server Session

Obviously, you will need to create a server before the client. Without the server, what exactly could the client do? In practice, you may find working on both the client and server portions of the code in tandem beneficial. That model will be followed here, but before you get to that point, you have to meet a "bare minimum" standard of working. You can define this as being able to host a dedicated session and having a client connect. So now you can get started with your server.

The majority of dedicated server applications pick an Internet port that they (and only they) run. For example, Web servers almost always listen on port 80 for any Web requests. FTP servers use port 21. Only one application may listen to any given port at one time, so it's common for the application to pick a unique port to run with. The first thousand ports are considered "reserved," but you may feel free to use any available port to listen on.

For this dedicated server application, you should pick a port you feel comfortable using. The code in the text will use port 9798 to listen on. There won't be any options for the server; it will just run and update its status via the UI. The only action the server will be able to perform other than what you program into it will be shutting down.

Naturally then, you'll want a windows form for the UI status update. Go ahead and create a windows form application, and make sure your references and using clauses have been defined for DirectPlay. You will want a form with a fixed dialog border style and a label. The label can be set to a dock mode of fill so that it covers the entire form.

You will first need a declaration for the server object (which, as mentioned earlier, is similar to the peer object):

```
private Server connection = null;
```

You will need to add the initialization function. Since it's not possible for a server object to connect to any other session, there is no need to have a Boolean parameter for this method. Add the method found in [Listing 19.1](#) to your application.

Listing 19.1 Initializing a Server

```
public void InitializeServer()
{
    // Create our server object
    connection = new Server();

    // Check to see if we can create a TCP/IP connection
    if (!IsServiceProviderValid(Address.ServiceProviderTcpIp))
    {
        // Nope, can't, quit this application
        MessageBox.Show("Could not create a TCP/IP service provider.", "Exiting",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        this.Close();
    }
    // Create a new address for our local machine
    Address deviceAddress = new Address();
    deviceAddress.ServiceProvider = Address.ServiceProviderTcpIp;
    deviceAddress.AddComponent(Address.KeyPort, SharedCode.DataPort);

    // Set up an application description
    ApplicationDescription desc = new ApplicationDescription();
    desc.SessionName = "MDX Book Server Session";
    desc.GuidApplication = SharedCode.ApplicationGuid;
    desc.Flags = SessionFlags.ClientServer | SessionFlags.NoDpnServer;

    try
    {
        // Host a new session
        connection.Host(desc, deviceAddress);
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Connecting a Client

With the bare minimum for the server session created, you can now do the small amount of work to get a client connected to this server. While it is technically possible to add the code for the client into the same executable as the server, in order to have a dedicated server, you will want the code separated. In order to do this, you will create an entirely new second windows forms application.

The client UI will be similar to the servers. You should have one large label covering the majority of the screen. The difference is that you should leave some room for buttons along the top of the form, since you will use these buttons to have the client perform particular actions. You also will want the clients to be able to have a name for this session, and you will want this name to be changeable. For now, just create the label covering most of the form, as well as a textbox above this label where you will be able to modify the client's name.

You will also need to ensure that you have access to the shared data between the server and the client, so you will need to add a link to the shared code file you created for the server. When adding the existing item, be sure to click the small arrow on the open button, and select the Link File option if the code file is in a different folder. Otherwise, simply selecting the Open option will create a copy of the file in your local folder, and it will no longer be shared between the two applications.

With that minor plumbing out of the way, you can start writing the client code. You will first need to create your local variables:

```
private Client connection = null;
private bool connected = false;
```

Much like the peer-to-peer session, you will need a variable to track whether or not you've been connected to a session. You also need to maintain the connection object as well. Once again, you will need to have an initialization method implemented, only this time it will only try to find existing hosts. Add the method from [Listing 19.2](#) to your client application.

Listing 19.2 Initializing a Client Connection

```
public void InitializeClient()
{
    // Create our client object
    connection = new Client();
    // Hook the events we want to listen for
    connection.ConnectComplete += new
        ConnectCompleteEventHandler(OnConnectComplete);

    connection.FindHostResponse += new
        FindHostResponseEventHandler(OnFindHost);

    // Check to see if we can create a TCP/IP connection
    if (!IsServiceProviderValid(Address.ServiceProviderTcpIp))
    {
        // Nope, can't, quit this application
        MessageBox.Show("Could not create a TCP/IP service provider.", "Exiting",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        this.Close();
    }
    // Create a new address for our local machine
    Address deviceAddress = new Address();
    deviceAddress.ServiceProvider = Address.ServiceProviderTcpIp;
    // Create a new address for our host machine
    Address hostaddress = new Address();
    hostaddress.ServiceProvider = Address.ServiceProviderTcpIp;
    hostaddress.AddComponent(Address.KeyPort, SharedCode.DataPort);

    // Set our name
```

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Detecting Users Joining and Leaving Your Session

Now that clients are connecting to the server, it would be nice to update the UI on the server side to indicate when a client has connected (and left, for that matter). However, currently no way to do this has been discussed. As luck would have it, there are two events you can hook to give you this information. In your initialization method for the server, you will need to add the following event hooks:

```
connection.PlayerCreated += new
    PlayerCreatedEventHandler(OnPlayerCreated);

connection.PlayerDestroyed += new
    PlayerDestroyedEventHandler(OnPlayerDestroyed);
```

Whenever a player either joins or leaves the session, the appropriate event will be fired. The implementations of the event handlers are pretty similar as well. Add them from [Listing 19.4](#) to your server application.

Listing 19.4 Player Management Event Handlers

```
private void OnPlayerCreated(object sender, PlayerCreatedEventArgs e)
{
    try
    {
        string playerName = ((Server)sender).GetClientInformation
            (e.Message.PlayerID).Name;

        string newtext = string.Format
            ("Accepted new connection from {0}, UserID: 0x{1}",
            playerName, e.Message.PlayerID.ToString("x"));

        this.BeginInvoke(new AddTextCallback(AddText),
            new object[] { newtext });
    }
    catch { /* Ignore this, probably the server */ }
}

private void OnPlayerDestroyed(object sender, PlayerDestroyedEventArgs e)
{
    string newtext = string.Format
        ("DirectPlayer UserID: 0x{0} has left the session.",
        e.Message.PlayerID.ToString("x"));

    this.BeginInvoke(new AddTextCallback(AddText),
        new object[] { newtext });
}
```

The player-destroyed handler is quite simple, as all it does is update the UI to notify you that the user has left the session. The player-created handler has a method that has never actually been discussed before, though. In the client's application, you set the client name with a call to `SetClientInformation`; in the server you retrieve this name with the call to `GetClientInformation`. When a server first starts a session, a default "hidden" player is created. Since there is no client associated with this player, the call to `SetClientInformation` will naturally fail. You can simply ignore that case for this example since it is expected.

SHOP TALK: CONTROLLING WHO JOINS YOUR SESSION

It's entirely possible that there are certain users who you do not want joining your session. However, as

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Sending Data Packets

You've already written some code for sending a simple data packet in the peer-to-peer session, but all you really did was send a single byte of data, and it didn't even really matter what that data was. In a real networking session, the traffic that will be sent between the server and the clients will be much more varied, and intended to perform specific actions.

To model more accurately what a "real" application might do, you should add a few options to the client that facilitate sending various data packets to the server. You can include the following actions for your client:

- Check the number of players currently in the session.
- Wave to other players in the session.
- Send "private" data to the server.
- Run away from the other players in the session.
- Change the client's name.

Obviously, both the client and server applications will need to know about these actions, so you will add the following enumeration to the shared code file that will track the type of data packet you will be sending to the server:

```
public enum NetworkMessages
{
    CheckPlayers,
    Wave,
    SendData,
    RunAway,
    ChangeName,
}
```

You have one value for each of the actions you want the client to perform. You could have just as easily created a list of constants to use, but the grouping of the enumeration made it more attractive for this simple case. Now you will need a way for the client to actually perform these actions. In your client application, create five buttons at the top of your form, one for each of these actions. In order to simplify the sending of the data, you will hook and create a single event handler for all of the buttons. The event handler method can be found in [Listing 19.5](#).

Listing 19.5 Sending Data Based on Button Clicks

```
private void button_Click(object sender, System.EventArgs e)
{
    // We want to send some data to the server
    NetworkPacket packet = new NetworkPacket();

    // Write the correct message depending on the button pressed
    switch(((Button)sender).Name)
    {
        case "button1":
            packet.Write(NetworkMessages.CheckPlayers);
            break;
        case "button2":
            packet.Write(NetworkMessages.Wave);
            break;
```

[\[Team LiB\]](#)

Making the Client React

It's clear now that the client will also need to hook the receive data event since you've updated the server to actually send data as well. Add the same hook you used for the server into your client application. The actual event handler for the client will be similar as well. Add the event handler found in [Listing 19.7](#) to your client application.

Listing 19.7 Handling Received Data on the Client

```
private void OnDataReceive(object sender, ReceiveEventArgs e)
{
    NetworkMessages msg = (NetworkMessages)e.Message.ReceiveData.Read
        (typeof(NetworkMessages));

    string newtext = string.Empty;
    int playerId = 0;

    switch (msg)
    {
        case NetworkMessages.ChangeName:
            playerId = (int)e.Message.ReceiveData.Read(typeof(int));
            string newname = e.Message.ReceiveData.ReadString();
            newtext = string.Format
                ("DPlay UserId 0x{0} changed name to {1}",
                 playerId.ToString("x"), newname);

            break;
        case NetworkMessages.CheckPlayers:
            int count = (int)e.Message.ReceiveData.Read(typeof(int));
            newtext = string.Format
                ("Server reports {0} users on the server currently.",
                 count);
            break;
        case NetworkMessages.RunAway:
            playerId = (int)e.Message.ReceiveData.Read(typeof(int));
            newtext = string.Format
                ("Server reports DPlay UserId 0x{0} has ran away.",
                 playerId.ToString("x"));

            break;
        case NetworkMessages.Wave:
            playerId = (int)e.Message.ReceiveData.Read(typeof(int));
            newtext = string.Format
                ("Server reports DPlay UserId 0x{0} has waved.",
                 playerId.ToString("x"));

            break;
    }
    // We received some data, update our UI
    this.BeginInvoke(new AddTextCallback(AddText),
        new object[] { newtext });
}
```

You can see that this is essentially a simplified version of the server's handler. All you ever do is get the message and update the UI to notify the user about what's going on. You also don't even bother listening for the SendData action, since the server will never send that message.

Handling the Server Lost Event

The last thing you will need to cover is ensuring that the client can react correctly if the server is lost. You've handled this in the peer-to-peer session already, and the code here is identical. First, you add a hook for the `SessionTerminated` event:

```
connection.SessionTerminated += new  
    SessionTerminatedEventHandler(OnSessionTerminate);
```

Plus you'll need the actual event handler:

```
private void OnSessionTerminate(object sender, SessionTerminatedEventArgs e)  
{  
    this.BeginInvoke(new DisconnectCallback(OnDisconnect), null);  
}  
private void OnDisconnect()  
{  
    EnableSendDataButtons(false);  
    AddText("Session terminated.");  
    connected = false;  
  
    // Dispose of our connection, and set it to null  
    connection.Dispose();  
    connection = null;  
}
```

All you really want to do in this scenario is make sure you dispose the connection and nullify the object. However, it's good practice to inform the user by updating the UI and disabling the buttons.

In Brief

- - Creating dedicated servers running on particular ports
- - Connecting to these servers using the client interface
- - Tracking player joining and disconnecting
- - Sending game-specific data

In our next chapter, we will look at advanced networking topics, including data throttling, bandwidth detection, statistics, launching applications, and even voice communication.

Chapter 20. Understanding Advanced Networking Features

IN THIS CHAPTER

- [Having a Detailed Look at the Event Model](#)
- [Detecting Bandwidth and Network Statistics](#)
- [Launching Applications Using a Lobby](#)
- [Making Your Application Lobby Aware](#)
- [Adding Voice Chat to Your Sessions](#)
- [In Brief](#)

This chapter will cover various features of DirectPlay that will allow us to enhance our networking performance and features, including

- A detailed look at all events and why they occur.
- Bandwidth and lag detection.
- Sending queue.
- Lobby-launching applications.

Using Voice chat in your application.

[\[Team LiB\]](#)

Having a Detailed Look at the Event Model

You've dealt with the basics of client/server sessions, and peer-to-peer as well. However, thus far you haven't really gotten into the advanced features of networking or how to use them. This chapter will begin to look further into how you should create an efficient networking layer.

You've looked at a few of the possible events that can be fired during the course of a networking session, but haven't yet looked at the exhaustive list and why they actually occur. Some of these events will not occur for certain class types, so the text will be sure to mention which class the event applies to. Without further ado, the events:

- ApplicationDescriptionChanged— Applies to Peer, Server, and Client. This event is fired when the application description has been changed; for example, possibly the name has been changed. If you care, you will need to get the new application description when you receive this event.
- AsyncOperationComplete— Applies to Peer, Server, and Client. This event is fired when an asynchronous operation has been completed. Any of the DirectPlay methods that can return an asynchronous operation handle (for example, the Connect method or the Receive method) will notify the application when they have completed via this event. It is also possible to cancel these methods by using the CancelAsyncOperation method and passing in either the handle or a flag to cancel all operations of a particular type. You should remember that certain operations are automatically cancelled by other actions. For example, the FindHosts method is automatically cancelled when you connect to a session.
- ConnectComplete— Applies to Peer and Client. We've already covered this event when dealing with our client and peer sessions. This event is fired after attempting to connect to a session, even if that connection was not successful. You can check the ResultCode member of the event arguments object to determine whether or not the connection was successful. For example, if the host rejected your connection, you would expect the result code to be HostRejectedConnection.
- FindHostQuery— Applies to Peer and Server. This event is fired when a new connection has called FindHosts and is attempting to find your session. If you do not capture this event or if you do not modify the RejectMessage property of the event arguments, the caller of FindHosts will successfully enumerate this session. Setting the RejectMessage property to true before leaving this event handler will cause the enumeration to be rejected, and the caller will not be able to see this session. A common practice is to use the ReceivedData from the FindHosts call to determine if this enumeration should be successful.
- FindHostResponse— Applies to Peer and Client. This event is fired once a host is found. The event arguments object returned in this event will contain enough information to connect to the server that was found, including the hosts address object and the application description. Since the enumerations happen continuously, if you do not connect to a host immediately, it's possible (and probable) that the same host will be found multiple times.
- GroupCreated— Applies to Peer and Server. DirectPlay allows the creation of groups, which naturally are a collection of players and even other groups. The hosts of a server session or any peer in a peer-to-peer session can call the CreateGroup method to create a new group. After this method has been called and the group has been created, this event will be fired.
- GroupDestroyed— Applies to Peer and Server. When a group is about to be destroyed, this event will be fired. A group can be destroyed in multiple ways, either because DestroyGroup was called, or because the session is ending. If the AutoDestruct flag was used when creating this group, the group will also be destroyed

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Detecting Bandwidth and Network Statistics

At least currently, network bandwidth can be a difficult problem to work around. The majority of people on the internet today still have "low speed" modem-based connections. Even those with high-speed or broadband connections (which are steadily increasing) have a maximum upload and download speed. The goal of the network programmer is to ensure that there is little enough information going through the wire so as to not take up all available bandwidth, while at the same time ensuring that enough data is passed through to maintain the information needed.

The application you are writing for has a lot of effect on this as well. For example, if you are writing the server code for a massive online multiplayer game, then you would expect that server to have a fat pipe, with tons of available bandwidth. It's expected to maintain connections with thousands of clients simultaneously, each of them constantly sending data to the server. Plus, the server itself must send out data to each client quite often as well. Tuning the data throughput for this application is much different than, say, for the client of this server. The client only sends a small amount of data, and only to one spot, the server. It also only receives a small amount of data from that same server.

DirectPlay's API is generic enough to allow you to connect the information you need in order to make decisions on how much data you are passing across clients. If you wanted to get an idea of the current stats of your session, you could call the `GetConnectionInformation` method on your session. For a server or peer object, you must pass in the player id for the player you wish to receive this information on; the client will return the information about its connection with the server. Let's look at a sample output of this data:

```
Connection information:
PacketsDropped: 0
BytesDropped: 0
PacketsRetried: 0
BytesRetried: 0
PacketsSentNonGuaranteed: 1
BytesSentNonGuaranteed: 0
PacketsSentGuaranteed: 99
BytesSentGuaranteed: 1344
PeakThroughputBps: 116
ThroughputBps: 80
RoundTripLatencyMs: 4
MessagesReceived: 79
PacketsReceivedNonGuaranteed: 0
BytesReceivedNonGuaranteed: 0
PacketsReceivedGuaranteed: 79
BytesReceivedGuaranteed: 1836
MessagesTimedOutLowPriority: 0
MessagesTransmittedLowPriority: 0
MessagesTimedOutNormalPriority: 0
MessagesTransmittedNormalPriority: 99
MessagesTimedOutHighPriority: 0
MessagesTransmittedHighPriority: 1
```

As you can see, there is a wealth of information here. You can easily see the number of packets (and bytes) that were dropped (only non-guaranteed packets would be dropped).

You can see the number of packets (and bytes) that were retried (guaranteed packets that failed to reach the destination). The number of packets (and bytes) you send, both guaranteed and non-guaranteed, is included as well.

The next few members are really where this information is useful. You can see the peak throughput of the pipe in bytes per second, as well as the average throughput. You could use this information to determine whether you are sending too much data, or possibly not enough. For example, if the average throughput was 100 bytes per second, and you're sending 30K of data per second, you are killing the bandwidth. However, if the opposite were the case, you'd have plenty of extra bandwidth to fill with extra information.

Another useful piece of information you can see here is the average round trip latency. This is the approximate time it takes any packet to be received by the remote system and returned to you. Long round trip latencies are another

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Launching Applications Using a Lobby

Anyone who's ever played games someplace such as MSN Games by Zone.com (<http://zone.msn.com>) understands the concept of a lobby, even if you don't realize it. A lobby is essentially where a group of players get together before they actually start playing the game. The game can then be "launched" from the lobby and all of the players in that lobby will automatically start the game and connect to each other.

Looking at the DirectPlay assemblies, you may have noticed that this is the first assembly that has "sub-namespaces." To group functionality inside this assembly, there are two extra namespaces included: Lobby and Voice. We will get to the Voice functionality later in this chapter, so for now we will concentrate on the Lobby namespace.

There are two main classes in the Lobby namespace that will control how you interact with lobby-aware applications: Application and Client. The Client class is used to enumerate and launch lobby-enabled applications on your local system. The Application class is used by the application that supports the lobby. Each of these classes has an event model that is similar to the one used by the Peer, Server, and Client classes, as well.

The easiest thing to do is to list the programs that are currently able to be launched via a lobby, and then launch them, so let's try that first. Create a new windows form application, adding a list box with the dock parameter set to fill, so that it encompasses the entire form. Also make sure that you have included the reference to the DirectPlay assembly and included the using clause for the lobby namespace as shown here:

```
using Microsoft.DirectX.DirectPlay.Lobby;
```

As we've already mentioned, the Client class will control enumerating and launching a lobby-enabled application, so we will need to declare this variable:

```
private Client connection = null;
```

Before we forget, we should also make sure we dispose of this object when the application quits. Add the following to the Dispose override:

```
if (connection != null)
    connection.Dispose();
```

You'll notice that the class name (Client) exists both in the core DirectPlay namespace as well as the Lobby namespace. You'll need to make sure that you include both of these namespaces in your using clauses to fully qualify these variables. You may also notice that the other class (Application) also exists in the System.Windows.Forms namespace, so you will need to fully qualify any references to that as well.

Now, let's fill our list box with the names of the applications that are registered as being lobby-aware. Add the following code to your forms constructor:

```
// Fill the list box
connection = new Client();
foreach(ApplicationInformation ai in connection.GetLocalPrograms())
{
    listBox1.Items.Add(ai.ApplicationName);
}
```

As you see, this is a relatively simple operation. After we create our lobby client object, we simply enumerate through each of the local programs registered and write the name of that application into our list box. Launching these applications is equally easy; add an event handler for the double-click event of the list box. Use the code found in [Listing 20.1](#).

Listing 20.1 Launching an Application

[\[Team LiB\]](#)

Making Your Application Lobby Aware

The Application class in the Lobby namespace is how you can actually receive information after you've been launched from a lobby, as well as how you register your application as being lobby-aware. The act of registering your application is quite simple; we will look at this first.

In order to register your application, you will first need to create a ProgramDescription structure. At the very least, you will need to fill out the application GUID and name parameters, as well as the executable path and filename members. The others are optional. Then you can call either the RegisterProgram or UnregisterProgram method, depending on the operation you wish to perform.

We never actually looked at the constructor for the Application object yet though. While it's possible to create the object without any parameters, the majority of time you will want to detect whether you have been launched from a lobby client. Let's look at the overload with the most parameters, as every other overload is a subset of this one:

```
public Application ( System.Int32 connectionHandle ,  
    Microsoft.DirectX.DirectPlay.Lobby.InitializeFlags flags ,  
    Microsoft.DirectX.DirectPlay.Lobby.ConnectEventHandler connectEventHandler )
```

The connection handle parameter is an out parameter that will be returned to you. If you were launched by a lobby client, this handle will be the same as the one returned from the ConnectApplication method. However, if you were not, you can expect this member to be zero. The InitializeFlags enumeration allows you to disable parameter validation, but for this context is wholly uninteresting. The last parameter is something we haven't seen before. This allows you to actually hook the Connect event before the object is created, mainly because the constructor can actually fire this event.

Once you've detected that you've been launched from a lobby client, you need to call the RegisterLobby method on the appropriate DirectPlay connection object you are using. In the Connect event handler, you can also detect whether there were any connection settings passed in, and if so, use them to get into a session. The ConnectionSettings structure that is included in the event arguments object has all the information you need to either host or join an existing session.

[\[Team LiB\]](#)

Adding Voice Chat to Your Sessions

Sitting around and typing chat messages while playing your favorite online games seems to be a common occurrence. Even though these players can type quite quickly, and use a form of shorthand many times, the fact remains that in many types of games, the act of stopping to type a chat message leaves you vulnerable, even if only briefly. What you need is a hands-free way to communicate that allows you to keep playing the game. As humans, we've relied on voice communication for thousands of years, and unless you're using sign language, it's completely hands-free.

Adding voice chat into your application can be quite simple, although with a little extra work you can have full control over everything. To handle this simple case first, let's take the peer-to-peer session application we wrote in [Chapter 18](#), "Simple Peer-to-Peer Networking," and add voice to it. No fancy options, just allow everyone in the session to speak with everyone else.

Before we begin, we will need to add a few things to the project file. First, the voice communication used in DirectPlay goes through DirectSound, so we will need to include a reference to the DirectSound assembly. Since there are Server and Client classes in our voice namespace as well, we should also add the following using clauses:

```
using Voice = Microsoft.DirectX.DirectPlay.Voice;
using Microsoft.DirectX.DirectSound;
```

In order to actually use voice during a session, we will need both a client and a server object. The server will be responsible for starting the voice session, while the client will naturally connect to the server's session. In a peer-to-peer session, where the host is also a member of the session, that peer will need to be both the server and a client of the voice session. Let's add our variable declarations:

```
private Voice.Client voiceClient = null;
private Voice.Server voiceServer = null;
```

Naturally, without a server, the client has nothing to connect to, so we should go about creating a server first. Find the section of code where we first host our peer-to-peer session. Directly after the EnableSendDataButton call in the InitializeDirectPlay method, add the following code:

```
// Create our voice server first
voiceServer = new Voice.Server(connection);

// Create a session description
Voice.SessionDescription session = new Voice.SessionDescription();
session.SessionType = Voice.SessionType.Peer;
session.BufferQuality = Voice.BufferQuality.Default;
session.GuidCompressionType = Voice.CompressionGuid.Default;
session.BufferAggressiveness = Voice.BufferAggressiveness.Default;

// Finally start the session
voiceServer.StartSession(session);
```

As you can see, when we create our voice object, we also need to pass in the DirectPlay object that will act as the transport for the voice communication. This will allow the voice to be transmitted along with the other data in your session. Before we can actually start a session, though, we need to have a basic way to describe the session.

Since we are in a peer-to-peer networking session, we choose the peer session type for our voice communication. This sends all the voice data directly between all of the players. Other types of sessions are

- **Mixing**— In this mode, all voice communication is sent to the server. The server then mixes the combined voice data and forwards it on to the clients. This dramatically reduces CPU time and bandwidth on the client, but subsequently raises each for the server.

[\[Team LiB\]](#)

In Brief

In this chapter we covered the more esoteric aspects of DirectPlay, including

- - A detailed look at all events and why they occur.
- - Bandwidth and lag detection.
- - Sending queue.
- - Lobby launching applications.
- - Using Voice chat in your application.

In our final chapter, we will look at putting most everything we've learned thus far together.

Chapter 21. Achieving Maximum Performance

IN THIS CHAPTER

- [Using Value Types as Objects](#)
- [Understanding the Performance Implications of the Event Model](#)
- [Understanding the Cost of Methods](#)
- [In Brief](#)

Items covered will include

- Boxing and unboxing
- The event model and its detriments
- Understanding the cost behind certain methods

[\[Team LiB\]](#)

Using Value Types as Objects

One of the biggest questions people ask about writing managed code for things like games is "Is it fast enough?" Speed is very important to an industry that prides itself in getting high-quality games that look near photo-realistic, all in essentially real time. The basic answer is yes, so long as you know how to write high-performance code. Of course, that's the same answer that was given when the switch from assembly to C was made, and that turned out all right.

The .NET Runtime and the C# language have changed the way people write code. They have opened up the world of programming to new developers, and expanded the productivity of the ones that have been around for a while. They handle so many things for you that it's quite easy to forget how much some operations may cost. Look at the real-world example in the following paragraph.

The Billboard sample that ships with the DirectX SDK has multiple versions: an unmanaged version written in C++, along with two managed versions, one written in C# and the other in VB.NET. Since each of the DirectX SDK samples includes the frame rate while you are running, you can easily tell which of the applications is running faster, and by approximately how much. Comparing the C++ Billboard sample with the C# Billboard sample, you'll notice the C# version runs at approximately 60% of the speed of the C++ sample.

Considering that the other samples run at similar speeds between the managed and unmanaged variations, there must be something special about this particular sample that causes this slowdown. Naturally, there is, and the culprit is boxing and unboxing.

The term "boxing" refers to what the .NET runtime does to convert a value type (for example a structure) into an object. The reverse of this process is obviously called "unboxing." In order to perform this operation, the .NET runtime will allocate a portion of the heap large enough to hold your value type, and then copy the data from the stack (where the value type resides) to the newly allocated heap space.

Now, when looking at the Billboard managed sample, you may notice that each tree that will be drawn has a corresponding structure that maintains the information needed to draw the tree. Since the trees are alpha blended, and will need to be drawn in a certain order (from back to front), each tree is sorted every frame. You'll see the code in the sample do this:

```
trees.Sort(new TreeSortClass());
```

This class implements `IComparer`, which is needed for comparison. If you look at the implementation of this class, though, you'll see quickly that the comparison method takes two objects in as the parameters, while we are using a structure. This structure must be boxed before it can be passed on to the compare method. Then, as soon as the compare method is started, the object is unboxed to be able to manipulate the data.

The sort method will be called approximately 4,300 times per frame. Each of these calls will perform a total of two boxing operations, followed immediately by two unboxing operations. The structure itself is defined as such:

```
public struct Tree
{
    public CustomVertex.PositionColoredTextured v0, v1, v2, v3;
    public Vector3 position;
    public int treeTextureIndex;
    public int offsetIndex;
};
```

Now, if you calculate the structures size, you'll see that it is quite large: 116 bytes. So, calculate the data that is being allocated and copied during a single sort operation. Multiply the number of bytes that will need to be allocated per object (116), by the number of objects (2), by the number of calls per frame (4,300), and you will come up with a whopping 997,600 bytes that will need to be allocated for every frame. This just covers the allocation for the boxing operation; it doesn't even consider the copy operation to get the data into the newly allocated object.

[\[Team LiB\]](#)

[\[Team LiB\]](#)

Understanding the Performance Implications of the Event Model

There's a similar scenario that is specific to Managed Direct3D. You've already learned that each resource that is created hooks the device events for at least disposing, and in many cases, device lost and device reset as well. Each of these event hooks can do a few things that can be disastrous to your performance if you aren't aware of them.

First, there is the actual memory allocation. Every time you create a vertex buffer, for example, aside from the normal vertex buffer allocations, just to cover the event handlers, there are three extra allocations of delegates. Couple this with the fact that these event handlers act as hard links to the object itself, and you can see that these objects will never be destroyed until the device itself is. Look at this piece of sample code:

```
public void WasteVideoMemory()
{
    VertexBuffer waste = new VertexBuffer(device, 5000,
        0, VertexFormats.None, Pool.Default);
}
```

It looks like a piece of dead code. Sure, a vertex buffer is created, but it's never used, so obviously the garbage collector will kick in and collect it. This is what you might expect anyway; however, nothing could be further from the truth.

In reality, even this simple piece of code does quite a bit outside of just creating a vertex buffer. After the vertex buffer has been created, the DeviceLost, DeviceReset, and Disposing events from the device are all hooked. These event handlers will maintain a link between the device and the vertex buffer. Even though this vertex buffer is never used anywhere, the garbage collector will never try to collect it because the device still maintains a reference to the object via the event handler.

If a method similar to the preceding one was called numerous times, you would slowly erode away the amount of free video memory you have (since the object was created in the default memory pool). Since you never maintained a reference yourself to this vertex buffer, once it has lost scope, you can never remove it. In a short time you would run out of memory, and be puzzling to figure out why.

Another side effect of this behavior is shutdown time. There have been a few people who have reported "bugs" in the Managed Runtime due to a misunderstanding between how they thought things should work and the reality of what really goes on. It's normally the same story, "I'm doing such and such, and my application locks up when I exit." The scenario is invariably something like this:

```
device.SetRenderTarget(0, mySwapChain.GetBackBuffer(0, BackBufferType.Mono));
// Render some stuff
```

This code does pretty much exactly the same thing the previous code did. It creates a "new" object (in this case, a surface) that is never used again. Running this line of code thousands of times creates thousands of surfaces that are orphaned and only accessible from the device. When the device is finally disposed on application shutdown, the device signals to each of its dependencies (these thousands of surfaces) that they need to dispose themselves now as well. This "event unwinding" takes a period of time; the more orphaned objects, the longer it takes. Most times, the customer's application hasn't locked up; it's simply trying to handle the events.

There are two ways around this type of behavior. The first one is the easiest to implement. You can simply use the using statement on objects you expect to have a short lifetime. The preceding statement should be

```
using(Surface backBuffer = mySwapChain.GetBackBuffer(0, BackBufferType.Mono))
{
    device.SetRenderTarget(0, backBuffer);
    // Render some stuff.
}
```


[\[Team LiB\]](#)

Understanding the Cost of Methods

One of the most important aspects of ensuring that you write applications with the best performance is to always know the cost of any method you call. While memorizing all the different permutations of APIs you can call and the performance they take is unreasonable, you should have at least a vague idea about the cost of any method.

In Managed DirectX there are a few cases where seemingly harmless code can cause problems. For example, any method that takes a string as an argument in Managed DirectX has a cost of allocating a new unmanaged string type and copying the string data over.

Take the case where you are using HLSL to render your scene, and numerous times throughout each frame, you switch the technique being used, such as

```
myEffect.Technique = "TransformWorldSpace";  
//Render some stuff  
myEffect.Technique = "TransformAndAddGlow";  
// Render some other stuff
```

Each time you switch the technique, you are allocating some memory and copying the technique name over. A much better solution to this problem would be to cache the handles returned, such as

```
// On device creations, cache technique handles  
EffectHandle handle1 = myEffect.GetTechnique("TransformWorldSpace");  
EffectHandle handle2 = myEffect.GetTechnique("TransformAndAddGlow");  
  
// Later on, for every render  
myEffect.Technique = handle1;  
//Render some stuff  
myEffect.Technique = handle2;  
//Render some stuff
```

Another common case of not understanding the cost of methods was covered earlier when dealing with resources. The difference between locking a resource and returning an Array or a GraphicsStream is quite drastic.

[\[Team LiB\]](#)

◀ PREVIOUS

NEXT ▶

In Brief

- - The horrors of boxing and unboxing.
- - Understanding the cost of all methods.
- - The pitfalls of using the event model.

[\[Team LiB\]](#)

◀ PREVIOUS

NEXT ▶

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Part VII: Appendices

A [Using the Diagnostics Assemblies](#)

B [Playing Music and Videos](#)

[\[Team LiB \]](#)

◀ PREVIOUS

NEXT ▶

Appendix A. Using the Diagnostics Assemblies

IN THIS APPENDIX

- - [Enumerating Everything in the System](#)
 - [Enumerating Specific Items You Care About](#)

You've just released your game to the public. Your test team has done a remarkable job, but you never got the chance to test your application on every possible set of systems out there. It seems that approximately 5% of all people who have paid for your game can't play it for some reason, and you don't know why. You've got a sneaky suspicion that it is a certain type of video card, but you can't be sure. What if you had a way to report the current items installed on the system in your game, and return the list to you? The Diagnostics namespace can be used for this. In this appendix we will cover these topics, including

- - Enumerating every option the diagnostic software knows about and reporting them
 - Checking for certain options we care about

[\[Team LiB\]](#)

Enumerating Everything in the System

The diagnostics namespace is built on the same code base that the DxDiag tool ships in the operating system and versions of DirectX. If you've ever run this tool, it can tell you just about everything you ever wanted to know about your system.

The diagnostics data is arranged in a hierarchal fashion. You have containers, which can contain zero-many properties, as well as zero-many children containers. Enumerating all objects is simple to do with a recursive function. In [Listing A.1](#), look at the function that is included with the DirectX SDK DxDiagOutput sample:

Listing A.1 Outputting Diagnostic Data

```
static void OutputDiagData(string parent, Container root)
{
    try
    {
        foreach (PropertyData pd in root.Properties)
        {
            // Just display the data
            Console.WriteLine("{0}.{1} = {2}", parent, pd.Name, pd.Data);
        }
    }
    catch
    {
    }

    try
    {
        foreach (ContainerData cd in root.Containers)
        {
            // Recurse all the internal nodes
            if (parent == null)
                OutputDiagData(cd.Name, cd.Container);
            else
                OutputDiagData(parent + "." + cd.Name, cd.Container);
        }
    }
    catch
    {
    }

    // We are done with this container, we can dispose it.
    root.Dispose();
}
```

If there are any properties in this container, we will enumerate each of them, and display them to the user, as well as the value they contain. The values will always be either a string, Boolean, or integer value, depending on the property. You can check the `pd.Data.GetType()` method to determine the type if it is important to you.

After we've dealt with the properties for this container, we need to check whether this container contains any children containers. If it does, we enumerate each of those, and for every one we find, we simply call this method once more recursively.

The only thing left to do for this to be useful is to create the first container and make the initial call into this method. The container object contains only one constructor that takes one Boolean value. This value is used to determine whether the diagnostics should include WHQL (Windows Hardware Quality Labs) information as well. Retrieving this information can be time consuming, so unless you need it, it's probably better to pass in false here. The DirectX SDK sample uses this entry into the executable (see [Listing A.2](#)):

Listing A.2 Starting the Application

[\[Team LiB\]](#)

Enumerating Specific Items You Care About

If you've run DxDiag, or even the sample app from the DirectX SDK, you know that enumerating everything that DxDiag knows about can take quite some time. What if you don't care about the vast majority of the stuff listed, and only want to maintain information on a small subset of items? Luckily, the diagnostic namespace can be used for this as well.

Let's say the first thing you wanted to know was what version of DirectX the system running your application was using. No need to go through all of the work enumerating everything when you only care about something so simple. Here is the code you would use to determine this:

```
Container parent = new Container(false);
Container child = parent.GetContainer("DxDiag_SystemInfo").Container;
int dxVersionMajor = (int)child.GetProperty("dwDirectXVersionMajor").Data;
int dxVersionMinor = (int)child.GetProperty("dwDirectXVersionMinor").Data;
string dxVersionLetter = (string)child.GetProperty("szDirectXVersionLetter").Data;
Console.WriteLine("DX
Version:{0}.{1}{2}", dxVersionMajor, dxVersionMinor, dxVersionLetter);
```

Here, we create the root container (without any WHQL information). We then get a child container from this root named DxDiag_SystemInfo. We then get three different properties of the DirectX version: the major version number, the minor version number, and then any "letter" associated with this version of DirectX (since DirectX is notorious for releasing versions such as DX8.1b).

You may be wondering where these property names came from, and although it isn't obvious initially, all of these names are discoverable. As we were enumerating the properties and containers, you can see that we printed out the name of each as we went. You would need to write the enumeration code at the very least (or use the DX SDK sample) to determine the container and property names for the information you want to get.

The property names prefixes also give you the default data type for the object. They are all given names that match Hungarian notation for C developers. Items with the prefix "sz" denote strings. Items with the prefix "b" are Boolean values. All other prefix values denote integer values.

Using the Diagnostics namespace can be an invaluable way to help track down issues in your application. Your application should use it for error reporting, at the very least for debugging purposes. It's even possible to use this in your shipping game to help support when trying to figure out why your game refuses to run on certain customers' computers. The time saved by detecting common errors (old outdated drivers, for example) and allowing support to fix them quickly is significant. Who knows, it's possible a bug could be found in your game that only happens on certain video cards; you wouldn't have been able to tell without getting this information from multiple customers having similar problems.

It's better to plan for problems early than to wait and be sorry later.

Appendix B. Playing Music and Videos

IN THIS APPENDIX

- [Playing Back an Audio or Video File Simply](#)
- [Using the Features of Video Files](#)
- [Using Video as Textures](#)

The Managed DirectX SDK included APIs to talk with each of the DirectX APIs, with the exception of DirectMusic and DirectShow. While the reasons for this decision could be debated, we understood early on that at the very least, developers were going to want (and need) a way to play DirectShow content, namely movies and encoded music (such as mp3s and wma files). We wanted a simple and easy-to-use interface for accessing this basic functionality, and thus the AudioVideoPlayback namespace was born. In this appendix we will cover this namespace, including

- Simple playback of an audio file
- Simple playback of a video file
- Playing a video file in your own window
- Using a video file as a texture in your 3D application

Playing Back an Audio or Video File Simply

In order to use the audio and video playback namespace, you will first need to add a reference to this namespace into your application. You may also want to include a using clause for the namespace in your code file.

Our goal when we designed these classes was to make them extremely simple to use, so if we wanted to play a file, it should be accomplished with just one line of code. If you wanted to play an audio file called "piano.mp3", let's say, the code would look like this:

```
Audio someAudio = new Audio("piano.mp3", true);
```

There are two main classes that are included in this namespace: the video class and the audio class. These names are pretty self-explanatory. Each class's constructor has two overloads; the first parameter of each overload is the name of the file you wish to play. The other constructor for each class contains a Boolean value that determines whether or not this file should begin playing automatically. The default option is false, and the file will not begin playing until you call Play on it.

As you can see by our call, we are creating an audio object, and we want it to begin playing automatically. For the simplest of cases, this is all that is required for you to do. Using the constructor without the Boolean parameter, or specifying false for the parameter when you call the constructor, will cause the file to be loaded, but it will not be played.

The constructors aren't the only way to create (or load) these files, though. The most similar method would be the static FromFile method, which takes the same parameters as the constructor does and returns a new instance of the class. There is also a static FromUrl method, which behaves exactly the same as the FromFile method, with the exception that it can be used to load the data from a Web site (or any valid URL). If possible, this data will be streamed from this URL to your system, so playback should begin much quicker than waiting for the entire file to be downloaded.

There are also Open and OpenUrl methods on the classes themselves that take the same parameters as their static counterparts. They will replace the data in an already created object with the data from the new file or URL you specify.

Using the Features of Video Files

In the previous example, we used an audio file as our "test" to show how simple it was to load and play the content. What if we wanted to do the same thing with the video file "butterfly.mpg" instead? Simple enough; just use the following:

```
Video someVideo = new Video("butterfly.mpg", true);
```

If you ran code similar to this in your application, the very first thing you would notice is that the video plays in an entirely new window. While this may be adequate for you, what if you didn't want that? What if you already had a window where you wanted to display your movie? Of course, this is quite simple to do; however, it requires an extra couple lines. First, you will want to skip the auto play feature. Instead, use code like this:

```
Video someVideo = new Video("butterfly.mpg");  
someVideo.Owner = this;  
someVideo.Play();
```

The default option is not to auto-play the data, so we use the constructor that takes a single argument. We then set the owner of the video file, which can be any Windows Forms control. For example, it could be your main form itself, or a picture box that is a child of your form. Finally, we play the video. Notice now that the video is played inside the window you specified in the owner property?

You can also control the audio of a video (if it contains audio data) directly from the video object. You will notice that the video object has a property "Audio" which will return the Audio object associated with this video. You may use this just like you would any other audio object.

[\[Team LiB\]](#)

◀ PREVIOUS

Using Video as Textures

Another common feature for video files that users must have is the capability to play these movies onto a 3D texture inside the application they are building. Many of the developers use movie files as the "cut-scenes" during their game, and need some way to show the movie in full screen mode. The Audio Video playback classes can handle both cases quite well.

For the simplest case of a developer wanting to use a full screen movie as a cut-scene in the game they are writing, we made it as simple as could be. Simply load the video file into your Video object, set the "Fullscreen" property to true, and play the video. Nothing could be simpler...well, unless there was a constructor that took these parameters.

For the more difficult scenario where the developer has some set of geometry they want to texture using a video, we've come up with a different solution. First, there is an event you will want to hook, TextureReadyToRender. This event will be fired every time there is a new frame in the video file that is ready to be rendered as a texture.

The TextureRenderEventArgs member that is passed into the event handler contains the newly created texture that is ready to be rendered. This event may be fired from many different threads, so ensure that you do not have multiple threads accessing variables simultaneously. Who knows what craziness you can end up with doing that? All you need to do to start the video playing and the event firing is call the method RenderToTexture on the video object. You should begin receiving the events and textures immediately.

The sample that ships with the DirectX SDK renders the entire scene at the same frame rate as the movie being played; it only renders each frame when a new texture is ready. In many cases, this is adequate for the developers, since movies being rendered as a texture is a fairly uncommon occurrence. However, you may still want to use the texture in your "real time" rendering application, so how would you go about that?

First, you will need to detect whether your video texture has been created in the system memory pool or the default pool (since you won't have control over this). You can do this by checking the level description of the first level of the texture.

If the texture was created in the default pool (most common), you will need to use the Direct3D device's method GetRenderTargetData on the surface of the texture and a surface of your own created in the system memory pool. You cannot render textures created in the system memory pool, so you will also need to create a texture in the default memory pool to use for rendering. Something like

```
SurfaceDescription ds = e.Texture.GetLevelDescription(0);

if (ds.Pool == Pool.Default)
{
    systemSurface = device.CreateOffscreenPlainSurface(ds.Width, ds.Height,
        ds.Format, Pool.SystemMemory);
}

texture = new Texture(device, ds.Width, ds.Height,
    1, Usage.Dynamic, ds.Format, Pool.Default);
```

This code will check to see whether your video texture has been created in the default memory pool, and if it has, it will create an off-screen plain surface to hold the texture data, plus a texture in the default memory pool that we can use to render our scene. After this is executed, there are two methods on the device that we need to use to get the data from one texture to the next:

```
using(Surface videoSurface = e.Texture.GetSurfaceLevel(0))
{
    using(Surface textureSurface = texture.GetSurfaceLevel(0))
    {
        device.GetRenderTargetData(videoSurface, systemSurface);
        device.UpdateSurface(systemSurface, textureSurface);
    }
}
```

[\[Team LiB\]](#)

◀ PREVIOUS

[\[TeamLiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[Y\]](#) [\[W\]](#)

[\[TeamLiB\]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

2D rendering

[DirectDraw](#)

[full-screen device creation](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#) [9th](#) [10th](#) [11th](#) [12th](#) [13th](#) [14th](#)

3D scenes

[geometry](#)

[3D sound](#) [2nd](#) [3rd](#) [4th](#) [5th](#)

[Buffer3D object](#)

[DirectSound API](#)

[InitializeSound method](#)

[Listener3D object](#) [2nd](#) [3rd](#) [4th](#)

[\[Team LiB \]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[actions in sessions, DirectPlay 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[AdapterDetails structure](#)

[adapters](#)

[enumeration 2nd 3rd 4th 5th 6th 7th 8th](#)

[addresses](#)

[DirectPlay 2nd 3rd 4th 5th 6th 7th](#)

[defining](#)

[TCP/IP 2nd](#)

[adjacency information](#)

[meshes](#)

[generating](#)

[adjacencyIn parameter](#)

[cleaning and](#)

[adjacencyOut parameter](#)

[cleaning and](#)

[All Items node](#)

[input devices and](#)

[AllParameters property](#)

[ambient light](#)

[games](#)

[ambient lighting](#)

[color values](#)

[white](#)

[reflecting](#)

[animation](#)

[meshes](#)

[loading 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[rendering animated 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th 17th](#)

[skeletons](#)

[programmable pipeline and 2nd 3rd](#)

[skeletal](#)

[skeletal animation](#)

[bone offset matrices 2nd](#)

[indexed mesh, animated bones 2nd 3rd 4th 5th 6th 7th 8th](#)

[sprites 2nd 3rd 4th 5th](#)

[DirectDraw 2nd 3rd 4th 5th](#)

[updates 2nd](#)

[timer](#)

[AnimationRootFrame variable](#)

[skeletal animation](#)

[APIs \(Application Programming Interfaces\)](#)

[DirectSound](#)

[applications](#)

[lobbies](#)

[awareness 2nd](#)

[ApplicationDescription structure](#)

[ApplicationDescriptionChanged event](#)

[applications](#)

[launching](#)

[lobbies and 2nd 3rd 4th 5th](#)

[arguments](#)

[DeviceType](#)

[renderWindow](#)

[arrays](#)

[of meshes](#)

[\[Team LiB\]](#)

[[Team LiB](#)]

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[back face culling](#)

[bandwidth 2nd 3rd 4th 5th 6th](#)

[BaseTexture class](#)

[BeginInvoke method](#)

[BeginScene method 2nd](#)

[blended textures](#)

[color and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[bone offset matrices, skeletal animation](#)

[set up](#)

[Boolean variables](#)

[connection checking](#)

[split meshes](#)

[box index buffer data, listing](#)

[box method](#)

[mesh creation](#)

[boxes](#)

[drawing](#)

[index buffers and](#)

[left-handed coordinate system](#)

[boxing](#)

[buffer](#)

[locking](#)

[default mechanism](#)

[Buffer3D object](#)

[properties 2nd 3rd](#)

[BufferDescription object](#)

[properties 2nd 3rd 4th 5th](#)

[buffers](#)

[depth buffers](#)

[rasterization and 2nd 3rd 4th 5th](#)

[DirectSound namespace](#)

[dynamic](#)

[Discard flag](#)

[NoOverwrite flag](#)

[formats](#)

[valid](#)

[index buffers 2nd 3rd](#)

[vertices and 2nd 3rd 4th 5th 6th 7th 8th](#)

[locking 2nd 3rd 4th 5th](#)

[controlling 2nd 3rd 4th](#)

[high performance](#)

[overloads](#)

[ranks parameter](#)

[ReadOnly flag](#)

[static](#)

[vertex](#)

[rendering and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th](#)

[vertex buffers 2nd 3rd](#)

[built-in effects types](#)

[[Team LiB](#)]

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

camera

depth position

[as variable](#)

options

[setting](#)

[text drawing](#)

Caps

[storage](#)

[Car class, Dodger game](#)

[car, adding to game](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#) [9th](#)

[chorus effect](#)

classes

[BaseTexture](#)

[Device](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#) [9th](#) [10th](#) [11th](#)

[GraphicsSprite](#)

[Line class](#)

[ProgressiveMesh](#)

[Resource](#) [2nd](#) [3rd](#)

[Sprite class](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#)

[cleaning meshes](#)

[parameters](#)

client connections

[initializing](#)

client/server sessions

DirectPlay

[connecting clients](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#)

[data packets, sending](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#)

[lost events on server](#) [2nd](#)

[received data, client](#) [2nd](#)

[DirectPlay and](#)

[dedicated server session](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#)

[ClientInformation event](#)

[Clone method](#)

cloning

[mesh data](#) [2nd](#) [3rd](#) [4th](#)

color

textures

[blended textures](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#) [9th](#) [10th](#)

color values

[ambient lighting](#)

diffuse lighting

[white](#)

[compression effect](#)

Connect method

[overloads](#)

[ConnectComplete event](#)

connections

[voice sessions, DirectPlay](#)

constants

[specular highlighting](#)

constructors

[device creation](#)

devices

[overloads](#)

[meshes](#)

[\[Team LiB\]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[D3DX \(Direct3D Extensions Library\)](#)

- [data packets](#)
 - [client/server sessions 2nd 3rd 4th 5th 6th 7th 8th](#)
- [dedicated server sessions](#)
 - [DirectPlay 2nd 3rd 4th 5th 6th](#)
- [depth buffers](#)
 - [rasterization and 2nd 3rd 4th 5th](#)
- [detail](#)
 - [progressive meshes and 2nd 3rd 4th 5th 6th](#)
 - [device class 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th](#)
- [Device class](#)
 - [DirectSound namespace](#)
- [device class](#)
 - [InitializeGraphics function](#)
- [Device property](#)
 - [Resource class](#)
- [device render states](#)
- [device reset event handler](#)
 - [animated meshes and](#)
- [device resets](#)
 - [automatic](#)
 - [resize and 2nd](#)
 - [device specific Control Panel](#)
- [DeviceCaps section](#)
- [DeviceReset event](#)
- [DeviceResizing event](#)
- [devices](#)
 - [capability checking 2nd 3rd 4th](#)
 - [Caps](#)
 - [storage](#)
 - [constructors](#)
 - [overloads](#)
 - [creating](#)
 - [constructors and](#)
 - [creation failures](#)
 - [defaults](#)
 - [full-screen rendering](#)
 - [DirectDraw 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th](#)
 - [full-screen rendering device 2nd 3rd 4th 5th](#)
 - [games](#)
 - [default state, setting](#)
 - [hardware](#)
 - [availability 2nd](#)
 - [initialization](#)
 - [DirectDraw](#)
 - [environment mapping](#)
 - [force feedback](#)
 - [lighting](#)
 - [states](#)
 - [transforms](#)
 - [user input](#)
 - [detection 2nd 3rd 4th 5th 6th](#)
 - [Treeview](#)
 - [DeviceType argument](#)
- [diagnostics](#)

[\[Team LiB\]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[echo effect](#)

[eer object](#)

[_initialization](#)

[EffectInstance parameter](#)

[effects](#)

[_built-in types](#)

[_GetEffects method](#)

[_SetEffects method](#)

[_sounds and 2nd 3rd 4th 5th](#)

[encapsulation](#)

[_meshes](#)

[_EndScene method](#)

[entry point](#)

[_game](#)

[enumeration](#)

[_adapters 2nd 3rd 4th 5th 6th 7th 8th](#)

[_diagnostics namespace 2nd 3rd 4th 5th 6th 7th](#)

[_mesh flags](#)

[environment maps](#)

[_device initialization](#)

[_rendering 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th](#)

[_view matrix](#)

[_environment reverberation effect](#)

[event](#)

[_DeviceReset](#)

[event handlers](#)

[_FindHosts method](#)

[_player management](#)

[sessions](#)

[_joining](#)

[event model](#)

[_DirectPlay 2nd 3rd 4th 5th 6th 7th](#)

[_performance and 2nd 3rd 4th 5th 6th](#)

[event model, networking 2nd 3rd 4th 5th 6th](#)

[events](#)

[_DeviceResizing](#)

[lost](#)

[_client/server sessions, DirectPlay 2nd](#)

[networking](#)

[_ApplicationDescriptionChanged](#)

[_AsyncOperationComplete](#)

[_ClientInformation](#)

[_ConnectComplete](#)

[_FindHostQuery](#)

[_FindHostResponse](#)

[_GroupCreated](#)

[_GroupDestroyed](#)

[_GroupInformation](#)

[_HostMigrated](#)

[_IndicateConnect](#)

[_IndicateConnectAborted](#)

[_PeerInformation](#)

[_PlayerAddedToGroup](#)

[_PlayerCreated](#)

[_PlayerDestroyed](#)

[\[Team LiB\]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

files

[loading meshes from](#)

filling

textures

[unsafe code](#)

fills

textures

[arrays](#)

filters

textures

[games](#)

[FindHostFlags method](#)

[FindHostQuery event](#)

[FindHostResponse event](#)

[FindHosts method](#)

[event handler](#)

[fixed-function pipeline](#)

[rendering and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th](#)

flags

meshes

[enumeration values](#)

[welding](#)

NoDirtyUpdate

[buffer locking](#)

NoOverwrite

[dynamic buffers](#)

ReadOnly

[buffer locking](#)

[flange effect](#)

flow control

[HLSL](#)

[specular highlighting and](#)

fonts

[declaring](#)

[performance enhancement](#)

[force feedback 2nd 3rd 4th](#)

formats

[conversiond detection](#)

[display](#)

Frame class

[skeletal animation](#)

frames

games

[per frame updates](#)

[full-screen rendering device 2nd 3rd 4th 5th](#)

[DirectDraw 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th](#)

functions

[DrawUserPrimitives](#)

[Optimize](#)

[RotationAxis](#)

[SetTexture](#)

[\[Team LiB \]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

game

 devices

[default state, setting](#)

 Dodger

[default options](#)

[finalizing 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[high scores](#)

 Dodger game

[obstacles 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th](#)

 movable car

[adding 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[game pads](#)

[DirectInput initialization](#)

games

 frames

[per frame updates](#)

 graphics

[initialization](#)

[main entry point](#)

 meshes

[loading routine](#)

 movement

[real-time](#)

 textures

[filter implementation](#)

[gargle effect](#)

 GenerateMipSubLevels method

[BaseTexture class](#)

[GenerateSkinnedMesh method, skeletal animation](#)

 geometry

[3D scenes](#)

 GetAttachedSurface method

[DirectDraw](#)

[GetEffects method](#)

[GetSendQueueInformation method](#)

[GLines property](#)

 global variables

[HLSL and](#)

 graphics

[initialization](#)

[animated meshes, loading](#)

[font drawing](#)

 initializing

[games](#)

[meshes](#)

 graphics cards

[vertex processing](#)

[GraphicsSprite class](#)

[GroupCreated event](#)

[GroupDestroyed event](#)

[GroupInformation event](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

hardware

[device availability](#) [2nd](#)

sound

[shared](#)

hierarchy

[meshes](#)

[skeletal animation](#)

[high scores, Dodger game](#)

highlighting

[specular highlighting](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#) [9th](#) [10th](#) [11th](#) [12th](#) [13th](#) [14th](#) [15th](#) [16th](#) [17th](#)

HLSL

[flow control](#)

[global variables](#)

meshes

[drawing](#)

[textured, rendering](#)

[pixel shader and](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#)

[shader programs and](#)

variables

[declaring](#)

[HLSL \(High Level Shader Language\)](#) [2nd](#)

[HostMigrated event](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[IComparer interface](#)

[index buffers 2nd 3rd](#)

[listing](#)

[memory and](#)

[vertices and 2nd 3rd 4th 5th 6th 7th 8th](#)

[indexed meshes](#)

[skeletal animation 2nd 3rd 4th 5th 6th 7th 8th](#)

[IndicateConnect event](#)

[IndicateConnectAborted event](#)

[initialization](#)

[devices](#)

[DirectDraw](#)

[environment mapping](#)

[force feedback](#)

[DirectInput object](#)

[full-screen rendering device](#)

[graphics](#)

[font drawing](#)

[games](#)

[Peer object](#)

[InitializeGraphics function](#)

[device class](#)

[InitializeSound method](#)

[3D sound and](#)

[input.](#) [See [user input](#)]

[interfaces](#)

[IComparer](#)

[intrinsic types](#)

[variable declaration](#)

[IsServiceProviderValid method](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[Y\]](#) [\[W\]](#)

joins

sessions

[event handlers](#)

[joysticks_2nd](#)

[DirectInput initialization](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

keyboard

[DirectInput devices](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#) [7th](#) [8th](#)

keyboards

[Treeview](#)

KeyPress event

[progressive meshes](#)

KeyPress event handler

[simplification meshes](#)

keystrokes

[Dodger game](#)

keywords

[using](#)

[\[Team LiB \]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

launching applications

[lobbies 2nd 3rd 4th 5th](#)

[left-handed coordinate system](#)

[boxes](#)

[cylinders](#)

[polygons](#)

[spheres](#)

[teapot](#)

[toruses](#)

LevelCount property

[BaseTexture class](#)

LevelOfDetail property

[BaseTexture class](#)

[lighting 2nd 3rd 4th 5th](#)

[ambient](#)

[reflecting](#)

[defining lights](#)

[devices](#)

[diffuse](#)

[reflecting](#)

[games](#)

[meshes 2nd 3rd 4th 5th 6th](#)

[realistig](#)

[reflecting](#)

[ambient light](#)

[diffuse light](#)

[specular highlighting 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th 17th](#)

[constants](#)

[textures 2nd 3rd 4th 5th 6th](#)

[vertices and](#)

[lights](#)

[text drawing](#)

[Line class](#)

[line drawing 2nd 3rd 4th 5th 6th](#)

[GILines property](#)

[Line class](#)

[points](#)

[random](#)

[listing](#)

[LineList primitive type](#)

[LineStrip primitive type](#)

[Listener3D object 2nd 3rd 4th](#)

listings

[cube creation](#)

loading

meshes

[animation and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[from file](#)

[normal data and](#)

sound

[static 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

lobbies

applications

[awareness 2nd](#)

[launching applications 2nd 3rd 4th 5th](#)

[\[Team LiB\]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

main method

[skeletal animation](#)

materials

[meshes 2nd 3rd 4th 5th 6th](#)

memory

[index buffers and](#)

Mesh

[defining](#)

[overview 2nd 3rd 4th 5th](#)

[Mesh.FromFile method](#)

[Mesh.FromStream method](#)

MeshContainer class

[overriding](#)

[skeletal animation 2nd](#)

meshes

[adjacency information](#)

[generating](#)

animated

[rendering 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th 17th](#)

[skeletons](#)

[arrays of](#)

[AttributeWeights structure](#)

[cleaning](#)

[complex models 2nd 3rd 4th 5th 6th 7th](#)

[constructors](#)

containers

[rendering 2nd 3rd](#)

creating

[box method](#)

[D3DX](#)

data

[cloning 2nd 3rd 4th](#)

[data optimization 2nd 3rd 4th 5th](#)

drawing

[HLSL and](#)

[drawing with](#)

[encapsulation](#)

flags

[enumeration values](#)

games

[loading routine](#)

generating

[skeletal animation and](#)

[graphics](#)

[hierarchy](#)

[skeletal animation](#)

[lighting 2nd 3rd 4th 5th 6th](#)

loading

[animation and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[normal data and](#)

[with normal data](#)

[loading from file](#)

[materials 2nd 3rd 4th 5th 6th](#)

[normal data](#)

objects

[\[Team LiB\]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

namespaces

[audio and video playback 2nd](#)

[diagnostics](#)

[enumeration and 2nd 3rd 4th 5th 6th 7th](#)

[DirectSound](#)

[Directsound](#)

[buffers](#)

[DirectSound](#)

[Device class](#)

[Lobby \(DirectPlay\)](#)

[Microsoft.DirectX.DirectSound](#)

[networking](#)

[bandwidth](#)

[detection 2nd 3rd 4th 5th 6th](#)

[event model 2nd 3rd 4th 5th 6th](#)

[events](#)

[ApplicationDescriptionChanged](#)

[AsyncOperationComplete](#)

[ClientInformatin](#)

[ConnectComplete](#)

[FindHostQuery](#)

[FindHostResponse](#)

[GroupCreated](#)

[GroupDestroyed](#)

[GroupInformation](#)

[HostMigrated](#)

[IndicateConnect](#)

[IndicateConnectAborted](#)

[PeerInformation](#)

[PlayerAddedToGroup](#)

[PlayerCreated](#)

[PlayerDestroyed](#)

[PlayerRemovedFromGroup](#)

[Receive](#)

[SendComplete](#)

[ServerInformation](#)

[SessionTerminated](#)

[statistics 2nd 3rd 4th 5th 6th](#)

[networking](#) [See also [DirectPay](#)]

NoDirtyUpdate flag

[buffers](#)

[locking](#)

[non-square surfaces](#)

NoOverwrite flag

[dynamic buffers](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

object

[Peer object](#)

objects

[Buffer3D](#)

[properties 2nd 3rd](#)

[BufferDescription](#)

[properties 2nd 3rd 4th 5th](#)

[Listener3D 2nd 3rd 4th](#)

meshes

[stock objects](#)

Peer

[initialization](#)

[SimplificationMesh](#)

[texture 2nd 3rd 4th 5th 6th 7th 8th](#)

[obstacles, Dodger game 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th](#)

[collection class](#)

[constants](#)

[drawing](#)

optimization

[mesh data 2nd 3rd 4th 5th](#)

[Optimize function](#)

overloads

[buffer locking and](#)

[Clone method](#)

[\[Team LiB \]](#)

[\[Team LiB\]](#)

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

parameter

[EffectInstance](#)

parameters

[cleaning meshes and](#)

[extended material parameter](#)

[vertex buffers](#)

[parametric equalizer effect](#)

passwords

[peer-to-peer connections](#)

[patch meshes 2nd 3rd 4th 5th 6th 7th 8th](#)

[Peer object](#)

[creation flags](#)

peer-to-peer connections

[actions in sessions 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[DirectPlay and 2nd 3rd 4th 5th 6th 7th](#)

[passwords](#)

sessions

[connecting to 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[lost 2nd 3rd 4th 5th](#)

[PeerInformation event](#)

[per-pixel specular highlights](#)

performance

[boxing](#)

[event model and 2nd 3rd 4th 5th 6th](#)

[IComparer interface](#)

[methods and](#)

[using statements and](#)

pixel shader

[HLSL and 2nd 3rd 4th 5th 6th 7th](#)

instruction set

[limitations](#)

[pixel shaders](#)

player management

[event handler](#)

[PlayerAddedToGroup event](#)

[PlayerCreated event](#)

[PlayerDestroyed event](#)

[PlayerRemovedFromGroup event](#)

[playing audio 2nd](#)

playing sound

[static 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[playing video 2nd](#)

[PointList primitive type](#)

points

[lines](#)

polygons

[left-handed coordinate system](#)

PreLoad method

[Resource class](#)

primitive types

[LineList 2nd](#)

[PointList](#)

rendering

[listing](#)

[TriangleFan](#)

[\[Team LiB\]](#)

[[Team LiB](#)]

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)]

[ranks parameter](#)

[buffers](#)

[locking](#)

[rasterization](#)

[depth buffers and 2nd 3rd 4th 5th](#)

[reference rasterizer](#)

[ReadOnly flag](#)

[buffers](#)

[locking](#)

[real light](#)

[real-time](#)

[movement](#)

[games](#)

[Receive event](#)

[reference rasterizer](#)

[references](#)

[loading to project](#)

[render targets 2nd 3rd](#)

[rendering](#)

[environment maps 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th](#)

[fixed-function pipeline and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th](#)

[full-screen device 2nd 3rd 4th 5th](#)

[DirectDraw 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th](#)

[meshes](#)

[animated 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th 17th](#)

[containers 2nd 3rd](#)

[programmable pipeline 2nd 3rd 4th](#)

[split meshes](#)

[primitive types](#)

[listing](#)

[shader programs](#)

[techniques 2nd 3rd 4th](#)

[sprites 2nd 3rd 4th 5th 6th 7th](#)

[text](#)

[texture](#)

[to surfaces 2nd 3rd 4th 5th 6th](#)

[vertex buffers and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th](#)

[renderWindow argument](#)

[resets, device](#)

[automatic during resize 2nd](#)

[resize](#)

[device rests](#)

[automatic 2nd](#)

[Resource class 2nd 3rd](#)

[methods](#)

[properties](#)

[resources](#)

[lifetimes](#)

[texture resources 2nd 3rd](#)

[RotationAxis function](#)

[\[Team LiB\]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[SendComplete event](#)

[SendFlags enumeration](#)

[DirectPlay sessions](#)

[ServerInformation event](#)

[servers](#)

[initializing](#)

[dedicated server session setup](#)

[lost events 2nd](#)

[sessions](#)

[connecting to 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[event handlers](#)

[joining](#)

[user detection 2nd 3rd 4th](#)

[sessions, peer-to-peer](#)

[actions in 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[sessions, peer-to-peer connections](#)

[host migration](#)

[lost 2nd 3rd 4th 5th](#)

[SessionTerminated event](#)

[SetEffects method](#)

[SetPosition method](#)

[SetTexture function](#)

[SetTexture method](#)

[SetupCamera function](#)

[Boolean variable and](#)

[Shader Model 2.0 Texture Blend listing](#)

[shader programs](#)

[HLSL and](#)

[rendering](#)

[techniques 2nd 3rd 4th](#)

[shaders](#)

[diffuse lighting shader](#)

[specular highlighting and](#)

[inputs](#)

[outputs](#)

[shiny objects.](#) [See [specular highlighting](#)]

[simplification mesh](#)

[creating](#)

[simplification meshes](#)

[KeyPress event handler](#)

[SimplificationMesh object](#)

[simplifying meshes 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th](#)

[single threading devices](#)

[forcing use](#)

[skeletal animation](#)

[bone offset matrices](#)

[setup](#)

[GenerateSkinnedMesh method](#)

[indexed meshes, bone animation 2nd 3rd 4th 5th 6th 7th 8th](#)

[MeshContainer class](#)

[missing](#)

[variables](#)

[sound](#)

[3D space and 2nd 3rd 4th 5th](#)

[effects and 2nd 3rd 4th 5th](#)

[\[Team LiB\]](#)

[\[Team LiB\]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

target

[render targets 2nd 3rd](#)

TCP/IP

[addresses 2nd](#)

teapot

[left-handed coordinate system](#)

techniques

[shader programs 2nd 3rd 4th](#)

[TransformDiffuse](#)

tessellation levels

[viewing 2nd](#)

tessellation level

[patch meshes](#)

[texel \(texture element\)](#)

text

[drawing 2nd 3rd 4th 5th 6th 7th](#)

[camera](#)

[declaring fonts](#)

[lights](#)

rendered

[progressive meshes](#)

[rendering](#)

texture

constructors

[parameters](#)

[objects 2nd 3rd 4th 5th 6th 7th 8th](#)

[rendering and](#)

[texture element](#) [See texel]

[texture resources 2nd 3rd](#)

textures

[BaseTexture class](#)

blended

[color and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[DirectDraw](#)

filling

[with arrays](#)

[with unsafe code](#)

filters

[games](#)

games

[filter implementation](#)

[lighting 2nd 3rd 4th 5th 6th](#)

[locking 2nd 3rd 4th 5th 6th 7th](#)

meshes

[rendering with HLSL](#)

[mipmaps](#)

[video as 2nd 3rd 4th](#)

[three dimensional objects 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th](#)

timer

[animation and](#)

toruses

[left-handed coordinate system](#)

transformations

[world transform](#)

[TransformDiffuse technique](#)

[\[Team LiB\]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[unboxing](#)

unsafe code

[compiling](#)

updates

[sprites](#)

[animated 2nd](#)

[DirectDraw](#)

URLs (Uniform Resource Locators)

[DirectPlay addresses and](#)

user input

[device detection 2nd 3rd 4th 5th 6th](#)

devices

[Treeview](#)

[DirectInput devices 2nd 3rd 4th 5th 6th 7th 8th](#)

[game pads 2nd](#)

[DirectInput initialization](#)

[joysticks 2nd](#)

[DirectInput initialization](#)

keyboards

[Treeview](#)

mouse

[Treeview](#)

users

detecting

[session join 2nd 3rd 4th](#)

[using keyword](#)

using statements

[performance and](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

[Validate method](#)

validation

[meshes](#)

variables

Boolean

[connection checking](#)

[camera depth position](#)

declaration

[HLSL](#)

global variables

[HLSL and](#)

[intrinsic types](#)

[skeletal animation](#)

[vertex buffers 2nd 3rd](#)

[COM pointers](#)

[parameters](#)

[rendering and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th](#)

[vertex shaders](#)

[VertexFormat property 2nd](#)

vertices

[graphics cards](#)

[index buffers and 2nd 3rd 4th 5th 6th 7th 8th](#)

meshes

[welding 2nd 3rd 4th](#)

processed

[programmable pipeline and](#)

random

[creating](#)

verticies

[primitives and](#)

video

[as textures 2nd 3rd 4th](#)

[playback 2nd](#)

[view frustum](#)

view frustum

[projection transform and](#)

view matrix

[environment maps and](#)

Visual Studio.NET

[loading](#)

projects

[new](#)

[voice chat, DirectPlay 2nd 3rd 4th 5th 6th 7th](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[Y\]](#) [\[W\]](#)

[waves reverberation](#)

[weights](#)

[_meshes](#)

[_welding verticies in meshes_2nd_3rd_4th](#)

[white color values](#)

[_lighting](#)

[_WHQL \(Windows Hardware Quality Labs\)](#)

[Windows Application project](#)

[_new](#)

[_world transform](#)

[\[Team LiB \]](#)