COPYRIGHT

Ultimate Game Programming with DirectX® Second Edition
Allen Sherrod

Publisher and General Manager, Course Technology PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Content Project Manager: Jessica McNavich

Marketing Manager: Jordan Casey

Senior Acquisitions Editor: Emi Smith

Project Editor: Kate Shoup

Technical Reviewer: Wendy Jones

Editorial Services Coordinator: Jen Blaney

Copy Editor: Ruth Saavedra

Interior Layout: Shawn Morningstar

Cover Designer: Mike Tanamachi

CD-ROM Producer: Brandon Penticuff

Indexer: Broccoli Information Services

Proofreader: Kate Shoup

Printed in the United States of America
1 2 3 4 5 6 7 11 10 09

Library of Congress Control Number: 2008929226

ISBN-10: 1-58450-559-1

eISBN-10: 1-58450-620-2

Course Technology
25 Thomson Place
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at **international.cengage.com/region**.

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com**. Visit our corporate Web site at **cengage.com**.

DEDICATION

This book is dedicated to my parents and my younger sister.

All that I do is to make those who came before us proud, especially my grandparents.

ACKNOWLEDGMENTS

Writing and publishing a book takes a lot of effort from many different individuals. Having a book published is a very rewarding experience but one must work long and hard to see the project through to completion. I would like to thank everyone at Course Technology and Cengage Learning, especially Kate and Emi for all of their patience and help throughout this process. I am sure I did not make it easy for them.

I would like to thank my friends Courtney and Marie for all they have taught me and the motivation they have given me to take a risk and become something more than I've ever thought possible. Also, I would like to thank Heather, Shireen, and Nicole for always being there for me when I needed it. Without them it would have been hard to work those late hours trying to complete this project. Thanks for always listening. I would also like to thank Professor Chris Howard for his friendship and for everything he has taught me throughout the years.

ABOUT THE AUTHOR

Allen Sherrod is an experienced writer on the topic of game development with several titles published on video game graphics, engineering, and game programming. A DeVry graduate, Allen has researched game development since he was in high school. Allen is currently developing multiple independent game titles.

ABOUT THIS BOOK

This book is the second edition of *Ultimate Game Programming with DirectX*. In the first edition, the main goal was to teach beginner- and intermediate-level C++ hobby programmers how to make games using DirectX 9.0. Throughout that book readers learned everything from drawing shapes to applying images on surfaces to the creation of their own extremely simple game engine and game project.

This book aims to build on the parts of the first edition that made it so successful while at the same time improving on all areas in which the first edition came up short. What separates this book from other books on DirectX that are targeted toward beginners is that this book is targeted toward the intermediate-level programmer. This book has relevance and importance to anyone coming to it from the first edition or from other DirectX 9 or DirectX 10 books. For readers coming from OpenGL, there is still a lot of knowledge to gain from this book because the prerequisite is that you are familiar with the basics of game and graphics programming and that you have a willingness to learn a new technology, DirectX 10. Complete beginners can still follow this book and learn a lot, but be aware that the pace is faster than your average novice-level book and is intended to go beyond what most beginner-level books offer.

Those coming to this book from another DirectX book can take comfort in noting that this book not only covers DirectX 10 in detail but also topics such as graphical interfaces, animation paths, level loading and rendering, lighting and shadows, and various surface mapping techniques.

By the end of this book you should have a firm understanding of DirectX 10 and each of its subsystems. You will be able to walk away from this book with a true sense of accomplishment and confidence that the knowledge you have gained will be useful to you in the future and that this book actually taught you something of interest, which is always a challenge when it comes to game-development–related books. Of course, this book is only the beginning, because everything dealing with modern game development could not possibly fit in one book. These topics cover everything from advanced computer graphics, to physics and collisions, to advanced artificial intelligence, and more.

The purpose of this book is to surpass what was done in the first edition as well as to surpass what is available on the market at the time of this writing. Game development is a complex and growing field, so the sooner you are able to stand on your feet knowledge-wise, the sooner you can further develop your skills and build the game you've always envisioned.

RECOMMENDED KNOWLEDGE AND BACKGROUND

Creating real-time interactive software is a tough and challenging business. A lot of talent and dedication go into creating even simple video games (compared to modern commercial video games). To create a video game, it is important to have the tools and experience necessary to take a vision and make it a playable product.

This book is targeted toward the intermediate-level programmer. This means it is important to have at least some prior knowledge and experience to get the most out of the text. Although it is not necessary to be a high-level intermediate or advanced programmer, it does help if you already have some knowledge that you can bring to the table. This includes the following.

- Beginner- to intermediate-level C++ skills (e.g., you know what a class is, are comfortable with pointers and references, and so forth)

- Experience debugging applications, which is key to being able to track and fix bugs in the code you write
- High comfort level with the development tool of your choice such as Microsoft's Visual Studio .NET 2008, which is used in this book
- High comfort level with Windows Vista
- High-school–level mathematics, at the least
- A desire to learn and grow by pushing your skills to the limit and beyond

This book assumes that you've never used DirectX 10 before and that you have some level of C++ experience. Although not necessary, to really help solidify your knowledge of 3D game programming, it would be extremely beneficial to readers to have experience in the following:

- Win32 application programming
- Some DirectX 9 or XNA experience
- Some OpenGL experience if you don't have DirectX 9 in your background
- Experience with algebra, calculus, and physics
- Experience with programmable shaders

The goal of this book is to take you from being a newcomer to DirectX 10 or a beginner-level DirectX 10 programmer to being an intermediate DirectX 10 programmer.

CHAPTER BREAKDOWN

This book is composed of 14 chapters and two appendixes. They break down as follows:

- Chapter 1, "Introduction to DirectX 10," offers a general overview and introduction to DirectX, the Windows Vista operating system, and programmable shaders.
- Chapter 2, "Direct3D 10," covers the basics of rendering 3D graphics and text using the Direct3D 10 graphics API. In this chapter you learn how to create a Direct3D window and how to display text to the screen.
- Chapter 3, "Rendering Geometry," covers what primitives are and how to render them, how to implement simple programmable shaders, and how to work with colors.
- Chapter 4, "Shader Model 4.0," is entirely dedicated to the new Shader Model 4.0, which was introduced in Direct3D 10 and is now also included as part of OpenGL. In this chapter you learn about the features, functions, keywords, and other aspects of vertex, pixel, and geometry shaders.
- Chapter 5, "Transformations," teaches you transformations and coordinate spaces. In this chapter you learn about projections, views, and world transformations and how they can be combined to represent a scene.
- Chapter 6, "Shading Surfaces," teaches how to load and render image data known as textures onto a surface in Direct3D 10. Also in this chapter, you learn how to use

multiple textures on a single surface as well as how to render an entire scene to an off-screen surface, a technique known as render-to-texture.

- Chapter 7, "Additional Texture Mapping," builds off of Chapter 6 and expands your knowledge of texturing surfaces to include being able to use alpha transparencies on the pixel level through alpha mapping, sprites, and geometry displacement. You also learn how to render dynamic reflections on the objects in your scene.

- Chapter 8, "Game Math," reviews the common game mathematics that you will encounter when making video games. The main topics covered in this chapter include vectors, matrices, planes, rays, quaternion rotations, and bounding geometry. Also covered in this chapter are 3D virtual cameras—how they are defined, created, and manipulated (i.e., moved and rotated).

- Chapter 9, "Sound in DirectX," covers how to play sound and music in your games using the Direct Sound API and XACT API. In this chapter you learn how to load, play, pause, and stop your sounds as well as how to use both APIs to play 3D audio. 3D audio is audio that is dynamically manipulated based on its 3D properties, which include its position in the virtual world. This chapter also covers how to use the XACT cross-platform tool, how to play audio cues and music, and how to stream music into your gaming applications. XACT is a new API added to DirectX 10 and is also part of Microsoft's XNA framework.

- Chapter 10, "Game Input," covers how to detect and respond to input in your games using DirectX. In this chapter you learn how to detect input from Xbox 360 game controllers using XINPUT, which works on both the PC and the Xbox 360. XINPUT is used for any Xbox 360 controller that includes game pads (for wired and, now, wireless game pads), guitar controllers, steering wheels, and so forth.

- Chapter 11, "3D Models," covers how to load and render complex 3D models to the screen using Direct3D 10. This chapter also covers the creation of 3D geometry using various software packages.

- Chapter 12, "Animations," covers how to perform various animation techniques in Direct3D 10. This includes key-framed animations, skeleton or bone animations, interpolations, and animation paths. Animation paths are used to move 3D objects along predefined paths. This is seen in modern video games in, for example, enemies patrolling an area and in real-time cut-scenes. Also in this chapter is a discussion on how to calculate the frames per second of your application, which can be useful for simple benchmarking tests.

- Chapter 13, "Lighting," covers how to perform real-time lighting in Direct3D 10 using programmable shaders. This chapter covers various types of lights, such as point lights, spot lights, area lights, and directional lights. In addition, this chapter covers how to compute and use different lighting contributions and surface materials on 3D objects. Chapter 13 also includes a discussion on light mapping and shadowing.

- Chapter 14, "Conclusions," discusses what the next recommended steps are in your game-development education. It talks briefly about topics such as scene management, game engines, multi-player games, and advanced game programming.

- Appendix A, "Answers to Chapter Questions," includes the answers to each of the book's chapter questions.

- Appendix B, "Recommended Resources," suggests resources that you can explore to increase your knowledge in various game-development–related areas.

CONTENTS OF THE CD-ROM

On the CD-ROM that accompanies this book you can find all the chapter samples that are discussed. The samples are organized by chapter and have programming projects for Microsoft's Visual Studio Visual Studio .NET 2008. The programming language used throughout this book and with the chapter samples is C++.

ERRATA

I and the men and women of Charles River Media work very hard to ensure that your book is error-free and of the highest quality. Sometimes errors do slip by our radar and make it into the final print, however. If you notice any errors, please feel free to submit your feedback at http://www.UltimateGameProgramming.com or to http://www.charlesriver.com, where you can also look at the current errata. If you don't see your error on the list, please let us know of it as soon as you can. These errors can include typos, mislabeled figures or listings, or anything else that you notice that you feel is not quite right. Your reporting of any errors allows us to make the necessary changes and helps future readers avoid confusion if they come across the same issue in their texts.

1. INTRODUCTION TO DIRECTX 10

In This Chapter

- Overview of Graphics and Game Development
- A Look at Windows Vista
- Microsoft's DirectX Technology
- Programmable Shaders
- About This Book

Game development has come a long way since the early days of video games. Today, modern video games are on par with the movie industry in terms of popularity, cinematic visuals, and sales. With the increased popularity video games are seeing in the hardcore and casual gaming arenas comes higher the demand by companies to find talented individuals who can keep up with the fast growing pace of the gaming industry. This includes all areas that contribute to a video game product, including but not limited to the following.

- Game graphics
- Artificial intelligence
- Scripting
- Engineering
- Physics
- Game design
- Storytelling

- Art

Video games are on par with the movie industry when it comes to sales. In 2007, the preorders for Halo 3 *for the Xbox 360 broke records across the entertainment industry, thus allowing* Halo 3 *to beat* Spider Man 3 *at its launch.*

The purpose of this book is to teach DirectX 10 game programming to beginner-and intermediate-level DirectX 9 and C++ programmers as well as beginner-level DirectX 10 programmers coming to this book from another text. In this book, topics ranging from graphical user interfaces to rendering complex virtual environments to advanced graphical effects using shaders are explored in great detail in a manner that is easy to follow and easy to pick up on and learn.

OVERVIEW OF GRAPHICS AND GAME DEVELOPMENT

Modern video games are often made with large budgets and large development teams composed of very talented and bright individuals. Budgets for some modern video games are in excess of $10 million. The large amount of money involved often times makes it hard for publishers to take a risk on new development studios and new game ideas.

Not all modern video games exceed the $10-million mark, but some do.

Recently the casual games market has been on the rise. Casual games are often far cheaper to create, and some require only a few individuals on a development team. Thanks to the popularity and success of the Nintendo Wii and the Nintendo DS, casual games and the gamers who play those games are starting to take up a large piece of the market share. Today most AAA large-budget games are 3D games, while some, although not all, casual games are 2D.

AAA has various meanings, but it mostly refers to a game with a budget over $10 million.

A casual game is a game that is simple in its mechanics and is easy to pick up and play for fun. Hardcore games are more complex in their mechanics and often require a great deal of skill and practice to master.

2D AND 3D GAMES

2D and 3D video games can be found all over the industry. Today, 3D games are standard on PCs, arcades, and home consoles, and they are also starting to be used more on mobile devices such as hand-held consoles, cell-phones, and PDAs. Although 3D games are the main focus in the professional industry, there is still plenty of opportunity for developers looking to make 2D video games. Nintendo took the idea one step further and created *Super Paper Mario* for the Nintendo Wii, which is a game that combines both 2D and 3D visuals to deliver a unique experience to the gamer.

In this book we look briefly at drawing 2D elements to the screen, while the main focus is on 3D rendering and effects. We use 2D elements for text and graphical user interfaces such as heads-up displays and main menus. Although not covered, it is not a stretch to take the topics and ideas learned in this book and use them to create a 2D video game.

Microsoft's XNA, which is a game development framework and toolset, allows for easy and rapid game development on Windows XP, Windows Vista, and the Xbox 360 gaming console. The creation of a 2D game is fairly straightforward using XNA, and XNA can prove to be a very valuable game development technology.

PARTS OF A GAME

Modern video games are very complex pieces of software engineering. Today many parts of a video game require dedicated professionals who often make careers out of their area of expertise. For example, video game jobs include, but are not limited to, the following areas.

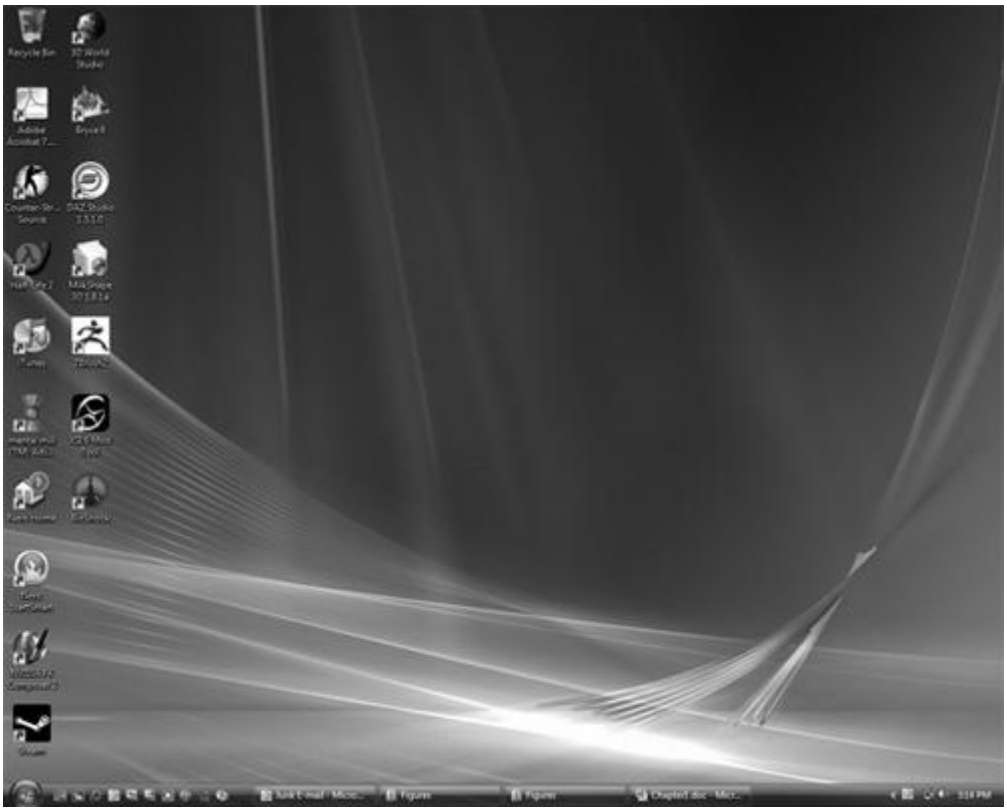
- Graphics programmers
- Engine programmers
- Physics programmers
- Artificial intelligence programmers
- Networking programmers
- Scripting programmers
- Artists
- Animators
- Sound engineers
- Quality assurance specialists
- Game designers
- Writers
- Level designers

The main focus of this book is game graphics, input, and sound with DirectX 10. By the end of this book you will have all the tools necessary to create your own fun 3D video game.

A LOOK AT WINDOWS VISTA

Windows Vista (see Figure 1.1) is the latest operating system from Microsoft Corporation. With Vista, users get more features and options than ever before. The features that affect PC video games include but are not limited to the following.

FIGURE 1.1. THE VISTA DESKTOP.



- The Windows Game Explorer
- Windows Live (i.e., Xbox Live for the PC)
- Flexible parental controls
- Connectivity between Windows Vista/Media Center and Xbox 360 consoles
- DirectX 10

One of the biggest additions that affect gamers and game developers is support for Microsoft's DirectX 10 technology framework. DirectX has been a huge force in game development since its incarnation in the mid 1990s and has been a major piece of technology for Windows-based video games, simulations, and multimedia applications. With DirectX 10, Microsoft looks to reinvent game development, and thus the game industry, with a new and more powerful version of DirectX that has been built from the ground up for Windows Vista. Along with DirectX 10 comes a new class in hardware, often dubbed DirectX 10–class hardware, and a new programmable shader version, Shader Model 4.0.

MICROSOFT'S DIRECTX TECHNOLOGY

DirectX is a set of technologies that was created by Microsoft in the mid 1990s and was released for their Windows 95 operating system. This set of technologies included several application programming interfaces (APIs) that could be used in game development and other multimedia applications developed on their Windows 95 operating system and were able to talk directly to the hardware using a layer known as the Component Object Model (COM).

All versions of DirectX from 1 through 10 have been released, with the exception of DirectX 4, which was dropped since DirectX 5 was also being developed during the same period.

Before DirectX, developers used DOS to program at a very low level directly to the hardware. Prior to the release of Windows 95, Microsoft developed, fairly quickly, the first version of DirectX for multimedia programming. Because Windows 95 was developed with a different model than DOS, the old DOS ways of doing things were being phased out. With Windows 95, it was still possible to run DOS applications, but eventually future Windows operating systems dropped DOS altogether.

Before DirectX and during its release, OpenGL, which today is one of the top graphics APIs along with Direct3D, was primarily used only by high-end work stations, mostly for engineering purposes. Once graphics hardware came along and consumers started to embrace it, both OpenGL and Direct3D evolved to keep up with the trend in the increases in computer power.

DirectX is composed of 2D and 3D graphics, input, sound, and networking functionality. The 2D graphics were once performed using DirectDraw, which has since been combined into Direct3D, which is the 3D graphical component of DirectX. Together the graphics API for DirectX is commonly referred to as just Direct3D. Even though all areas of DirectX are explored in this book, the bulk of the book deals with 3D graphics using Direct3D 10. DirectX is a general set of libraries and utilities that is used for computer graphics, input, sound, and networking functionality.

Input in DirectX is performed using DirectInput, networking was performed with the now obsolete API DirectPlay, and sound is performed using DirectMusic and DirectSound, which are now combined into one. Another API that was once part of DirectX, DirectShow, was used for the manipulation of multimedia files such as videos. In April 2005 DirectShow was removed from DirectX and is now part of the Windows platform software development kit (SDK).

Recently Microsoft has added two new APIs that are included with the DirectX SDK. These new APIs are XINPUT and XACT. XINPUT allows developers to code applications using the Xbox 360 game controllers. These controllers include game pads, steering wheels, guitar controllers, and anything else that can and will be developed as an Xbox 360 controller.

XACT is an audio creation and playback tool that comes complete with a graphical user interface–based tool and API.

DirectInput is being phased out over time in favor of XINPUT, and DirectSound will be replaced by XACT. Both XINPUT and XACT can also be found on Microsoft's XNA framework, which is a new technology based on Microsoft's C# programming language and their Visual Studio toolset for game development on Windows and the Xbox 360. Since DirectInput and DirectSound are not completely phased out of DirectX, they will be discussed later in this book along with XINPUT and XACT. DirectSound is being replaced by a new low-level sound API called XAudio 2 for the XDK (Xbox development kit) and DirectX SDK.

Throughout this book we discuss each of the APIs that make up DirectX. DirectShow and DirectPlay are no longer part of DirectX, so those APIs are not covered in this book.

There is also DirectSetup, which is not an API but is used for the installation of DirectX components on a Windows PC.

DIRECTX 9 VERSUS DIRECTX 10

All versions prior to DirectX 10 were backward compatible with graphics drivers. When DirectX 10 was developed and released, it was no longer backward compatible and was not compatible with operating systems prior to Windows Vista. DirectX 10 does currently include versions of Direct3D 9, so Direct3D 9 applications can still be developed and executed on Windows Vista.

Microsoft's Xbox game console used a version of DirectX 8, while their Xbox 360 used a version of DirectX 9. The original Xbox had the code name DirectXbox since it used DirectX. The Direct was dropped and we now just have Xbox.

Originally, DirectX was planned to end with DirectX 9 after a Vista version of 9 (DirectX 9L) was released. The Direct3D 10 we know today started out as the Windows Graphics Foundation (WGF). WGF was a graphics API that was built from the ground up and was not compatible with prior versions of Direct3D. Today it is known as Direct3D 10 instead of WGF; the name change occurred sometime before the release of Windows Vista and before the completion of what we know now as DirectX 10.

DirectX 9L was given an L at the end of its name because at the time, Vista was known by its code name, which was Longhorn.

There are several differences between DirectX 9 and DirectX 10 aside from DirectX 10 being Windows Vista (and later) only. The major difference is that DirectX 10 has Direct3D 10, while many of the other aspects of Direct X such as DirectInput and DirectSound have not

been updated since DirectX 8. Therefore, it is really Direct3D 10 that is exclusive to Windows Vista since XINPUT and XACT are also available on previous versions of Windows (e.g., Windows XP), and all other areas except Direct3D have not been updated since or before DirectX 8. Each piece of DirectX can be upgraded without affecting the others and the major upgrade in DirectX 10 is the new graphics API Direct3D 10.

One of Direct3D 10's major selling points is that it features a new shading model called Shader Model 4.0. Graphics hardware that supports this new technology is new itself, and the promise of Shader Model 4.0 is hyped quite a bit. This brings us to another difference: DirectX 10 can only use programmable shading technology, while previous versions of DirectX offered what is known as the fixed-function pipeline. Programmable shading technologies and the fixed-function pipeline are discussed later in this chapter. A shader model refers to the version of the shading language that the hardware supports.

INSTALLING THE DIRECTX SDK

It is important to have all the necessary tools ready to begin working in this book. This includes having a development integrated development environment (IDE) such as Visual Studio .NET 2005, which is the tool used for the code projects for this book. The book's sample code and projects can be found on the accompanying CD-ROM.

If you are using Visual Studio .NET 2005, be sure to have the Platform SDK installed. If you have been successfully using .NET 2005 for Win32 development, then the Platform SDK is already installed.

To code DirectX-powered applications, you need the DirectX SDK. This can be downloaded from Microsoft's Web site along with the most recent run-time version of DirectX. Also be sure you have the latest graphics drivers installed for your hardware. To use Direct3D 10 in hardware mode, you also need DirectX 10–compatible graphics hardware, such as NVIDIA's GeForce 8800. You can use Direct3D in reference mode if you have older graphics hardware that does not support Direct3D 10, but this is not recommended because reference mode runs extremely slowly since Direct3D 10 in this state is essentially falling back to a software rendering mode instead of a hardware rendering mode. To install the DirectX SDK you only need to download the installer, run it, and follow the onscreen prompts.

THE DIRECTX SDK SAMPLE BROWSER

The DirectX SDK Sample Browser is one of the tools you receive when you install the DirectX SDK. This tool is full of sample source applications that cover many areas of DirectX as well as detailed documentation and references. These source samples include C++ and C# versions and can prove to be a great starting point when learning DirectX or expanding your knowledge about a particular topic. There are many samples in the SDK Sample Browser, and once you have installed the SDK it is recommended that you look through it and check out what it has to offer, which ranges from beginner topics to advance subjects. A screenshot of the DirectX SDK Sample Browser is shown in Figure 1.2.

FIGURE 1.2. THE DIRECTX SDK SAMPLE BROWSER.

MANAGED DIRECTX

Microsoft has developed DirectX support for managed code. Managed code is code that is compiled into an intermediate language by using a shared unified set of class libraries. In a managed environment, a run-time–aware compiler takes the intermediate code and translates it to native code during the application's execution. During translation, functions such as array bounds checking, garbage collection, type safety, exception handling, and so forth are handled. The languages that are part of Microsoft's managed .NET framework include:

- .NET C++
- .NET C#
- .NET J#
- .NET JScript
- .NET Visual Basic

The DirectX SDK the toolset comes with examples in both unmanaged C++ code and in managed C# code. In this book we focus on the C++ programming language for all code samples. Managed DirectX is also commonly known as MDX. Currently DirectX10 cannot be coded using a managed language.

Managed code does not use the COM layer, which can, according to Microsoft's managed DirectX documentation, improve performance, reduce the volume of code, and increase a developer's productivity.

XNA

Microsoft has developed a new game development framework called XNA, which builds off of the ideas of managed DirectX. XNA is a framework and set of tools used to make creating games for Windows XP, Windows Vista, and the Xbox 360 much easier. XNA uses a managed development environment and the C# programming language. XNA is becoming very popular and is gaining support. Although XNA is a very powerful framework, it is not a replacement for DirectX and is actually a higher-level technology that is built on top of it.

XNA is not an acronym.

PROGRAMMABLE SHADERS

In the late 1990s and early 2000s, programmable graphics hardware started to emerge on the common marketplace. These graphics cards allowed developers to write and execute their own custom code directly on the graphics hardware. This flexibility in control allowed graphics programmers in all industries to push the limits of computer graphics by creating visuals that otherwise did not exist and were neither possible nor reasonable using a fixed-function pipeline.

Before programmable graphics hardware, graphics programmers relied on the fixed-function pipeline, which is a series of algorithms and states (e.g., lighting, materials, and so forth) that is provided with a graphics API to render geometry to the screen. The fixed-function pipeline came down to essentially enabling and disabling states and features as needed by an application. Unfortunately, having a fixed-function pipeline was highly restrictive because it could not be customized.

When programmable graphics hardware hit the scene, it helped change the way graphics programmers create visuals. This was extremely important in the game industry, where graphics are an important aspect of a game. Programmable graphics hardware allowed individual art styles to emerge in games that weren't possible with the fixed-function pipeline. One example can be seen in the game *Team Fortress 2*, where the developers use a cartoon-like art style for the game's graphics.

With programmable hardware, it is possible for developers to create shaders. A shader is a custom-written shading algorithm that can be executed on the graphics hardware. Starting with Direct3D 10, all graphics programming is done using shaders. This means there is no longer a fixed-function pipeline. This is also true with Microsoft's XNA framework.

LOW-LEVEL SHADERS

Low-level shader developers created graphical effects on programmable hardware using a reduced instruction set computer (RISC)-oriented assembly language. Using assembly language gives developers a lot of control over the instructions on a lower level but at a cost of being harder to develop and maintain than a high-level language such as C or C++. However, before high-level shading languages became commonplace, developers had to work with low-level shaders. Each graphics API had a slightly different syntax, which meant developers couldn't just create one effect and use it in different projects regardless of the API. Low-level shaders, when compared to high-level shaders, have the following generally accepted truths:

- Low-level shaders are harder to implement.
- Low-level shaders are harder to maintain.
- Low-level shaders give low-level control over the instructions, thus giving developers more control over performance.
- High-level shaders are faster to develop.
- High-level shaders are easier to read, especially without comments.
- The syntax of a higher-level language is much more developer friendly overall.

Fortunately low-level shaders are a thing of the past in DirectX. In Direct3D 10, Microsoft's High Level Shading Language (HLSL) is the only option available to graphics programmers and developers. Throughout this book we examine HLSL.

HIGH-LEVEL SHADERS

High-level shading languages have a higher level than the assembly used in low-level shaders. These higher-level shading languages are often modeled after the C programming language. High-level shading languages have become a very important tool in computer graphics programming and are generally considered a major step forward from when developers were faced with assembly-based shading languages.

Three main shading languages are used with OpenGL and Direct3D. These are Direct3D's HLSL, OpenGL's GLSL (OpenGL Shading Language), and NVIDIA's cross platform/API Cg (which stands for C for graphics). Cg and HLSL started off as the same language that was developed by both Microsoft and NVIDIA, but they eventually took separate paths.

Three types of shaders can be used to operate on the various pieces of information that compose a virtual scene: vertex, geometry, and pixel shaders. When combined, these shaders together form one effect, known as a shader program. During the rendering of a scene, only one shader type can be active at one time. For example, this means it is not possible to enable two vertex shaders at the same time to operate on the same data. The same goes for geometry and pixel shaders.

VERTEX SHADERS

Vertex shaders are code that is executed on each vertex (i.e., point) of a piece of geometry that is passed to the rendering hardware. The input of a vertex shader comes from the

application itself, whereas the other types of shaders receive their input from the shader that comes before it, excluding uniform and constant variables, which we discuss in more detail later on in this book. Vertex shaders are often used to transform vertex positions using various mathematical matrices such as the model-view project matrix, and they are used to perform calculations that need to be performed once per vertex. Examples of operations that are often done on a per-vertex level include:

- Per-vertex level lighting
- GPU animations
- Vertex displacements
- Calculating values that can be interpolated across the surface in the pixel shader (e.g., texture coordinates, vertex colors, vertex normals)

GEOMETRY SHADERS

Geometry shaders sit between vertex shaders and pixel shaders. Once data has been operated on by the vertex shader, it is passed to the geometry shader, if one exists, since their existence is optional. Geometry shaders can be used to create (actually generate) new geometry and can operate on entire primitives. Geometry shaders can emit zero or more primitives, where emitting more than the incoming primitive generates new geometry and emitting zero primitives discards the original primitive that was passed to the geometry shader. Geometry shaders are a new type of shader that is available in Shader Model 4.0, which is currently supported by the Direct3D 10 and OpenGL 3.0 graphical APIs.

A primitive is a simple piece of geometry. In modern video games this often refers to a three-point (three-vertex) polygon known as a triangle.

PIXEL SHADERS

The last type of shader is the pixel shader, also known as the fragment shader. A pixel shader operates on each shaded pixel displayed on the screen that makes up a piece of geometry. The input for the pixel shader can be either the output from the vertex shader or, if one exists, the output from the geometry shader.

We discuss programmable shaders in much more detail in Chapter 4, "Shader Model 4." We also discuss in more detail the rasterization process and common game mathematics such as vectors and matrices in Chapter 5, "Transformations," and Chapter 8, "Game Math."

SUMMARY

The goal of this book is to learn DirectX 10 in a fun and exciting manner. By the time you are finished reading this book you should have an intermediate level of knowledge of the technology and be ready to proceed to more advanced topics in game and graphics programming. In this chapter we covered:

- What DirectX is

- The history of DirectX
- A brief discussion of Vista
- A brief introduction to shading technology

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What is DirectX? When was DirectX released and for what operating system?

2. What does COM stand for?

3. List at least four APIs that make up DirectX 10.

4. List two APIs that are no longer part of DirectX 10 but were part of previous versions of DirectX.

5. What are XINPUT and XACT? How do they fit into the DirectX technology?

6. Describe what a shader is and why it is so important to graphics programmers.

7. Describe what a vertex shader is. Describe how the vertex shader is used in relation to the geometry and pixel shaders.

8. Describe what a geometry shader is. Describe how the geometry shader is used in relation to the vertex and pixel shaders.

9. Describe what a pixel shader is. Describe how the pixel shader is used in relation to the vertex and geometry shaders.

10. What is the difference between managed and unmanaged code? List at least three programming languages that are supported by the .NET managed environment.

11. What does MDX stand for? What does XNA stand for?

12. What is WGF? What new name did it get?

13. What was the code name for Windows Vista? What was the codename for the original Xbox?

14. List the three high-level shading languages discussed in this chapter.

15. List four of the five features we've discussed for Windows Vista.

16. True or false: The first version of DirectX was released for Windows 3.1.

17. True or false: All versions of DirectX, versions 1 through 10, have been released.

18. True or false: Direct3D 10 uses Shader Model 4.0 for programmable shaders along with a fixed-function pipeline.

19. True or false: Geometry shaders were first introduced in Shader Model 3.0 and are now being used in Direct3D 10 and OpenGL 3.0.

20. True or false: XNA is a high-level framework built from DirectX.

2. DIRECT3D 10

In This Chapter

- Direct3D 10 Basics
- Font and Text
- Additional DirectX Topics

Video games of any nature are heavily based on their visual representations. Through these visuals we are able to interact with the virtual story or world that we are being presented. Using visuals, we are able to manipulate objects based on what we see through various input devices. Couple input with sound, and it is easy for gamers to get immersed in a world outside of their own.

The purpose of this chapter is to discuss the basics of how to use the visual component of DirectX, called Direct3D. Direct3D handles all visual representations of DirectX dealing with 2D and 3D graphics. In this chapter you will learn the following:

- What Direct3D 10 is
- The difference between Direct3D 9 and 10
- Rendering primitives
- Displaying text
- Working with colors
- Using transformations
- The basics of effects

This book is based on the August 2007 DirectX SDK, which you can find on the CD-ROM or on Microsoft's DirectX Web site. On the DirectX Web site you can also find newer versions of the DirectX SDK as they become available, which I recommend that you use.

DIRECT3D 10 BASICS

DirectX Graphics, also commonly known as Direct3D, includes Direct3D 9 for Vista and Direct3D 10. Direct3D 9 for Vista was once known as Direct3D 9L, where the L represented Vista's development codename called Longhorn. Because so many Direct3D 9 applications are on the market and in development, it is important for Microsoft to continue its support in Windows Vista. Direct3D 10, the major advancement in DirectX 10, is a new API that aims to reinvent the way computer graphics are created in real-time applications on Windows-based products.

For readers who are new to DirectX, Direct3D is the tool we developers use to communicate directly with the hardware in a standardize way. In the past, before major graphics APIs became the de facto standard in the games industry, development studios had to write many routines themselves for the devices they wished to support. This extended past graphics and also included audio, input, and so forth. This made it very difficult to develop games because development teams had to develop, test, and debug code for many hardware configurations that were sometimes written in low-level code. If new hardware was released after a game, development teams had the additional burden of either having to patch their games or do without support for that hardware.

Patching of a game has traditionally been done primarily to fix bugs discovered on some configurations, devices, and so on after a game's release. Patching is also used to add content and features to a game.

Standardization makes development of any type of application easier. With a standard API for graphics, for example, it is up to the hardware manufacturers to write compatible drivers that work for a specific piece of hardware. This places the burden of developing, maintaining, and optimizing graphics routines on the manufacturers rather than the game developers, and that is mostly what DirectX is: a standard set of APIs that can be used to communicate with a host of different hardware devices without any additional work on the part of the developer. Of course, this means hardware makers have to supply drivers for specific operating systems and hardware specifications, which today is normal and is commonly done in many types of applications.

Direct3D can also fall back to software emulation if hardware is not found. This is great for testing features that your hardware does not support but it is extremely inefficient for commercial products.

Direct3D was added to the DirectX family of APIs in DirectX 2.0. Before DirectX 8.0 there existed two APIs for graphics: Direct3D and DirectDraw. DirectDraw was a low-level API that was mostly used for 2D game graphics. As Direct3D evolved, so did its abilities to do both 2D and 3D graphics efficiently. Today, starting with DirectX 8.0 and higher, both APIs have been merged into one. Although it is known as DirectX Graphics, it is simply called Direct3D. In DirectX 7.0 a new feature called hardware-accelerated transformations and lighting, or T&L, was added to the graphics of DirectX. Transformations are the algorithms

that transform geometry from its local representation to one that can be displayed in screen pixels, which was handled by the hardware in and after DirectX 7.0, and lighting was done with built-in lighting algorithms that could be enabled to shade the geometry of a scene based on specified lighting and surface properties. These lighting algorithms were based on Blinn and Phong, which are classic diffuse and specular lighting algorithms that are discussed, among others, in Chapter 13, "Lighting."

Before hardware-accelerated T&L, developers had to perform these functions manually using the CPU. Hardware support allowed that work load to be placed in the rendering pipeline of the graphics hardware's fixed-function pipeline, which allowed the CPU to focus on other tasks.

Along with the merger of the DirectX graphics APIs, later DirectX 8.0 gained support for programmable shading languages. Prior to 2000, graphics hardware relied on a set of prebundled algorithms that together were known as the fixed-function pipeline. For example, to use hardware lighting you simply enabled it with a few function calls; to draw an object you simply sent the geometry down the rendering pipeline with a draw call, and transformations, shading, and so on were taken care of for you. The fixed-function pipeline was truly about enabling and disabling features in your application as desired.

Around the end of the last millennium, video games started to grow in complexity at a fast rate. Game developers required more control over what they could do in their applications, and when it came to graphics, this was restricted by the fixed-function pipeline. Using the fixed-function pipeline, developers came up with many tricks and ways around certain limitations, but they were still extremely limited in what they could do. Graphics hardware that could be programmable to the metal, as the saying goes, was the solution to this problem. By allowing graphics programmers to replace the fixed-function pipeline with their own algorithms, developers were freed of the graphical shackles that had restricted them since the dawn of graphics hardware and graphics APIs.

The programmable shading technology evolved along with the game development industry. In the beginning of programmable shading technology, developers used low-level assembly language to program to the metal of the hardware with limited support in terms of hardware capabilities. Today we have complex high-level languages that resemble popular programming languages such as C and C++, with much highly evolved graphics hardware.

Cg was the first OpenGL/Direct3D high-level programmable shading language. Cg was originally developed jointly by Microsoft and NVIDIA, but somewhere along the way Microsoft broke off from the development of Cg and developed HLSL, which explains why the two languages exist and why they are so much alike.

Today, programmable shading languages and the ability to create your own algorithms are at the forefront of computer graphics. In video games specifically they have been of the utmost importance. Programmable technology is so important that Direct3D 10 is the first graphics API to completely remove any support for a fixed-function pipeline. XNA, another of Microsoft's game development technologies that is built on top of DirectX, also takes a programmable shading-only approach.

Working strictly with programmable shading technology is considered a good thing. The mathematics is not difficult to grasp and, in the past, to be a great graphics programmer you had to be at least somewhat familiar with what was going on behind the scenes to be able to accurately dictate in code what you wanted to do. Of course, today graphics programmers need knowledge of computer graphics that goes beyond the API of their choice, but again, this is a good thing for game developers and the games they produce.

Graphics programmers always had to have knowledge of computer graphics beyond API calls, but today they also need to be able to efficiently implement it themselves, whereas before the fixed-function pipeline took care of enough details that almost anyone could create scenes.

SETTING UP DIRECT3D 10

To use Direct3D 10 in an application, you need to set it up in code. This requires a Win32 development environment and the DirectX SDK, both of which should be set up and ready at this point.

Direct3D 10 uses the header files d3d10.h and d3dx10.h, which are installed when you install the DirectX SDK. The first file, d3d10.h, is the main Direct3D header file. The second file, d3dx10.h, contains Direct3D utility-related header information. The Direct3D X utility is very useful and is mostly a set of functions that makes performing certain tasks faster by using one function call. An example of this is loading shading effects with the `D3DX10CreateEffectFromFile()` function, which we discuss in more detail later on in this chapter.

Along with the header files, Direct3D 10 also requires that the application use the Direct3D 10 dynamic link libraries (DLLs). This is done by linking to the libraries d3d10.lib and, if using any of the Direct3D 10 X functions, d3d10x.lib.

Inside the application itself is an object that is created to allow Direct3D to be used. This object is the Direct3D 10 device. The Direct3D 10 device uses the object type `ID3D10Device*` and is a pointer to the device itself. Once created, the device object is used to perform all hardware tasks in Direct3D.

Those experienced with Direct3D 9 and previous versions know that there were two objects that had to be created: the device and an object used to initialize the SDK.

A Direct3D 10 device can be either in hardware or in software rendering mode. As the names imply, hardware mode runs everything on your graphics hardware, while software mode, also known as reference or REF mode, runs everything on the CPU. In Direct3D 9 and earlier versions, the hardware mode was executed through the Hardware Abstraction Layer (HAL), while reference mode was run through the Hardware Emulation Layer (HEL). Direct3D 10 does not have these layers, as it is a new API built from the ground up for Vista. When creating a Direct3D 10 device, it is common to try to create a hardware device first and, if that fails, fall back to software. If the game cannot run in software mode, which none, if any, can, then software mode can be left out or used as a means to display a message to the users informing them that the application requires hardware acceleration.

A function called D3D10CreateDeviceAndSwapChain() is used to create the Direct 3D device. This function creates the device as well as the window's swap chain. We discuss swap chains in more detail in the following section. The function prototype for the D3D10CreateDeviceAndSwapChain() function is as follows:

```
HRESULT D3D10CreateDeviceAndSwapChain(
    IDXGIAdapter *pAdapter,
    D3D10_DRIVER_TYPE DriverType,
    HMODULE Software,
    UINT Flags,
    UINT SDKVersion,
    DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
    IDXGISwapChain **ppSwapChain,
    ID3D10Device **ppDevice
);
```

The D3D10CreateDeviceAndSwapChain() function's first parameter is the address to an adapter object that can be created upon success of this function call. The adapter is used for the creation of subsystem resources such as additional graphics-processing units (GPUs) and video memory. This parameter can be optionally set to NULL.

The second parameter for the D3D10CreateDeviceAndSwapChain() function is the driver type. The driver type specifies if the device will run in hardware or software rendering mode and can be one of the following values, where the last value, D3D10_DRIVER_TYPE_SOFTWARE, is not used but is reserved by Direct3D for possible future use:

```
typedef enum D3D10_DRIVER_TYPE
{
    D3D10_DRIVER_TYPE_HARDWARE = 0,
    D3D10_DRIVER_TYPE_REFERENCE = 1,
    D3D10_DRIVER_TYPE_NULL = 2,
    D3D10_DRIVER_TYPE_SOFTWARE = 3,
} D3D10_DRIVER_TYPE;
```

Don't confuse D3D10_DRIVER_TYPE_REFERENCE *with* D3D10_DRIVER_TYPE_SOFTWARE. *The former is for software rendering (reference), while the latter is reserved.*

The third parameter is a handle to a DLL that implements a software rasterizer. This parameter and must be set to NULL if you are using hardware rendering. You do not need to use this parameter in reference mode unless you are using a specific external DLL that implements the software rasterizer, which is outside the scope of this book.

The fourth parameter is the optional device creation flag. These flags can be used for setting Direct3D in single-threaded mode, debug mode, and so forth. The flags that can be used for this parameter are as follows.

```
typedef enum D3D10_CREATE_DEVICE_FLAG
```

```
{
   D3D10_CREATE_DEVICE_SINGLETHREADED = 0×1,
   D3D10_CREATE_DEVICE_DEBUG = 0×2,
   D3D10_CREATE_DEVICE_SWITCH_TO_REF = 0×4,
   D3D10_CREATE_DEVICE_PREVENT_INTERNAL_THREADING_OPTIMIZATIONS
= 0×8,
} D3D10_CREATE_DEVICE_FLAG;
```

By default, Direct3D is set to multi-threaded mode, but you can set it to single-threaded mode. The debug flag is used to have Direct3D generate additional information that can be useful for the debugging process. The flag used to switch to reference mode tells the `D3D10CreateDeviceAndSwapChain()` function to create two devices, one hardware and one reference, so that the application can freely switch between the two modes. The last flag is reserved by Direct3D.

The fifth parameter of the, `D3D10CreateDeviceAndSwapChain()` function, which should be set to `D3D_SDK_VERSION`, tells Direct3D which DirectX SDK version it is using.

The sixth parameter is a description of how the swap chain is created by the `D3D10CreateDeviceAndSwapChain()` function, while the seventh parameter is the swap chain object that is created upon success of this function. We discuss swap chains in more detail in the next section of this chapter.

The last parameter is the Direct3D 10 rendering device that is created upon this function's success. If the `D3D10CreateDeviceAndSwapChain()` function returns anything other than `S_OK` or if the Direct3D device or swap chain is `NULL`, it means the function failed to set up Direct3D.

If you want to create only a Direct3D 10 device and not a swap chain at the same time, you can use the function `D3D10CreateDevice()`. The function prototype for the `D3D10CreateDevice()` function is as follows.

```
HRESULT D3D10CreateDevice(
   IDXGIAdapter *pAdapter,
   D3D10_DRIVER_TYPE DriverType,
   HMODULE Software,
   UINT Flags,
   UINT SDKVersion,
   ID3D10Device **ppDevice
);
```

The parameters between the `D3D10CreateDevice()` function and the `D3D10CreateDeviceAndSwapChain()` function are identical except for the swap chain parameters.

SWAP CHAINS

Each window that is created can have a scene displayed to it. A swap chain is an object made up of various rendering buffers that is tied to a specific window. You can think of a swap chain as a window, and in order to render to a window in Direct3D 10 you must create a swap chain. In previous versions of Direct3D a swap chain was tied to a window, and a swap chain could be switched between windows (i.e., tied to another window). In Direct3D

10 a swap chain is tied to a device, and a change in the swap chain—that is, if the swap chain is to change the device it is tied to—requires the swap chain to be released and then re-created.

The creation of a swap chain requires a swap chain description of the type DXGI_SWAP_CHAIN_DESC. Swap chains are part of the DirectX Graphics Infrastructure (DXGI), which is discussed at the end of this chapter. A swap chain has the following structure.

```
typedef struct DXGI_SWAP_CHAIN_DESC {
    DXGI_MODE_DESC BufferDesc;
    DXGI_SAMPLE_DESC SampleDesc;
    DXGI_USAGE BufferUsage;
    UINT BufferCount;
    HWND OutputWindow;
    BOOL Windowed;
    DXGI_SWAP_EFFECT SwapEffect;
    UINT Flags;
} DXGI_SWAP_CHAIN_DESC;
```

The first property of the swap chain description is the buffer description. The buffer description is made up of properties that describe the rendering buffer, which includes the buffer's width and height, the window refresh rate (i.e., how many times it is updated), the buffer's internal format, and the method by which each line of pixels that make up the display is rendered (i.e., the order, progressive scan, etc.). A buffer is an array of pixels, in this case, that is drawn to during the rendering. This property of the swap chain description controls how it is created and used. The buffer description has the following structure:

```
typedef struct DXGI_MODE_DESC {
    UINT Width;
    UINT Height;
    DXGI_RATIONAL RefreshRate;
    DXGI_FORMAT Format;
    DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;
    DXGI_MODE_SCALING Scaling;
} DXGI_MODE_DESC, *LPDXGI_MODE_DESC;
```

The second property in the swap chain description is the sample description, which describes how the device handles multi-sampling. Multi-sampling is a graphics-hardware-optimized form of anti-aliasing. We discuss anti-aliasing more in Chapter 7, "Additional Texture Mapping."

The third property of the swap chain description is the DXGI usage flag. This flag specifies CPU usage options and tells Direct3D how the swap chain will be used. For example, a rendering destination in the swap chain can be set to be used as an input into an effect. The valid values for this flag are:

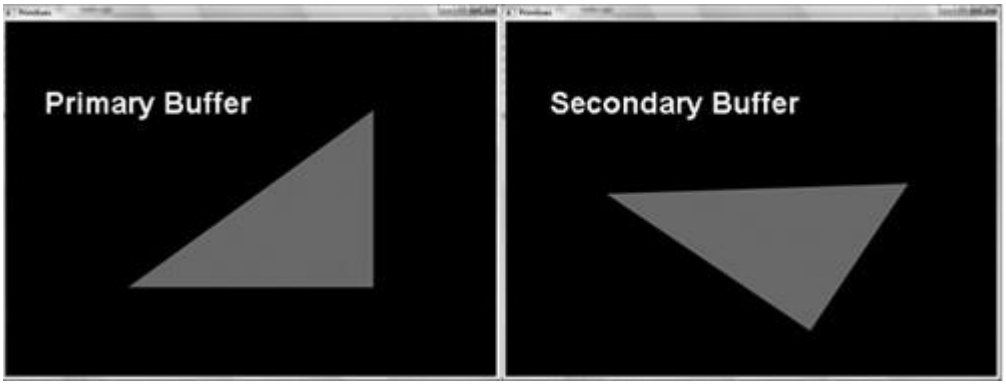
- DXGI_USAGE_SHADER_INPUT, which uses the surface or resource as an input to a shader
- DXGI_USAGE_RENDER_TARGET_OUTPUT, which uses the surface or resource as an output render target

- DXGI_USAGE_BACK_BUFFER, which uses the surface or resource as a rendering back buffer
- DXGI_USAGE_SHARED, which shares the surface or resource
- DXGI_USAGE_READ_ONLY, which specifies the swap chain for read only, no writing

DXGI_USAGE *is implemented as a typedef to an integer.*

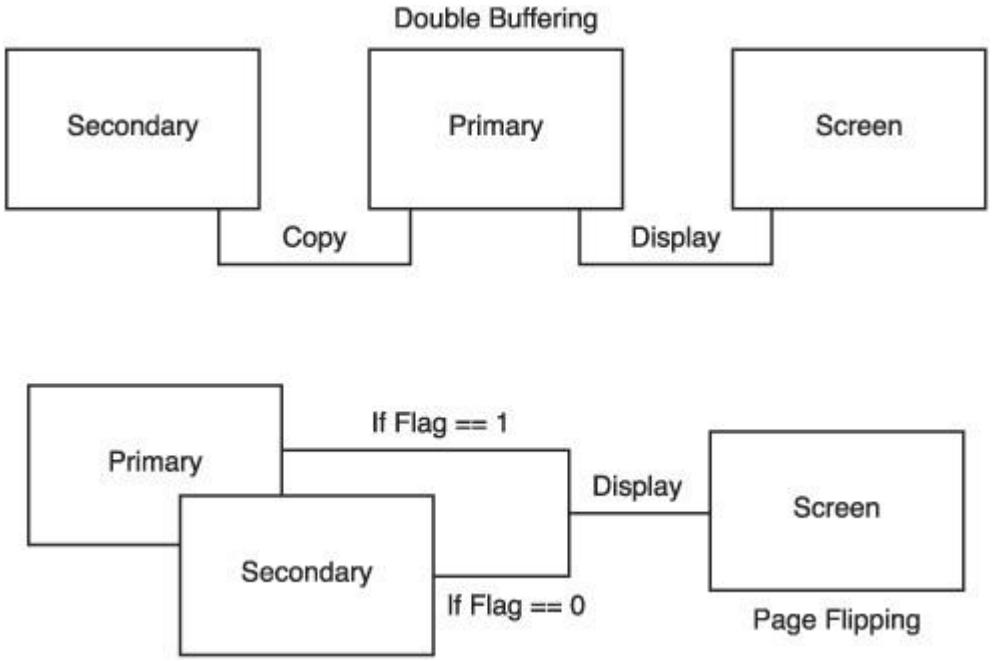
The fourth property of the swap chain description is the number of buffers. To render anything you need at least one buffer. In video games we traditionally use two or more buffers for smooth animations. The problem with using one buffer is that the display can need to be updated before the rendering of the next frame of the scene is complete. If this happens, artifacts can appear, and the scene can be visibly distorted between the half that makes up the last frame and the half that makes up the next frame. By using two rendering buffers, one buffer can be used for display while the other is being operated on. Once the buffer is ready, the new content, which is rendered to the secondary buffer when you have two buffers, is transferred to the primary buffer, which is the buffer that is displayed to the screen. Figure 2.1 shows a visual of buffers.

FIGURE 2.1. RENDERING USING A PRIMARY AND SECONDARY BUFFER.



The technique of copying the contents from an off-screen secondary back buffer to the primary buffer is known as double buffering. Copying large amounts of data can be expensive, so another technique is used, called page flipping. Page flipping is a technique where you don't copy data from one buffer to another, but you simply switch between the buffers so that the one being rendered to is automatically considered the secondary buffer, and the one that is not being rendered to is considered the primary buffer. A visual of this is shown in Figure 2.2. Direct3D handles the page flipping of buffers internally, which we see how to do later in this chapter.

FIGURE 2.2. DOUBLE BUFFERING VERSUS PAGE FLIPPING.



The key ideas to take from this are that you must have one buffer, or the creation of the device will fail. If you have more than one buffer (two is the standard), they allow smooth renderings of multiple frames to occur without artifacts. (A frame is a single rendered image.) Adding more buffers consumes more video memory, so do so keeping that in mind.

The fifth property of the swap chain description is the window handle. All Win32 windows create a window handle when the window is first created. This property takes that handle and is used to tie the swap chain to the window.

The sixth property of the swap chain description is a Boolean flag for whether Direct3D should enter full-screen or windowed mode. If this flag is true, the window is not full-screen; if it is false, the window is full-screen. To support full-screen rendering, which is a window that takes up the entire display, this flag simply needs to be set to false. The resolution of the buffer description determines the full-screen window's display resolution. If the application is using a different resolution than the desktop, the resolution is changed while the application is running. The resolution is returned once the application exits, assuming the resolution that it attempted to change to is supported by the hardware.

The seventh property of the swap chain description tells Direct3D how the contents behave once they are displayed to the screen. The values can be any one of these enumerations.

```
typedef enum DXGI_SWAP_EFFECT
{
    DXGI_SWAP_EFFECT_DISCARD = 0,
    DXGI_SWAP_EFFECT_SEQUENTIAL = 1,
} DXGI_SWAP_EFFECT, *LPDXGI_SWAP_EFFECT;
```

If the discard flag is used, Direct3D will determine the best way to present the scene for the swap chain. If the sequential flag is used, the swap chain cannot be used with multi-sampling.

The eighth and last property of the swap chain description is the swap chain flag. This is used to turn off automatic image rotations and to allow full-screen mode switches in the

application. Automatic image rotation deals with full-screen applications. When rendering the contents of a buffer, the buffer might need to be rotated and changed to match that of the monitor. This is automatically done, but this property can be turned off in Direct3D. The other flag for full-screen mode allows the application to try to match the closest resolution to what the window was before the change in window mode. Without this flag the resolution of the desktop is used if the application is changed to full-screen mode.

Once you have a swap chain description you can call `D3D10CreateDeviceAndSwapChain()` to create the Direct3D device and the swap chain. A swap chain has the type `IDXGISwapChain`. The next step is to create a render target view, which is discussed in the next section.

RENDERING TARGET VIEWS

Later in this book we discuss how to create rendering targets that allow us to render the contents of a scene to an image that can then be used on top of any surface in the virtual scene. This practice is known as off-screen rendering, and it can be used to create effects such as mirrors, TV monitors, and many more interesting things in a rendered scene.

Although we do not cover off-screen rendering in this chapter, there is information related to it that is important to being able to render anything in Direct3D, and this topic is called render target views. Once a swap chain is created, we must create a render target view out of the swap chain's back buffer, which is the buffer that is not being displayed but that is being rendered to. To do this we first get the swap chain's buffer with a call to the swap chain object's member function `GetBuffer()`, which has the following function prototype.

```
HRESULT GetBuffer(
   UINT Buffer,
   REFIID riid,
   void **ppSurface
);
```

The `GetBuffer()` function takes as parameters the buffer index, the ID, and a pointer to the address that points to the buffer. Since we are rendering to a texture, which is another word for image, we would use the following for the ID.

```
_uuidof(ID3D10Texture2D)
```

The `_uuidof()` function is used to get the ID of any structure, which for a 2D image texture is `ID3D10Texture2D` (discussed in more detail in Chapter 6, "Shading and Surfaces"). The ID allows the function to know what type of resource is being fetched.

Once the buffer is obtained, a call to `CreateRenderTargetView()` is made to create the render target out of the swap chain buffer's destination image. The function takes as its first parameter the resource that acts as the rendering target destination, which must have been created using the flag `D3D10_BIND_RENDER_TARGET`. The second parameter is the render target description, and the last parameter is a pointer to the created render target's address. The `CreateRenderTargetView()` function has the following function prototype.

```
HRESULT CreateRenderTargetView(
   ID3D10Resource *pResource,
   const D3D10_RENDER_TARGET_VIEW_DESC *pDesc,
```

```
    ID3D10RenderTargetView **ppRTView
);
```

The render target description can be set to `NULL` to access the entire resource. If you were to fill out the description, it would have the following structure.

```
typedef struct D3D10_RENDER_TARGET_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D10_RTV_DIMENSION ViewDimension;
    union {
        D3D10_BUFFER_RTV Buffer;
        D3D10_TEX1D_RTV Texture1D;
        D3D10_TEX1D_ARRAY_RTV Texture1DArray;
        D3D10_TEX2D_RTV Texture2D;
        D3D10_TEX2D_ARRAY_RTV Texture2DArray;
        D3D10_TEX2DMS_RTV Texture2DMS;
        D3D10_TEX2DMS_ARRAY_RTV Texture2DMSArray;
        D3D10_TEX3D_RTV Texture3D;
    };
} D3D10_RENDER_TARGET_VIEW_DESC;
```

The description for render target views deals a lot with textures, so a discussion of this structure occurs in Chapter 6. Once a render target is created, it can be set as the current render target for the display at any time using the function `OMSetRenderTargets()` of the Direct3D 10 device. The first parameter of the function takes the number of views that will be set, an array of one or more views to set, and an array of one or more depth or stencil views to set for each view. The function prototype for the `OMSetRenderTargets()` function is as follows.

```
void OMSetRenderTargets(
    UINT NumViews,
    ID3D10RenderTargetView *const *ppRenderTargetViews,
    ID3D10DepthStencilView *pDepthStencilView
);
```

Once a render target view is created and set, the application is ready to begin rendering. To review, the steps used to create a Direct3D 10 device are:

1. Create the device.

2. Create the swap chain, which can be done during the creation of the device with the function `D3D10CreateDeviceAndSwapChain()`.

3. Get the swap chain's back buffer.

4. Create a render target view from the swap chain's buffer.

5. Set the render target view when you are ready to start rendering graphics.

By creating a render target that points to the swap chain, which is essentially the window, you can set the rendering of different windows at will. You can also set the renderings to occur to surfaces that are not a swap chain or tied to the window, which is what off-screen rendering (discussed in Chapter 6) is. It is important to keep in mind that render targets tell Direct3D where to draw. If each of the steps mentioned above is successful, you are ready to start rendering 3D scenes.

To avoid memory leaks, be sure every object that is created, such as the Direct3D 10 device and any render targets, is freed at the end of the application or whenever you no longer need it. This can be done by calling the `Release()` member function of the objects. For the Direct3D device object you must first call `ClearState()` on the device before calling `Release()`, which returns the device to the state it was at when it was created.

CLEARING AND DISPLAYING SCREENS

Once a scene has been rendered, you can display it to the window by calling the swap chain's `Present()` function. `Present()` has always been the function in Direct3D used to display a scene once you are finished rendering, and it takes as parameters the sync interval and the presentation flags. The sync interval can be 0, which tells Direct3D to display immediately, or it can be 1, 2, 3, or 4, which tell Direct3D to display after the *n*th vertical blank. A vertical blank is when the monitor blanks for each line of vertical pixels that make up the display.

The flags for the `Present()` function can be 0, which presents the contents of each buffer, `DXGI_PRESENT_DO_NOT_SEQUENCE`, which presents a frame from the current buffer to the output, `DXGI_PRESENT_RESTART`, which presents a frame from each buffer, starting with the first buffer, to the output, and `DXGI_PRESENT_TEST`, which does not display the results but is used to test the swap chain when switching from an idle state. During the development of your applications, you'll find yourself often using 0 for both of the parameters for the `Present()` function.

Before you display a scene, you'll want to render something first. The most basic rendering that can be performed is the clearing of the screen to a specified color, which can be any color you want. You can clear a scene by calling the `ClearRenderTargetView()` function of the Direct3D 10 device, which has the following function prototype.

```
void ClearRenderTargetView(
   ID3D10RenderTargetView *pRenderTargetView,
   const FLOAT ColorRGBA[4]
);
```

The `ClearRenderTargetView()` function takes as parameters the render target view object, which for the clearing of the screen should be the swap chain's render target view that was created, and a color. The color is filled across the entire surface and is a red, green, blue, alpha color value. We discuss colors in more detail later on in this chapter. An example of clearing and the display of a scene using a Direct3D 10 device object is shown as follows, where the scene is set to all black (specified by the color 0, 0, 0, 0).

```
ID3D10Device *g_d3dDevice = … Create Device …;
IDXGISwapChain *g_swapChain = … Create Swap Chain …;
ID3D10RenderTargetView *g_renderTargetView = … Create RT View …;

…
float col[4] = { 0, 0, 0, 0 };
```

```
… START DRAWING …

g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

g_swapChain->Present(0, 0);
```

In the following sections we use the information discussed so far to create 3D applications using Direct3D 10. These applications are simple demos that consist of a demo used to render a blank window and a demo used to display text. This chapter serves mostly as an introduction to setting up and beginning to use the Direct3D API.

BLANK WINDOW DEMO

The first demo implemented in this chapter is the Blank Window demo, which can be found on the CD-ROM in the Chapter 2 folder. The Blank Window demo creates a Direct3D application that displays an empty window. This demo application uses each of the Direct3D functions mentioned up to this point in this chapter.

The demo is made up of a single C++ source file called main.cpp. The main source file starts by including the Windows and Direct3D header files followed by the Direct3D library files. Direct3D uses d3d10.lib and, if using the Direct3D utility, d3dx10.lib. The header files used include windows.h, d3d10.h, and d3dx10.h. The headers and libraries from the global section of the Blank Window demo application are shown in Listing 2.1.

LISTING 2.1. THE GLOBAL SECTION OF THE BLANK WINDOW DEMO

```
#include<windows.h>
#include<d3d10.h>
#include<d3dx10.h>

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")
```

The Blank Window demo only displays a blank window. Since the demo is limited in its application, the only objects needed are the Direct3D device, a swap chain, and a render target view so that Direct3D can be told to render to the swap chain (window) as the destination. Each of these objects are discussed in this chapter, and they are shown in Listing 2.2 for the Blank Window demo. Notice how the demo uses pointers, which are freed later on, as will be seen in an upcoming section.

LISTING 2.2. THE GLOBAL OBJECTS FROM THE BLANK WINDOW DEMO

```
#define WINDOW_NAME        "Blank Window"
#define WINDOW_WIDTH       800
#define WINDOW_HEIGHT      600

ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;
```

The first function from the Blank Window demo that will be discussed is the demo's `InitializeD3D10()` function. This function's purpose is to initialize the Direct3D

rendering device, swap chain, and render target view. As a parameter, it takes a window handle that is created in the main function of the demo later on in this section.

The device initialization function starts by creating a swap chain description based on the type of device to be created. The swap chain is created at a resolution of 800 × 600, with a buffer format of RGBA8 and two buffers (primary and secondary). Once the swap chain description is created, the function uses a loop to try to create the Direct3D device. In the first attempt the loop tries to create a hardware device. If that fails, it tries to create a software reference device. If this fails, Direct3D could not be initialized.

If the device is created successfully, the final steps, mentioned in detail earlier in this chapter, are to create the render target view from the swap chain's rendering buffer and to set that target as the current rendering destination. The device initialization function ends by calling another function, which we discuss later, that sets the rendering viewport. The entire `InitializeD3D10()` function from the Blank Window demo is shown in Listing 2.3.

LISTING 2.3. THE DIRECT3D DEVICE INITIALIZATION FUNCTION FROM THE BLANK WINDOW DEMO

```
bool InitializeD3D10(HWND hwnd)
{
   DXGI_SWAP_CHAIN_DESC swapDesc;
   ZeroMemory(&swapDesc, sizeof(swapDesc));

   // A swap chain needs to be created first. Once created
   // we can create a rendering target that can allow us to
   // actually draw to the swap chain (the window).

   swapDesc.BufferCount = 2;
   swapDesc.BufferDesc.Width = WINDOW_WIDTH;
   swapDesc.BufferDesc.Height = WINDOW_HEIGHT;
   swapDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
   swapDesc.BufferDesc.RefreshRate.Numerator = 60;
   swapDesc.BufferDesc.RefreshRate.Denominator = 1;
   swapDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
   swapDesc.OutputWindow = hwnd;
   swapDesc.SampleDesc.Count = 1;
   swapDesc.SampleDesc.Quality = 0;
   swapDesc.Windowed = TRUE;


   HRESULT hr = S_OK;
   unsigned int flags = 0;

   // This next flag gives us debug information
   // during development.   Not used in release versions.

#ifdef _DEBUG
   flags |= D3D10_CREATE_DEVICE_DEBUG;
#endif
   D3D10_DRIVER_TYPE driverType = D3D10_DRIVER_TYPE_NULL;

   D3D10_DRIVER_TYPE driverTypes[] =
```

```
   {
      D3D10_DRIVER_TYPE_HARDWARE,
      D3D10_DRIVER_TYPE_REFERENCE,
   };

   // Loop through each device type and see if we can create
   // at least one of them.  If they all fail then there is
   // a huge problem with your hardware or DirectX runtime
   // installation.

   unsigned int numDriverTypes = sizeof(driverTypes) /
                                 sizeof(driverTypes[0]);

   for(unsigned int i = 0; i < numDriverTypes; i++)
   {
      driverType = driverTypes[i];

      hr = D3D10CreateDeviceAndSwapChain(NULL, driverType, NULL,
                                         flags,
D3D10_SDK_VERSION,
                                         &swapDesc,
&g_swapChain,
                                         &g_d3dDevice);
      if(SUCCEEDED(hr))
         break;
   }

   if(FAILED(hr))
      return false;


   // Get the back buffer from the rendering swap chain so we
   // can create a destination rendering target from it.
   ID3D10Texture2D *buffer = NULL;
   hr = g_swapChain->GetBuffer(0, __uuidof(ID3D10Texture2D),
                               (LPVOID*)&buffer);

   // Check for problems creating the previous object.

   if(FAILED(hr))
      return false;

   // Create the default render destination for the drawing
calls.

   hr = g_d3dDevice->CreateRenderTargetView(buffer, NULL,

&g_renderTargetView);

   // No longer need this.

   buffer->Release();
```

```
   // Check for problems creating the previous object.

   if(FAILED(hr))
      return false;

   // Set default render target. Can be set during the rendering
   // but since there is only 1 render target it can be set
once.

   g_d3dDevice->OMSetRenderTargets(1, &g_renderTargetView,
NULL);

   // Ensure window is ready to be drawn to.

   ResizeD3D10Window(WINDOW_WIDTH, WINDOW_HEIGHT);

   return true;
}
```

The next functions in the Blank Window demo are `ResizeD3D10Window()`, `InitializeDemo()`, and `Update()`. The `ResizeD3D10Window()` function is used to set the view port and any view-related information that affects the size of a window. Whenever a window is resized, this function is called to tell Direct3D about the change. A viewport specifies the area of the rendering target that will be rendered to. In this demo this starts at the beginning of the window, which is the upper-left corner of the screen, and renders to the entire surface, which goes until the lower-right of the screen. Viewports are set by creating a `D3D10_VIEWPORT` object and passing it to the Direct3D device object's function `RSSetViewports()`, which takes as parameters the number of viewports that are being passed to the function and an array of one or more `D3D10_VIEWPORT` objects. In a geometry shader we can choose which viewport we want to use by specifying the array index if more than one viewport is set.

Later on we create a split-screen, multi-view scene using "player 1" and "player 2" style cameras. This can be done in part by using viewports. For the top player you can have a viewport that renders only to the top half of the window (instead of the entire window), while the second view displays to the bottom half. Using viewports and rendering the scene once for each player's camera, you can create a split-screen game.

The demo-specific initialize and updating functions are empty in this demo since its purpose is to render a completely blank window, which does not need any special loading or updating functionality. Future demos use these functions, so for now they act as a placeholder. When coding along with future demos, you can use the Blank Window demo as a template to fill out the specifics of whatever it is you are trying to create. Everything in this demo appears in all future demos, so there is no need to rewrite what you've already written for each demo. The purpose of the demo initialize function is to load demo-specific assets and resources, such as models, images, effects, and environments. The purpose of the updating function is to perform update operations before rendering each frame, such as

animation updates and artificial intelligence. The resizing, demo-initialize, and update functions are shown in Listing 2.4.

LISTING 2.4. THE RESIZING, DEMO-SPECIFIC INITIALIZATION, AND UPDATING FUNCTIONS

```
void ResizeD3D10Window(int width, int height)
{
   if(g_d3dDevice == NULL)
      return;
   D3D10_VIEWPORT vp;

   vp.Width = width;
   vp.Height = height;
   vp.MinDepth = 0.0f;
   vp.MaxDepth = 1.0f;
   vp.TopLeftX = 0;
   vp.TopLeftY = 0;

   g_d3dDevice->RSSetViewports(1, &vp);
}

bool InitializeDemo()
{
   // Nothing to initialize.

   return true;
}

void Update()
{
   // Nothing to update.
}
```

The next two functions from the Blank Window demo are the `RenderScene()` and `Shutdown()` functions. The rendering function, `RenderScene()`, clears the rendering target to a solid bright red color and displays that result to the screen. The color (1, 0, 0, 1) specifies 100% for the red channel, 0% for the green and blue channels, and 100% for the alpha channel. Since alpha is not used in this demo, using 0 for the alpha channel will have no effect. Alpha, along with colors in general, is discussed in more detail in Chapter 3, "Rendering Geometry." After the render target is cleared, which is set to the swap chain's window as the destination, the rendered image is composed of a single color across the surface. Normally, more rendering would take place with models, levels, and so on, but for this demo `Present()` is called after the clearing to display the solid color.

The shutdown function `Shutdown()` is used to release all allocated objects that were used in the demo. This includes the Direct3D device, the window's swap chain, and the render target's view. The rendering and shutdown functions are shown in Listing 2.5.

LISTING 2.5. THE BLANK WINDOW'S RENDERING AND SHUTDOWN FUNCTIONS

```
void RenderScene()
{
```

```
    float col[4] = { 1, 0, 0, 1 };

    // Clear the rendering destination to a specified color.

    g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

    // Display the results to the target window (swap chain).

    g_swapChain->Present(0, 0);
}

void Shutdown()
{
    // Release all used memory.

    if(g_d3dDevice) g_d3dDevice->ClearState();
    if(g_swapChain) g_swapChain->Release();
    if(g_renderTargetView) g_renderTargetView->Release();

    if(g_d3dDevice) g_d3dDevice->Release();
}
```

The last two functions are the main function and the windows callback function. In this book it is assumed that you are familiar with Win32 programming. In this section we briefly discuss the Win32 code from this demo before moving on. The first function we look at is the window's callback function. Win32 applications can specify a callback function that is used to respond to operating system messages. Each callback function that is defined must have a specific function prototype, which can be seen in the callback function's implementation in Listing 2.6. In this demo, and most demos in this book, the callback function responds to close and destroy messages that are used to close the application when the X button is clicked (or a force quit occurs) on the window, to sizing messages that are used to inform the application that the window was resized, and, for now, to key-down messages that are used to detect keyboard input. In the chapters dealing with input, we replace the key-down message handler with DirectX code in the `Update()` function. For now, the key-down handler is used to detect when the Esc key is pressed so that it can force the application to shut down. The window's callback function for the Blank Window demo is shown in Listing 2.6.

LISTING 2.6. THE BLANK WINDOW'S CALLBACK FUNCTION

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT m, WPARAM wp, LPARAM
lp)
{
    // Window width and height.
    int width, height;

    switch(m)
    {
        case WM_CLOSE:
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
            break;
```

```
      case WM_SIZE:
         height = HIWORD(lp);
         width = LOWORD(lp);

         if(height == 0)
            height = 1;
         ResizeD3D10Window(width, height);
         return 0;
         break;

      case WM_KEYDOWN:
         switch(wp)
         {
            case VK_ESCAPE:
               PostQuitMessage(0);
               break;

            default:
               break;
         }
         break;


      default:
         break;
}

   // Pass remaining messages to default handler.
   return (DefWindowProc(hwnd, m, wp, lp));
}
```

The last function from the demo application is the `WinMain()` function, which is the Win32 main function. The main function creates the window class and the window itself. The window handle that is created is passed to the `InitializeD3D10()` function discussed earlier in this chapter. If the Direct3D device is created, the demo initialization function is called. If that function passes, the application enters the message loop, where it updates and renders the scene during each pass. Outside of the message loop is the shutdown function, which is called during the final stages of the application's execution. The main function for the Blank Window demo application is shown in Listing 2.7. Figure 2.3 shows a screenshot of the demo application.

LISTING 2.7. THE BLANK WINDOW'S MAIN FUNCTION

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prev,
                   LPSTR cmd, int show)
{
   MSG msg;

   // Describes a window.
   WNDCLASSEX windowClass;
   memset(&windowClass, 0, sizeof(WNDCLASSEX));
   windowClass.cbSize = sizeof(WNDCLASSEX);
   windowClass.style = CS_HREDRAW | CS_VREDRAW;
```

```
    windowClass.lpfnWndProc = WndProc;
    windowClass.hInstance = hInstance;
    windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
    windowClass.lpszClassName = "DX10CLASS";
    windowClass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&windowClass))
        return 0;

    // Create the window.
    HWND hwnd = CreateWindowEx(NULL, "DX10CLASS", WINDOW_NAME,
        WS_OVERLAPPEDWINDOW | WS_VISIBLE | WS_SYSMENU |
        WS_CLIPCHILDREN | WS_CLIPSIBLINGS, 100, 100,
        WINDOW_WIDTH, WINDOW_HEIGHT, 0, 0, hInstance, NULL);

    if(!hwnd)
        return 0;

    ShowWindow(hwnd, SW_SHOW);
    UpdateWindow(hwnd);
    // If initialize fail don't run the program.
    if(InitializeD3D10(hwnd) == true)
    {
        if(InitializeDemo() == true)
        {
            // This is the messsage loop.
            while(1)
            {
                if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
                {
                    // If a quit message then break;
                    if(msg.message == WM_QUIT) break;
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
                else
                {
                    Update();
                    RenderScene();
                }
            }
        }
    }

    // Release all resources and unregister class.
    Shutdown();
    UnregisterClass("DX10CLASS", windowClass.hInstance);

    return (int)msg.wParam;
}
```

FIGURE 2.3. A SCREENSHOT FROM THE BLANK WINDOW DEMO.

FONT AND TEXT

In-game text is one of the most common objects that is rendered in video games. Before moving on to geometry, the first topic to discuss is the rendering of text. Text can be used to give video game players textual feedback that can be used for a host of different purposes including:

- Game play information
- Player information
- Titles
- Menus and menu controls (e.g., buttons, text boxes, etc.)
- Player and game statistics
- Timers
- Character dialog

A visual example of what text looks like as part of a larger interface is shown in Figure 2.4. In Direct3D there are two ways you can display text to the screen. You can opt to manually create and render your text by using textured 2D (or even 3D) geometry. This is often an optimal way of rendering text, but the topic of textures does not come until later on in this book, so an implementation of a custom text system could not be covered in this chapter without jumping quickly in many different topics.

FIGURE 2.4. AN EXAMPLE OF TEXT.

The second option is to use Direct3D to render text to the screen. This is the easiest method because Direct3D function calls can be used to perform the work of rendering text. In this chapter we see how to use Direct3D to display 2D text to the application's window.

TEXT DEMO

On the book's accompanying CD-ROM is a demo application called Text in the Chapter 2 folder. The Text demo uses Direct3D to display text to the screen. Text is created and rendered using a Direct3D object of the type `ID3DX10Font`. The `ID3DX10Font` object, which is a global variable in the Text demo, can be used for all text drawing operations. The global section from the Text demo is shown in Listing 2.8. If you are coding along with this section, you can use the Blank Window demo's main source file as a template source file to work with when implementing Direct3D text.

LISTING 2.8. THE GLOBAL SECTION FROM THE TEXT DEMO

```
#include<windows.h>
#include<d3d10.h>
#include<d3dx10.h>

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Direct3D 10 Text"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600

// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
```

```
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

// Renderable font for text display.
ID3DX10Font *g_font = NULL;
```

The two functions we need to examine are the demo's initialization and rendering functions. Every other function that was seen in the Blank Window demo is identical to the Text demo and goes mostly unchanged throughout the book. The `InitializeDemo()` function is used to create the Direct3D 10 font object, while the rendering function uses it to draw text. The Direct3D 10 font is created by calling the Direct3D utility function `D3DX10CreateFont()`. The `D3DX10CreateFont()` function returns a flag that determines its success or failure and has the following function prototype.

```
HRESULT D3DX10CreateFont(
    ID3D10Device *pDevice,
    UINT Height,
    UINT Width,
    UINT Weight,
    UINT MipLevels,
    BOOL Italic,
    UINT CharSet,
    UINT OutputPrecision,
    UINT Quality,
    UINT PitchAndFamily,
    LPCTSTR pFaceName,
    LPD3DX10FONT *ppFont
);
```

The `D3DX10CreateFont()` function takes as its first parameter the Direct3D rendering device. The second and third parameters are the font's size in logical units. The fourth parameter is the weight, which controls the boldness of the font. The fifth parameter controls the number of mip map levels the font should have. Mip maps are discussed in detail in Chapter 6. The sixth parameter is a flag for indicating whether italics are to be used with the font. The seventh parameter is the character set, which can be `ANSI_CHARSET` if you are using ANSI strings, or you can use Unicode strings. The eighth parameter is the output precision, which controls how Windows decides how to match desired font sizes with the actual fonts. Using `OUT_TT_ONLY_PRECIS` for the eighth parameter gives you a TrueType font. The ninth and tenth parameters are used for matching the font's desired quality with the font's default quality and to set the font's pitch and family indexes. The last two parameters are used to specify the name of a font that is installed on your system and the output address for the font object to be created by this function.

Once the font has been created, it can be rendered to the screen. Rendering text to the screen can be done by using the font's member function `DrawText()`. The function `DrawText()` is actually a typedef for `DrawTextA()`, which uses an ANSI character set, opposed to `DrawTextW()`, which uses a Unicode character set. The function prototypes for both text drawing functions are the same, with the exception of the string parameter's data type. The function prototype for the `DrawText()` function is as follows.

```
INT DrawText(
    LPD3DX10SPRITE pSprite,
    LPCTSTR pString,
```

```
    INT Count,
    LPRECT pRect,
    UINT Format,
    D3DXCOLOR Color
);
```

The `DrawText()` function takes as its first parameter a Direct3D 10 sprite object that contains the string to be drawn, which is optional and can be `NULL`, and can be used to speed up the rendering of text if that text is to be rendered more than once. The second parameter is the string that is to be displayed to the screen. The third parameter is the number of characters in the string. The fourth parameter is a rectangle area that specifies the region in which the text can be drawn, which essentially specifies the starting location for the text and how far right and downward the text can be drawn. Text that does not fit in this rectangle is cropped. The fifth parameter is the format in which the text should be displayed, which can use any of the following flags.

- `DT_BOTTOM` justifies the text to the bottom of the rectangle and must be combined with `DT_SINGLELINE`.
- `DT_CALCRECT` has Direct3D calculate the rectangle instead of you having to specify it. Using this flag does not cause any text to actually render.
- `DT_CENTER` centers text horizontally.
- `DT_EXPANDTABS` expands tab characters (the default is eight characters per tab).
- `DT_LEFT` left-aligns the text.
- `DT_NOCLIP` draws text without clipping, which can sometimes be faster to render.
- `DT_RIGHT` right-aligns the text.
- `DT_RTLREADING` displays text in right-to-left reading order for bi-directional text for Hebrew or Arabic fonts. The default is left-to-right.
- `DT_SINGLELINE` displays text on one line and ignores any new-line characters.
- `DT_TOP` top-justifies the text.
- `DT_VCENTER` centers text vertically and can only be used with single-line-only text.
- `DT_WORDBREAK` adds word breaks for lines of text that do not fit on the line specified by the rectangle.

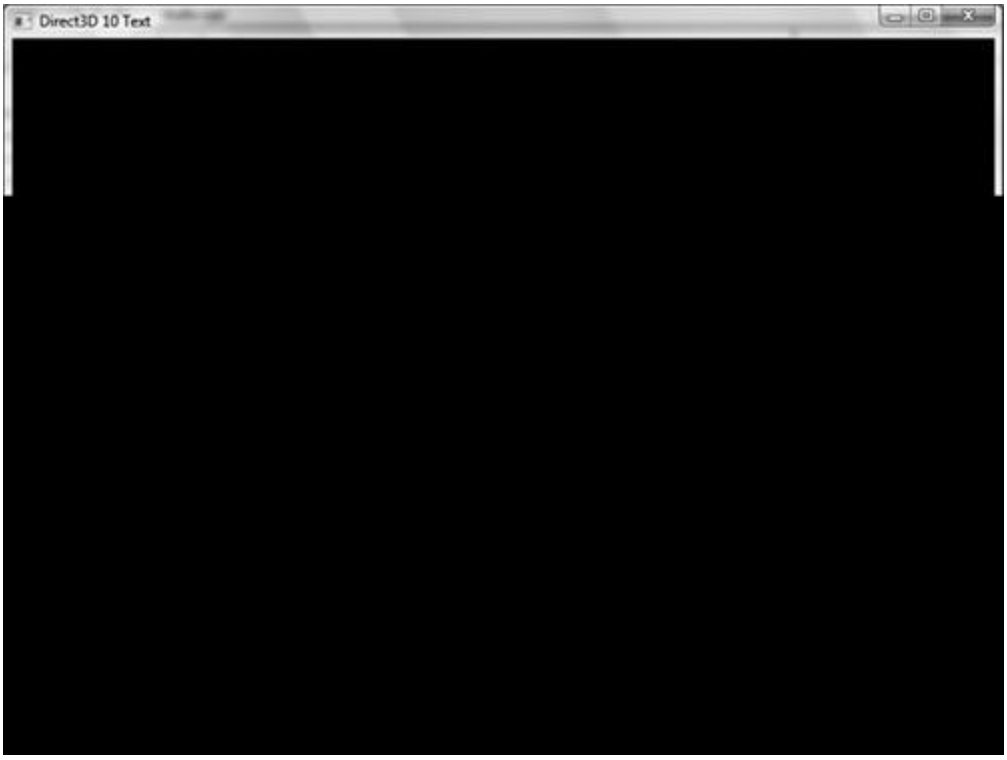
The last parameter is the RGBA color in which the text should be displayed. With two functions, one to create the font and one to render the text, we can display strings using Direct3D. Once you are completely done with a font object, it is important to call the object's `Release()` function to release it from memory. The Text demo's initialization, rendering, and shutdown functions are shown in Listing 2.9. Figure 2.5 shows a screenshot of the demo's results.

LISTING 2.9. THE DEMO'S INITIALIZATION FUNCTION

```
bool InitializeDemo()
{
   HRESULT hr;

   // Create the font.
   hr = D3DX10CreateFont(g_d3dDevice, 24, 0, FW_NORMAL, 0,
                         false, ANSI_CHARSET,
OUT_OUTLINE_PRECIS,
                         PROOF_QUALITY, VARIABLE_PITCH |
FF_SWISS,
                         "Arial", &g_font);

   if(FAILED(hr))
      return false;

   return true;
}

void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   // Clear the rendering destination to a specified color.

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

   // Display the text.
   D3DXCOLOR textCol = D3DXCOLOR(1, 1, 1, 1);

   RECT rect;
   rect.left = 300; rect.right = 800;
   rect.top = 300; rect.bottom = 800;

   char *str = "Hello World from DirectX 10";

   // NOTE you can have the rectangle calculated by using
   // DT_CALCRECT for the format.   Keep in mind that using
   // that will not draw that text so you will need two calls
   // With one using DT_CALCRECT and another using, for example,
   // DE_LEFT.

   g_font->DrawText(NULL, str, -1, &rect, DT_LEFT, textCol);

   // Display the results to the target window (swap chain).

   g_swapChain->Present(0, 0);
}

void Shutdown()
{
   // Release all used memory.
```

```
    if(g_d3dDevice) g_d3dDevice->ClearState()
    if(g_swapChain) g_swapChain->Release()
    if(g_renderTargetView) g_renderTargetView->Release();

    if(g_font) g_font->Release();

    if(g_d3dDevice) g_d3dDevice->Release();
}
```

FIGURE 2.5. A SCREENSHOT FROM THE TEXT DEMO.



The rectangle, which uses the `RECT` type, allows each corner of the rectangle to be specified. It is not necessary to fill in this object if you are using `DT_CALCRECT` as the flag in the rendering. If this flag is used, you must use the draw call twice, once with the flag and once without it. Using the flag causes the text not to draw, so that the call can be used for calculating the `RECT`. Calling the text drawing function a second time with the `RECT` object that was passed to the first call draws the text to the screen. Using `?1` for the string's length causes the string's length to be calculated internally by the drawing function.

ADDITIONAL DIRECTX TOPICS

A few additional DirectX-related topics should be discussed before moving on. These topics include the DXGI and DXUT.

DXGI

The DirectX Graphics Infrastructure's (DXGI) main goal is to manage low-level tasks related to rendering that is independent of the run-time. The DXGI is used to support future graphics components through a common framework. In an application the DXGI is used to

talk directly to the kernel mode of the operating system, which at the time of writing this is Windows Vista.

The Direct3D 10 core uses the DXGI for you. You have the option of talking directly to the DXGI, allowing you to bypass the Direct3D 10 core, which can be useful if your application needs direct control over the enumeration, acquiring, and control of graphics-related devices. We saw an example of the DXGI in the function `D3D10CreateDeviceAndSwapChain()`, where the first parameter is an adapter that can be used to enumerate multiple GPU devices.

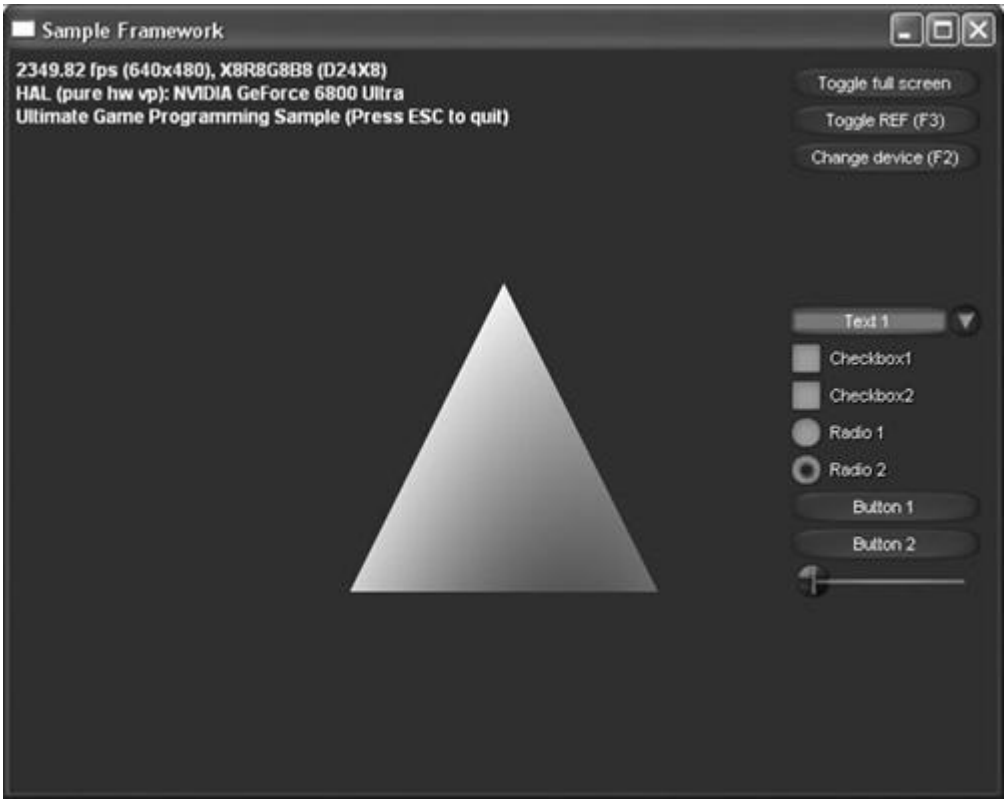
Advanced use of the DXGI is beyond the scope of this book. It might be worth looking into advanced DXGI uses if a machine has more than one device, such as a graphics card, and the application is able to use them independently of one another.

Swap chains are part of the DXGI.

DXUT FRAMEWORK

The DirectX Utility (DXUT) is a utility layer built on top of Direct3D. The purpose of the DXUT is to make it easier to create devices and windows and to manage messages in an application. Along with device and window creation and management, optional components are part of the DXUT, such as cameras, graphical user-interface systems, and mesh handling. The DXUT works by exposing a range of callback functions that programmers can implement for their applications. A screenshot of an application using the DXUT is shown in Figure 2.6.

FIGURE 2.6. A SCREENSHOT FROM AN APPLICATION USING THE DXUT.



A project using the DXUT can be created through the DirectX SDK Sample Browser. It is recommended that you use the DirectX SDK Sample Browser to create an empty DXUT project in order to examine its code. The DXUT is not used in this book. Instead, all of the demos are manually created the long way for educational purposes. Once you are familiar and comfortable with DirectX, it might be useful to look into using the DXUT.

SUMMARY

The following elements were discussed in this chapter:

- Direct3D 9
- Direct3D 10
- DirectDraw
- HAL and HEL
- Direct3D devices
- Swap chains
- Render target views
- Hardware versus reference mode
- Rendering buffers
- Clearing and displaying rendering targets
- Double buffering
- Page flipping
- Viewports
- Text and font in Direct3D 10
- DXGI
- DXUT

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What is Direct3D? What is DirectDraw? How are Direct3D and DirectDraw related?

2. What does HAL stand for?

   A. Hardware Application Layer

   B. Hardware Abstraction Layer

   C. It is not short for anything

   D. None of the above

3. What does HEL stand for?

   A. Hardware Emulation Layer

   B. Hardware Experience Layer

   C. It is not short for anything

   D. None of the above

4. What is the name of the Direct3D 9 version from Vista?

   A. Direct3D 9V

   B. Direct3D 9 Vista

   C. Direct3D 9L

   D. Vista is Direct3D 10 only

5. What does REF stand for in Direct3D?

6. Describe page flipping.

7. Describe double buffering.

8. What are swap chains? How do they differ from Direct3D 9 and Direct3D 10?

9. What are render target views? What is their purpose in Direct3D?

10. When drawing text, what flag can be used to calculate the rectangle of the text? What is the side effect of using this flag when it comes to drawing text?

11. List three high-level programmable shading languages.

12. What is T&L? When was T&L added to Direct3D?

13. List four purposes to displaying text in a video game.

14. List the two functions needed to create and display text using Direct3D 10. Describe each of the parameters the functions take.

15. True or false: HAL and HEL were introduced in Direct3D 10.

16. True or false: Direct3D supports software rendering and hardware graphics.

17. True or false: Direct3D 9 is the first API to do away with the fixed-function pipeline.

18. True or false: Page flipping copies data from one buffer to another when it is time to display a rendered scene.

19. True or false: A swap chain in Direct3D 10 is tied to the window.

20. True or false: Render targets inform Direct3D where to store the results of a rendering.

CHAPTER EXERCISES

Exercise 1: Change the clear color in the Blank Window demo to blue. Change it again to gray.

Exercise 2: Using the Text demo, display a paragraph of text, at least four lines, using word wrap. Allow Direct3D to calculate the rectangle for the text you are trying to draw.

Exercise 3: Add two additional font objects to the Text demo. Make each font a different type (i.e., Times New Roman, Arial, etc.) and make each one twice as big as the one before it.

3. RENDERING GEOMETRY

In This Chapter

- Primitives
- Colors

Rendering geometry is highly important to the success of all modern video games. Let's face it, video games today have a tremendous amount of data that must be stored, maintained, and processed to create the visuals necessary for the products we enjoy. The rendering of a scene needs to be efficient to allow the application to get the most performance from the drawing of any environment.

The purpose of this chapter is to discuss the basics of rendering geometry using Direct3D 10. This chapter will be the basis for most of the chapters that follow. In this chapter you will learn how to render geometry and work with colors, and you'll get a brief introduction to

effect files. Throughout this book we will look at more advanced rendering techniques and topics as we progress.

PRIMITIVES

In computer graphics there is this notion of primitives. A primitive is a very simple shape that when combined with other shapes can create more complex objects. The geometry of a 3D scene is defined entirely by primitives in modern video games.

The top three most common types of primitives include:

- Points
- Lines
- Triangles

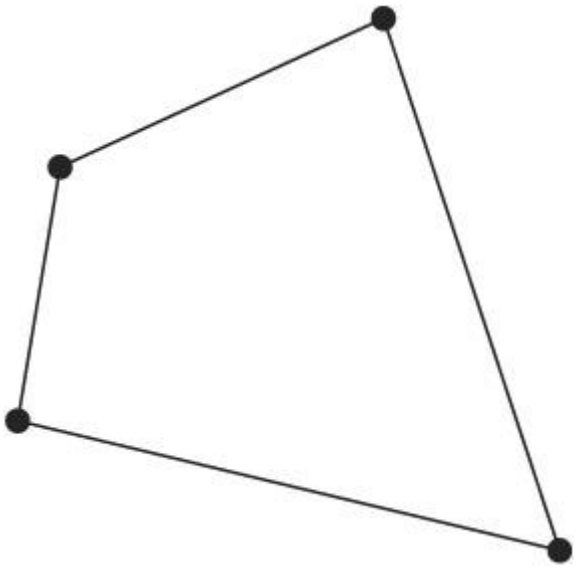
Other types of primitives exist in other systems. For example, in ray tracing, which is a way of generating images out of 3D data, most ray tracers offer primitives for spheres, boxes, and other simple shapes. In Direct3D the most complex primitive is the triangle. To render more complex shapes than that, such as spheres and boxes, an array of triangles is often used to create the virtual representation.

Points and lines are the least common types of basic primitives in most 3D games, while triangles are the most abundant. In this section we briefly discuss each of these types of primitives that can be rendered using Direct3D.

POINTS

Points are the smallest and most basic primitive that can be rendered in computer graphics. Points can be individual pixel locations or they can be 2D or 3D points in a virtual space. Points are the building blocks that are used to define other shapes, where the points of a shape are connected together and the area that fills in that shape is shaded. An example of this is shown in Figure 3.1.

FIGURE 3.1. POINTS BEING CONNECTED TO CREATE A SHAPE.



A single point that is part of a larger piece of geometry is known as a vertex point, and a vertex can be made up of any number of axes (e.g., the X and Y axis for a 2D vertex). In a 3D game using 3D data, this point is a 3D vertex. All geometry in a game defines vertex points, and in order to draw anything to the screen you must learn to define the vertex points of a piece of geometry. For complex objects such as character models and entire levels, modeling applications are usually used to create this geometry, which is then saved out to a file so that it can be loaded at runtime. Although complex models such as these use triangles, a triangle is made up of three connected points.

A vertex is a structure that defines a number of axes. In a 2D game, a 2D vertex has an X axis and a Y axis. In a 3D game, a vertex has an X axis, Y axis, and Z axis. An example of defining a 3D vertex in C++ can take on the following form.

```
struct Vertex3D
{
  float x;
  float y;
  float z;
};
```

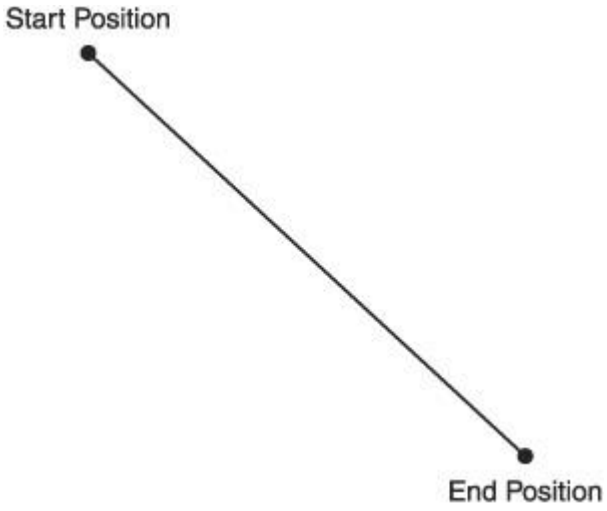
Usually, floating-point variables are used for the members of a vertex for their precision and are what Direct3D expects. Specifying a point using axis positions is similar to plotting points on a piece of graph paper in school. For example, if given a vertex at 1 unit for the X axis, 3 units for the Y axis, and 0 units for the Z axis, we can plot the position. A unit is a virtual form of measurement and is used as a means to give us a label much like inches and feet are used in the real world.

As mentioned previously, all geometry is defined by vertex positions that are connected together to form the outline of a shape. This shape is then shaded in using whatever technique you decide (e.g., with a solid color, with the color data of an image, etc.). Vertex points and vectors go hand-in-hand, and the terms are sometimes used synonymously. Vectors are discussed in more detail in Chapter 8, "Game Math." A vertex is a point that makes up a piece of geometry, while a vector is a direction.

LINE LISTS AND LINE STRIPS

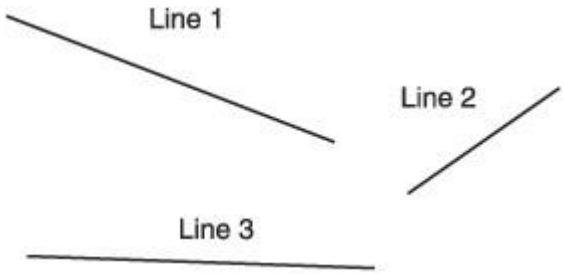
A line is the second most basic primitive that exists in computer graphics. A line is defined by two connected points, where one point is often referred to as the starting point and the other is the ending point. A visual of a line is shown in Figure 3.2.

FIGURE 3.2. A LINE DEFINED BY TWO CONNECTED POINTS.



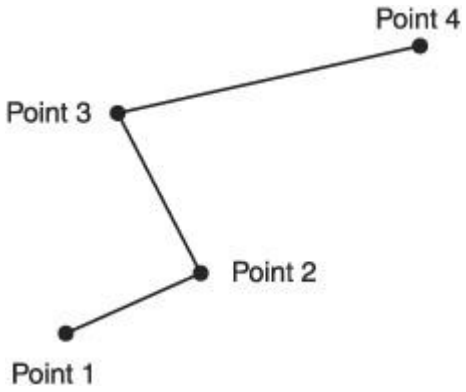
Two types of lines can be rendered in Direct3D: line lists and line strips. A line list is a list of lines. If you had an array of line objects, you would have a line list. A line list is a line where each pair of vertex points specifies a unique line. A visual of a line list is shown in Figure 3.3.

FIGURE 3.3. LINE LIST.



The second type of line is a line strip. The difference between a line strip and a line list is that a strip does not define two vertex points for each line separately. For example, for a line list it doesn't matter if one line shares a point with another line because that point has to be defined twice, uniquely for each line. In a line strip the first two points of the strip represent the first line. The third point marks the end of the second line because in a line strip the next line is defined by the ending point of the last line and the current point. The fourth point in a line strip specifies the third line because the third point, which is the end point of the second line, and the fourth point define yet another line. By using line strips that create one large connected line, you can save bandwidth because fewer vertex points are being used to create the shape. This occurs because the strip is reusing data instead of re-defining it for each line primitive. The key ideas to note are that a line strip cannot define disconnected lines, while a line list can have lines appear connected or disconnected. A visual of a line strip is shown in Figure 3.4.

FIGURE 3.4. A LINE STRIP.



TRIANGLE LISTS, STRIPS, AND FANS

A triangle is a three-point, or three-vertex, polygon. A polygon is any shape that is made up of three or more connecting points. This means a line is not considered a polygon because each line is essentially just two points. Also, for a primitive to be considered a polygon, it has to be a shape where each of the connecting points forms a closed area, which means it creates a solid shape with no holes leading out into the virtual universe. Three different types of triangles can be rendered in Direct3D: the triangle list, triangle strip, and triangle fan.

A triangle list is similar to a line list in the sense that it is an array of separately defined triangles where the triangle that came before and the triangle that comes after have no impact on the current triangle being rendered. A triangle strip is also like a line strip in the sense that the first three points make up the first triangle, the fourth point, when combined with the second and third point, make up the second triangle, and so forth. An example of a triangle list is shown in Figure 3.5, and an example of a triangle strip is shown in Figure 3.6. Notice how each triangle polygon creates a closed shape of connecting points.

FIGURE 3.5. A TRIANGLE LIST.

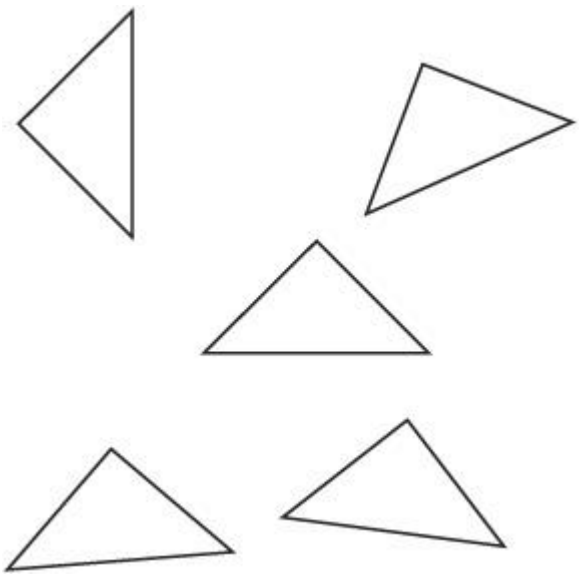
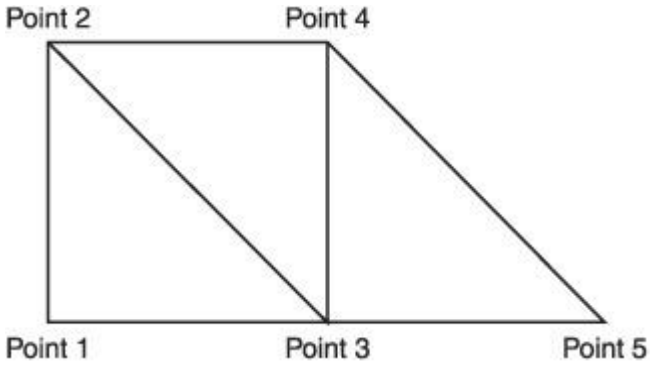


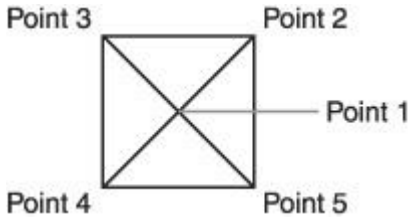
FIGURE 3.6. A TRIANGLE STRIP.



Because of the nature of triangle strips, they are often used to render terrain. Terrain in video games is the outside landscape of a virtual world. For example, to create a simple terrain, you can take a grid of triangles specified as a triangle strip (note that you could also use a list, but strips save more storage memory) and simply vary the Y axis for each point.

The last type of triangle is the triangle fan. A triangle fan is similar to a triangle strip, but the difference lies in how the points are connected. In a triangle fan the first three points represent the first triangle. The second triangle is defined by taking the first and third points and the fourth point, the third triangle is found by taking the first and fourth points and the fifth point, and so forth. This creates a fan-like object where all triangles of the object connect directly to the first vertex point. An example of this is shown in Figure 3.7.

FIGURE 3.7. A TRIANGLE FAN.



In this book we work extensively with triangles to create and draw complex objects such as characters, level objects, and level environments.

VERTEX BUFFERS AND INPUT LAYOUTS

Geometry in Direct3D must be stored and passed to the graphics hardware to be rendered. The storage of geometry is done using objects known as vertex buffers. A vertex buffer is essentially a buffer of memory that is used to store the geometry of a model. When you create a vertex buffer in Direct3D, you can send that buffer to the Direct3D device for rendering at any time.

A vertex buffer is an object of the type `ID3D10Buffer`. In fact, any type of buffer can be represented by the `ID3D10Buffer` type, which we discuss throughout this book. When using a vertex buffer, you can fill it with any vertex structure you want that you use to describe the geometry of your objects. Because of this, something known as an input layout is also used to tell Direct3D how your vertex points are represented. This allows developers to create their own vertex structures however they choose. A vertex point can be made up of more than just a position.

Other common attributes are properties for applying texture images, colors, directions, etc. An input layout is an object that uses the type ID3D10InputLayout. An example of creating two different types of vertex structures can be seen in the following, where the first structure creates a vertex that defines just a position property, while the second defines a structure that defines a position and color property.

```
struct DX10_Vertex
{
   float x, y, z;
};


struct DX10_VertexCol
{
   float x, y, z;
   char red, green, blue;
};
```

To create a vertex buffer we must first create a buffer description. A buffer description tells Direct3D how the buffer should be created; in this case we are discussing the creation of a vertex buffer. The buffer description takes on the following form and has the type D3D10_BUFFER_DESC.

```
typedef struct D3D10_BUFFER_DESC {
   UINT ByteWidth;
   D3D10_USAGE Usage;
   UINT BindFlags;
   UINT CPUAccessFlags;
   UINT MiscFlags;
} D3D10_BUFFER_DESC;
```

The first member of the buffer description is the number of vertex points in bytes. For a vertex buffer this number can be calculated by taking the number of vertex points and multiplying it by the size of your vertex structure in bytes. If you had six vertex points that you were defining this could have the following form.

```
D3D10_BUFFER_DESC.ByteWidth = sizeof(DX10_Vertex) * 6;
```

The second member of the buffer description is the usage flag, which tells Direct3D how the buffer is going to be used. This flag can be any one of the following values, where the DEFAULT states that the buffer needs read/write access by the GPU, IMMUTABLE states that the buffer can only be read by the GPU, DYNAMIC states that the buffer can be accessed by both the GPU (read-only) and CPU (write-only), and STAGING supports data transfer from the GPU to the CPU.

```
typedef enum D3D10_USAGE
{
   D3D10_USAGE_DEFAULT = 0,
   D3D10_USAGE_IMMUTABLE = 1,
   D3D10_USAGE_DYNAMIC = 2,
   D3D10_USAGE_STAGING = 3,
```

```
} D3D10_USAGE;
```

The third member of the buffer description is the BIND flag, which tells Direct3D how the buffer will be bound to the pipeline. This flag can be any one of the following values: VERTEX_BUFFER states that the buffer is a vertex buffer, INDEX_BUFFER states that the buffer is an index buffer (more on this later), CONSTANT_BUFFER is a buffer used by shaders (more on this in Chapter 6, "Shading and Surfaces"), SHADER_RESOURCE means the buffer has data used as a resource to a shader (e.g., texture, matrix, etc), STREAM_OUTPUT is an output buffer that is written to, RENDER_TARGET is a rendering destination, and DEPTH_STENCIL are additional types of destination buffers that are discussed later in this book.

```
typedef enum D3D10_BIND_FLAG
{
   D3D10_BIND_VERTEX_BUFFER = 0×1L,
   D3D10_BIND_INDEX_BUFFER = 0×2L,
   D3D10_BIND_CONSTANT_BUFFER = 0×4L,
   D3D10_BIND_SHADER_RESOURCE = 0×8L,
   D3D10_BIND_STREAM_OUTPUT = 0×10L,
   D3D10_BIND_RENDER_TARGET = 0×20L,
   D3D10_BIND_DEPTH_STENCIL = 0×40L,
} D3D10_BIND_FLAG;
```

The fourth member of the buffer description is the CPU access flag. A value of 0 can be used, which states that there is no access to this buffer by the CPU. If 0 is not used, the buffer's CPU flag can be set as CPU read or CPU write by using one of the following values.

```
typedef enum D3D10_CPU_ACCESS_FLAG
{
   D3D10_CPU_ACCESS_WRITE = 0×10000L,
   D3D10_CPU_ACCESS_READ = 0×20000L,
} D3D10_CPU_ACCESS_FLAG;
```

The last member of the buffer description is the buffer's MISC flag. A value of 0 can be used if this flag is unused by the application, or else it can be any of the following values, where the first and third enumerations are used by texture resources (discussed in Chapter 6) and the SHARED miscellaneous flag means the buffer can be shared by multiple GPU devices.

```
typedef enum D3D10_RESOURCE_MISC_FLAG
{
   D3D10_RESOURCE_MISC_GENERATE_MIPS = 0×1L,
   D3D10_RESOURCE_MISC_SHARED = 0×2L,
   D3D10_RESOURCE_MISC_TEXTURECUBE = 0×4L,
} D3D10_RESOURCE_MISC_FLAG;
```

An example of creating a vertex buffer using the buffer description follows.

```
D3D10_BUFFER_DESC buffDesc;
```

```
buffDesc.Usage = D3D10_USAGE_DEFAULT;
buffDesc.ByteWidth = sizeof(DX10_Vertex) * g_total_points;
buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
buffDesc.CPUAccessFlags = 0;
buffDesc.MiscFlags = 0;
```

Using one of the sample vertex descriptions from earlier in this section, you can create a list of vertex points by using the following, which creates six vertices.

```
DX10_Vertex vertices[] =
{
   { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) },
   { D3DXVECTOR3( 0.5f, -0.5f, 0.5f) },
   { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },

   { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },
   { D3DXVECTOR3(-0.5f,  0.5f, 0.5f) },
   { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) }
};
```

Vertices *is the plural of vertex.*

`D3DXVECTOR3` is a structure that defines a 3D vector, where a vector is a vertex that does not describe a point but instead describes a direction. Since vectors and vertex points have the same structure, they are often used synonymously. The `D3DXVECTOR3` structure is part of the Direct3D utility (D3DX).

Once a buffer description is filled and you have a list of geometry that makes up some object, with the example above creating a square shape (more on this later in this chapter), we are ready to create the vertex buffer itself. A vertex buffer is created by calling the Direct3D device function `CreateBuffer()`, which has the following function prototype.

```
HRESULT CreateBuffer(
   const D3D10_BUFFER_DESC *pDesc,
   const D3D10_SUBRESOURCE_DATA *pInitialData,
   ID3D10Buffer **ppBuffer
);
```

The first parameter for the `CreateBuffer()` function is the buffer description object, the second parameter is the data that makes up the geometry, and the last parameter is the address to the buffer object that will be created upon this function's success. The vertex data is passed to the `CreateBuffer()` function by using a `D3D10_SUBRESOURCE_DATA` object, which has the following structure, where the first member is a void pointer to the actual data, the second member is the size in bytes of the

system memory's pitch (which is only used by textures, as discussed in Chapter 6), and the last member is the system memory slice in bytes (which is only used by 3D textures).

```
typedef struct D3D10_SUBRESOURCE_DATA {
   const void *pSysMem;
   UINT SysMemPitch;
   UINT SysMemSlicePitch;
} D3D10_SUBRESOURCE_DATA;
```

Using each of the examples above you can take an array that makes up your geometry and create a vertex buffer out of it using the following example.

```
DX10_Vertex vertices[] =
{
   { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) },
   { D3DXVECTOR3( 0.5f, -0.5f, 0.5f) },
   { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },

   { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },
   { D3DXVECTOR3(-0.5f,  0.5f, 0.5f) },
   { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) }
};


// Create the vertex buffer.

D3D10_BUFFER_DESC buffDesc;
ID3D10Buffer *vertexBuffer;

buffDesc.Usage = D3D10_USAGE_DEFAULT;
buffDesc.ByteWidth = sizeof(DX10_Vertex) * 6;
buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
buffDesc.CPUAccessFlags = 0;
buffDesc.MiscFlags = 0;

D3D10_SUBRESOURCE_DATA resData;
resData.pSysMem = vertices;

hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
&vertexBuffer);

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

To render the vertex buffer, we also need to create the input layout. An input layout can be created by calling the `CreateInputLayout()` function, which has the following function prototype.

```
HRESULT CreateInputLayout(
   const D3D10_INPUT_ELEMENT_DESC *pInputElementDescs,
   UINT NumElements,
```

```
    const void *pShaderBytecodeWithInputSignature,
    SIZE_T BytecodeLength,
    ID3D10InputLayout **ppInputLayout
);
```

The first parameter of the `CreateInputLayout()` function is the input element description. An input element description tells Direct3D how each property of the vertex is defined. Since, as we've seen earlier in this section, it is possible to create your own vertex structures however you wish, it is important to tell Direct3D how these structures are laid out so that it knows how to access its data. An input element description uses the type `D3D10_INPUT_ELEMENT_DESC` and has the following structure.

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D10_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D10_INPUT_ELEMENT_DESC;
```

In the input element description, the first member is the name of the property that is referred to from within a shader file, which is discussed later in this chapter. The next member is the index for the element, which can be used if you have more than one element with the same name. The third member is the format (i.e., the data type of the element), and it can be any of the many `DXGI_FORMAT` types. The fourth member is a value between 0 and 15 that is used as the input assembler slot, which can be used for using multiple buffers at the same time. The fifth member is the byte offset, which is the total number of bytes between each element. The sixth member tells Direct3D if the element is used for each vertex (`D3D10_INPUT_PER_VERTEX_DATA`) or by each instance of an object (`D3D10_INPUT_PER_INSTANCE_DATA`). The last member, which also has to do with instancing, is the number of instances to draw using the same per-instance data. Instancing is discussed later in this book. An example of creating an input element description is as follows.

```
D3D10_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
```

In this example of an input element description, a single element is defined, which is the vertex position property. The first member in the element is the name that can be referenced by the shader followed by the index, format type, input slot, bytes between positions, input class (which states that the position is specified for each point), and instance rate, which is 0 since this example is not on instancing. Instancing is generally the ability to draw multiple objects to the screen using a single draw call and the same geometric data. An example of this can be seen by using instancing to draw a forest of trees or an asteroid field. Geometry instancing can greatly increase an application's performance, and the topic is discussed elsewhere in this book.

The second parameter to the `CreateInputLayout()` function is the number of elements, or properties, that make up the vertex structure. If you are specifying a position and a color, this would be two elements.

The third and fourth parameters deal with shaders. The third parameter is a pointer to compiled shader code, while the fourth parameter is the size of the compiled shader code in bytes. The last parameter is the address to the input buffer layout object that will be created upon this function's success.

For the third and fourth parameter of the `CreateInputLayout()` function, the data can be obtained from a loaded shader. Each shader, once loaded, has an input signature. This input signature is essentially used for these two parameters of this function. For example, if you have a shader technique, which is discussed in detail in Chapter 4, "Shader Model 4," you can get a pass description from it that also supplies us with the input signature of the shader. An example of doing so is as follows.

```
D3D10_PASS_DESC passDesc;

g_technique->GetPassByIndex(0)->GetDesc(&passDesc);

hr = g_d3dDevice->CreateInputLayout(layoutDesc, numElements,
                                    passDesc.pIAInputSignature,

passDesc.IAInputSignatureSize,
                                    &input_layout);
```

An example of putting the creation of the input layout with the creation of a vertex buffer follows.

```
g_technique = g_shader->GetTechniqueByName("PassThrough");


// Create the geometry.   The layout of each vertex is made up
// of just a position so that is all we need.

D3D10_INPUT_ELEMENT_DESC layoutDesc[] =
{
   { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D10_INPUT_PER_VERTEX_DATA, 0 },
};

unsigned int numElements = sizeof(layout) / sizeof(layout[0]);

D3D10_PASS_DESC passDesc;
ID3D10InputLayout *input_layout = NULL;

g_technique->GetPassByIndex(0)->GetDesc(&passDesc);

hr = g_d3dDevice->CreateInputLayout(layoutDesc, numElements,
                                    passDesc.pIAInputSignature,

passDesc.IAInputSignatureSize,
                                    &input_layout);
```

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

To recap, the steps to prepare geometry for rendering are to create an input layout using the information of the vertex structure you've created and the input signature of the shader, which is a validation that they are compatible, and the creation of a vertex buffer. The input layout is created by calling `CreateInputLayout()` on the Direct3D device, and the vertex buffer is created by calling `CreateBuffer()` and specifying that a vertex buffer is being created in the buffer description.

DRAWING PRIMITIVES

Drawing primitives in Direct3D 10 requires up to four calls. The steps to draw geometry are to set the input layout, set the vertex buffer, set the type of primitives you are drawing (e.g., triangle lists, line lists, etc.), and draw the geometry with the `Draw()` function of the Direct3D device. Before drawing the actual geometry with `Draw()`, the data is set to the device using the input assembler. The input assembler functions start with the prefix IA.

To set the input layout you use a call to the Direct3D device's function `IASetInputLayout()`. This function takes as a parameter the input layout object created when you created the vertex buffer. The function prototype for the `IASetInputLayout()` function can be seen as follows.

```
void IASetInputLayout(
   ID3D10InputLayout *pInputLayout
);
```

To set the vertex buffer the Direct3D device, call the device's `IASetVertexBuffers()` function. The `IASetVertexBuffers()` function's first parameter, the start slot, can allow you to set the first vertex buffer to one of 16 slots, which allows you to set multiple buffers at the same time. By default, this value is 0. The second parameter is for the number of vertex buffers you are setting with the function call. The third parameter is a list of one or more vertex buffers to set. The fourth parameter is the size of each vertex for each vertex buffer, and the last parameter is an offset from the start of the buffer to the first element you want to start with for each vertex buffer. The function prototype for the `IASetVertexBuffers()` function is as follows.

```
void IASetVertexBuffers(
   UINT StartSlot,
   UINT NumBuffers,
   ID3D10Buffer *const *ppVertexBuffers,
   const UINT *pStrides,
   const UINT *pOffsets
);
```

Before you can draw geometry, Direct3D has to know what type of primitives are being drawn. This is done by calling the function `IASetPrimitiveTopology()`, which takes as a parameter the type of primitive that is being drawn. The function prototype for `IASetPrimitiveTopology()` is:

```
void IASetPrimitiveTopology(
```

```
    D3D10_PRIMITIVE_TOPOLOGY Topology
);
```

The type of geometry you are drawing determines the value for the `PRIMITIVE_TOPOLOGY` function. This value can be any one of the following.

```
typedef enum D3D10_PRIMITIVE_TOPOLOGY
{
    D3D10_PRIMITIVE_TOPOLOGY_UNDEFINED = 0,
    D3D10_PRIMITIVE_TOPOLOGY_POINTLIST = 1,
    D3D10_PRIMITIVE_TOPOLOGY_LINELIST = 2,
    D3D10_PRIMITIVE_TOPOLOGY_LINESTRIP = 3,
    D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST = 4,
    D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP = 5,
    D3D10_PRIMITIVE_TOPOLOGY_LINELIST_ADJ = 10,
    D3D10_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ = 11,
    D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ = 12,
    D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ = 13,
} D3D10_PRIMITIVE_TOPOLOGY;
```

Once the input layout, vertex buffer, and topology have been set, the last step is to draw the actual geometry. Again this is done by calling the Direct3D device's `Draw()` function. Direct3D draws the data in the input assembler, which was set with the three functions discussed in this section. The two parameters the function takes are the total number of vertices that are to be drawn and the index for the starting vertex. Specifying the starting index allows you to partially draw a model, or else you can use 0 to draw the entire model. The function prototype for the `Draw()` function is as follows.

```
void Draw(
    UINT VertexCount,
    UINT StartVertexLocation
);
```

RENDERING STATES

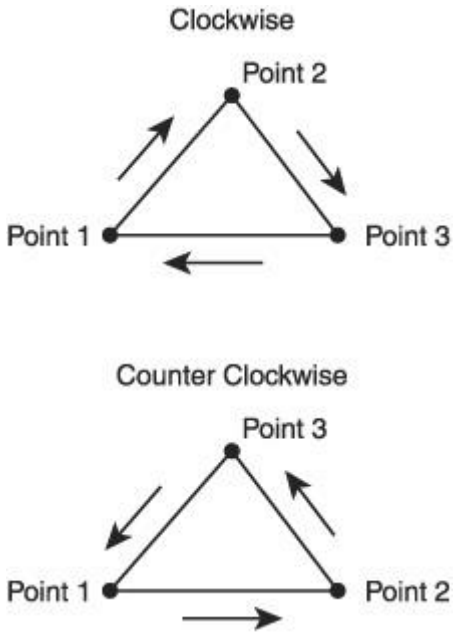
Before concluding the discussion on the basics of rendering geometry, we will also discuss rendering states. A rendering state, or in Direct3D 10 terms, a rasterizer state, allows you to specify special states that affect how geometry is drawn. A rendering state is represented by an object of type `ID3D10RasterizerState`, and the description of the state uses the object type `D3D10_RASTERIZER_DESC`. The description is filled in, a function is called, and the rasterizer state object is created. This object can then be set at any time for Direct3D.

In versions before Direct3D 10, rasterizer states were called rendering states and were simply flags

The creation of a rasterizer state object is done with a call to the device's function `CreateRasterizerState()`, which takes as parameters the address to the state description and the address to the object that will be created by the function. The state can then be set with a call to the `RSSetState()` function of the rendering device, which takes only the state object as a parameter.

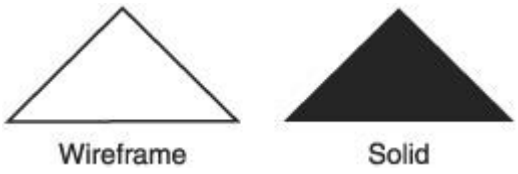
The states that can be set include the cull and fill mode, depth bias, depth clipping, scissor test, and anti-aliasing properties. The cull mode can be set to front face culling or back face culling. Front face culling means that any polygon facing toward the camera will not be rendered, while back face culling, which is a popular way to increase performance in some scenes, does not render geometry facing away from the camera. When working with 3D models, this is important since polygons facing away from the camera are often blocked by polygons facing the camera. By not rendering them, an application can potentially increase performance by not drawing geometry that can't be seen anyway. The flags for the cull mode can be either `D3D10_CULL_NONE` for no culling, `D3D10_CULL_FRONT` for front face culling, or `D3D10_CULL_BACK` for back face culling. You can also set a property that tells Direct3D if clockwise or counterclockwise order is used for the polygons. Clockwise order means the first, second, and third vertex points of a triangle, for example, move in the same direction the hands of a clock move, while counterclockwise order is the opposite. An example is shown in Figure 3.8. You can specify vertex points in any order, so the only way Direct3D knows which order is front facing and which is back facing is through this property. We'll see how to set it later in this section.

FIGURE 3.8. CLOCKWISE VERSUS COUNTERCLOCKWISE ORDER.



The fill mode can be set to either `D3D10_FILL_SOLID` for normal solid rendering or `D3D10_FILL_WIRE` for wireframe rendering. Wireframe rendering draws just the outline of polygons instead of the surface that makes them up, while solid renders the inside of the surface. An example of fill mode versus wireframe mode is shown in Figure 3.9.

FIGURE 3.9. FILL VERSUS WIREFRAME.



The remaining properties deal with topics that are discussed throughout this book. For now, an example of creating a rasterizer state object and description and of setting the state are shown in the following example.

```
ID3D10RasterizerState *rsState;
D3D10_RASTERIZER_DESC rsStateDesc;

rsStateDesc.FillMode = D3D10_FILL_SOLID;
rsStateDesc.CullMode = D3D10_CULL_NONE;
rsStateDesc.FrontCounterClockwise = true;
rsStateDesc.DepthBias = false;
rsStateDesc.DepthBiasClamp = 0;
rsStateDesc.SlopeScaledDepthBias = 0;
rsStateDesc.DepthClipEnable = true;
rsStateDesc.ScissorEnable = false;
rsStateDesc.MultisampleEnable = false;
rsStateDesc.AntialiasedLineEnable = false;

g_d3dDevice->CreateRasterizerState(&rsStateDesc, &rsState);
g_d3dDevice->RSSetState(rsState);
```

Although we do not have any need for rasterizer states in this chapter, we see a simple example of how to use them in the next demo. Later on, we dive deeper in rasterizer states.

PRIMITIVES DEMO

On the accompanying CD-ROM is a demo called Primitives in the Chapter 3 folder. This demo draws two triangles in wireframe mode. In the demo's main source file the global section defines an input layout, vertex buffer, and vertex structure. The vertex structure is the custom structure used to define each individual vertex. In this demo we only specify vertices with positions, which are done using the type `D3DXVECTOR3`. Technically, we can use just the `D3DXVECTOR3` structure, but in future demos we add properties to the vertex structure seen in this demo. The global section from the Primitives demo application is shown in Listing 3.1. The code also specifies a shader and a shader technique, both of which are discussed in more detail in Chapter 4. Shaders are created by using the `ID3D10Effect` type. A technique is created using the type `ID3D10EffectTechnique`. A shader can have multiple effects or variations of effects, known as techniques, so we must tell Direct3D which one to use. Shaders are a complex topic and are discussed in more detail in the next chapter.

LISTING 3.1. THE GLOBAL SECTION FROM THE PRIMITIVES DEMO

```
#include<windows.h>
#include<d3d10.h>
#include<d3dx10.h>
```

```
#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME      "Primitives"
#define WINDOW_WIDTH     800
#define WINDOW_HEIGHT    600

// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

// Effect objects.
ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_technique = NULL;

// Display object to store scene geometry.
ID3D10InputLayout *g_layout = NULL;
ID3D10Buffer *g_vertexBuffer = NULL;

// Structure used to represent a single vertex.
struct DX10_Vertex
{
   D3DXVECTOR3 pos;
};
```

The initialization function from the Primitives demo starts by loading the shader that will be used for rendering. The shader is loaded by calling the function `D3DX10CreateEffectFromFile()`. This function takes as its first parameter the file name of the shader and as the next to last parameter the shader object to be created. The remaining parameters of this function are discussed in the shader chapter (Chapter 4). Once a shader is loaded, the technique specified in the file is obtained by name.

After the shader is loaded and prepared, the input layout is created. This is followed by the creation of a vertex buffer, which specifies two triangles to create the shape of a square. Once the shader, input layout, and vertex buffer are created, the initialization is complete. The demo's initialization function is shown in Listing 3.2.

LISTING 3.2. THE DEMO'S INITIALIZATION FUNCTION

```
bool InitializeDemo()
{
   // Load the shader.

   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   HRESULT hr = D3DX10CreateEffectFromFile("shader.fx", NULL,
NULL,
                                           "fx_4_0", shaderFlags,
```

```
0,
                                                g_d3dDevice, NULL,
NULL,
                                                &g_shader, NULL,
NULL);
   if(FAILED(hr))
      return false;

   // There is only one technique in this shader.   Since this
   // is the only effect we can grab it once now.

   g_technique = g_shader->GetTechniqueByName("PassThrough");


   // Create the geometry.   The layout of each vertex is made
up
   // of just a position so that is all we need.

   D3D10_INPUT_ELEMENT_DESC layout[] =
   {
      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
   };

   unsigned int numElements = sizeof(layout) /
sizeof(layout[0]);
   D3D10_PASS_DESC passDesc;

   g_technique->GetPassByIndex(0)->GetDesc(&passDesc);

   hr = g_d3dDevice->CreateInputLayout(layout, numElements,
                                       passDesc.pIAInputSignature,

passDesc.IAInputSignatureSize,
                                       &g_layout);
   if(FAILED(hr))
      return false;

   DX10_Vertex vertices[] =
   {
      { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) },
      { D3DXVECTOR3( 0.5f, -0.5f, 0.5f) },
      { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },

      { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },
      { D3DXVECTOR3(-0.5f,  0.5f, 0.5f) },
      { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) }
   };

   // Create the vertex buffer.

   D3D10_BUFFER_DESC buffDesc;
```

```
    buffDesc.Usage = D3D10_USAGE_DEFAULT;
    buffDesc.ByteWidth = sizeof(DX10_Vertex) * 6;
    buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
    buffDesc.CPUAccessFlags = 0;
    buffDesc.MiscFlags = 0;

    D3D10_SUBRESOURCE_DATA resData;
    resData.pSysMem = vertices;

    hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
                                   &g_vertexBuffer);

    if(FAILED(hr))
        return false;

    return true;
}
```

The next function is the rendering function. The rendering function from the Primitives demo application starts by clearing the rendering target for the screen. Next it sets the rasterizer state to render in wireframe mode. Technically only the `FillMode` property needs to be set to `D3D10_FILL_WIREFRAME` for this effect. Since the purpose is to show how to work with rasterizer states, all of the states are set to their default values so that you can see how they are set and with what values.

After the rasterizer state is set, the rendering function proceeds to set up the input layout and vertex buffer using the input assembler functions. Once that is set, the function renders the geometry using the `Draw()` method. In a shader an effect can be implemented in multiple passes, so a loop is used to render the geometry once for all necessary passes. Most effects seen in this book are done using one pass, so this is optional in those cases. Looping through the passes of an effect technique is done here to show how it is done. There is no harm in looping through the passes of a shader if the shader only has one pass. (The discussion on everything dealing with shaders, effects, and techniques is in Chapter 4.) In a game situation you might not be able to assume that the shaders loaded use only one pass, so looping will become necessary in those situations.

In addition to the other Direct3D objects released in past demos, this demo also releases the vertex buffer, input layout, and shader effect from memory. These objects consume allocated resources, and their release is important. The rendering and shutdown functions from the Primitives demo application are shown in Listing 3.3.

LISTING 3.3. THE RENDERING FUNCTION FROM THE PRIMITIVES DEMO

```
void RenderScene()
{
    float col[4] = { 0, 0, 0, 1 };

    // Clear the rendering destination to a specified color.

    g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

    // Do not need to set the render state each frame but this
    // is here so it doesn't get lost in the long initialize
```

```
code.
   // Don't want readers overlooking it.

   ID3D10RasterizerState *rsState;
   D3D10_RASTERIZER_DESC rsStateDesc;

   //rsStateDesc.FillMode = D3D10_FILL_SOLID;
   rsStateDesc.FillMode = D3D10_FILL_WIREFRAME;
   rsStateDesc.CullMode = D3D10_CULL_NONE;
   rsStateDesc.FrontCounterClockwise = true;
   rsStateDesc.DepthBias = false;
   rsStateDesc.DepthBiasClamp = 0;
   rsStateDesc.SlopeScaledDepthBias = 0;
   rsStateDesc.DepthClipEnable = true;
   rsStateDesc.ScissorEnable = false;
   rsStateDesc.MultisampleEnable = false;
   rsStateDesc.AntialiasedLineEnable = false;

   g_d3dDevice->CreateRasterizerState(&rsStateDesc, &rsState);
   g_d3dDevice->RSSetState(rsState);
   unsigned int stride = sizeof(DX10_Vertex);
   unsigned int offset = 0;

   // Setup the geometry buffer that will be rendered.

   g_d3dDevice->IASetInputLayout(g_layout);
   g_d3dDevice->IASetVertexBuffers(0, 1, &g_vertexBuffer,
&stride,
                                   &offset);
   g_d3dDevice->IASetPrimitiveTopology(
      D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   // Prepare the effect we will use to draw the geometry.

   D3D10_TECHNIQUE_DESC techDesc;
   g_technique->GetDesc(&techDesc);

   // Loop through each pass of the technique and draw.

   for(unsigned int i = 0; i < techDesc.Passes; i++)
   {
      g_technique->GetPassByIndex(i)->Apply(0);
      g_d3dDevice->Draw(6, 0);
   }

   // Display the results to the target window (swap chain).

   g_swapChain->Present(0, 0);
}


void Shutdown()
```

```
{
    // Release all used memory.
    if(g_d3dDevice) g_d3dDevice->ClearState();
    if(g_swapChain) g_swapChain->Release();
    if(g_renderTargetView) g_renderTargetView->Release();

    if(g_shader) g_shader->Release();
    if(g_layout) g_layout->Release();
    if(g_vertexBuffer) g_vertexBuffer->Release();

    if(g_d3dDevice) g_d3dDevice->Release();
}
```

In this chapter we briefly discuss the shader-related code without any detail. This changes in the next chapter when we discuss shaders and their related code in detail. The topic is large and requires a chapter all to itself. Although it is assumed that you have no experience with shaders, we will look at the simple shader used for this demo to see what a shader looks like. This shader is written in Direct3D's shading language HLSL and uses Shader Model 4.0, each of which are discussed further in the next chapter.

The shader for the Primitives demo is stored in a file called shader.fx. In this file it starts by declaring an incoming vertex, which matches the vertex structure used to define the vertex data in the application. In this demo the incoming vertex only specifies a position, which is what the shader expects as well. Along with an input vertex, it also specifies an input to the pixel shader, which is supplied by the vertex shader, and an output to the pixel shader. The pixel shader's output is a single color that is used to color in that pixel on the screen. The input to the pixel shader, which is the vertex shader's output, simply passes the transformed vertex down the pipeline.

The vertex shader is responsible for passing along the data down the pipeline, and the pixel shader is responsible for passing along a color to the shader for the screen pixel. The color used for the pixel shader's output is yellow. Because no special operations need to take place on the incoming vertex, it is simply passed along. The final vertex is stored in the register marked by the tag `SV_POSITION`. The final color value from a pixel shader is stored in the register marked by the tag `SV_TARGET`. These are known as semantics and are discussed in more detail in the next chapter.

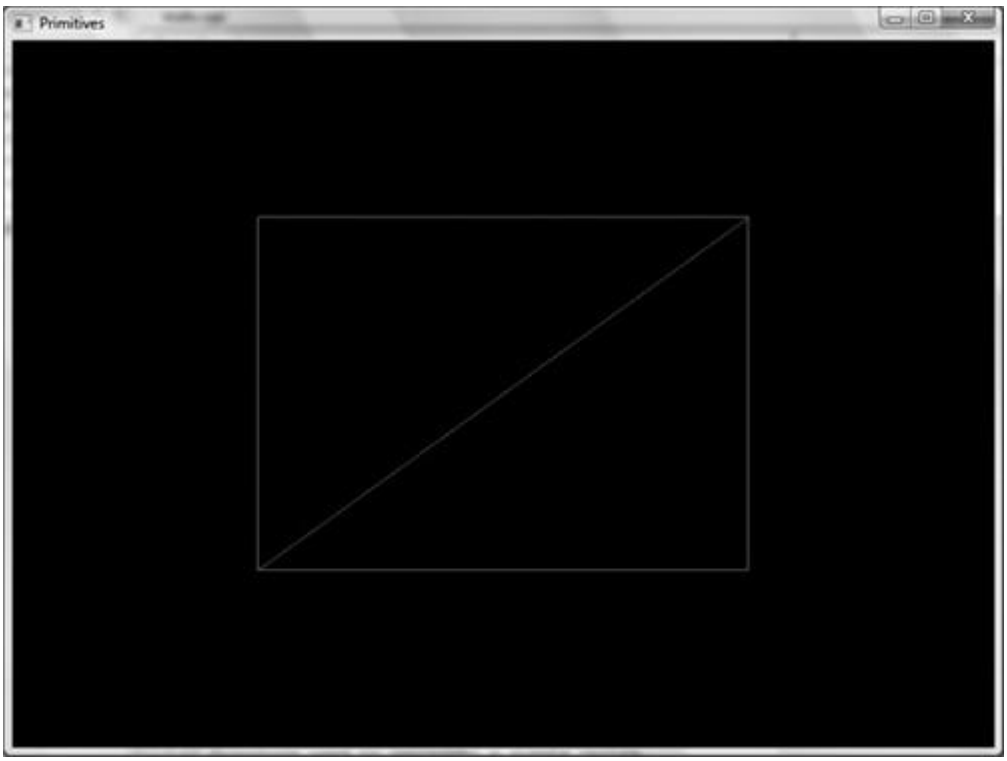
Once the vertex and pixel shaders have been defined, the shader ends by defining a technique. A technique is an effect. In a shader you can have multiple code functions for vertex, pixel, and geometry shaders. You can mix and match these to create different effects (techniques). In the application you can choose which effect you want to use from the shader. Declaring a technique requires the use of the `technique10` keyword followed by any name you choose. This name is referenced by the application when obtaining a technique object.

Inside the technique you set the vertex, geometry, and pixel shader. This is done by calling the `SetVertexShader()`, `SetGeometryShader()`, and `SetPixelShader()` HLSL functions, which take as parameters the compiled code from the vertex, geometry, and pixel shader functions, respectively. To compile a function to be used by one of these set shader functions, you call the HLSL function `CompileShader()`, which takes as a parameter the shader version you want to use (`4_0` if using Shader Model 4.0) and a function that stores the shader's code. The `NULL` keyword can be used for any shader that is not specified, such as the geometry shader in this demo. The Primitives demo's shader is shown in Listing 3.4. Figure 3.10 shows a screenshot of the demo. Again, HLSL and everything dealing with shaders in Direct3D 10 are discussed in the next chapter.

LISTING 3.4. THE PRIMITIVES DEMO'S SHADER

```
struct VS_INPUT
{
   float4 Pos : POSITION;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
};

PS_INPUT VS(VS_INPUT input)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = input.Pos;

   return output;
}
float4 PS(PS_INPUT input)  : SV_Target
{
   return float4(1, 0, 1, 1);
}

technique10 PassThrough
{
   pass P0
   {
      SetVertexShader(CompileShader(vs_4_0, VS()));

      SetGeometryShader(NULL);

      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

FIGURE 3.10. A SCREENSHOT FROM THE PRIMITIVES DEMO.



INDICES AND PRIMITIVES

So far, the geometry of objects has been defined entirely by specifying vertex points. Many of these vertex points tend to overlap one another. For example, the two triangles in the Primitives demo application share two vertex points. Instead of defining those points more than once, wouldn't it be great if they could be defined only once and used multiple times? If we had a complex object, such as a terrain or a character, there could be hundreds if not thousands of duplicate points, each costing memory resources. In Direct3D this can be solved by using something known as a vertex index.

A vertex index (indices for its plural form) is an array index. When you give Direct3D a list of vertex points, it essentially receives an array of vertices. Each triangle of an object can be defined as having an array of unique vertex points and an indices list. The indices list specifies each primitive of the object by indexing each vertex from the vertex list. Using indices, each vertex can be used as many times as necessary. Without indices, each vertex that shares a specific position must be duplicated.

On the CD-ROM in the Chapter 3 folder is a demo application called Indices. In this section we briefly discuss the demo application, which is built on top of the Primitives demo. To create an index list you can literally create an array of integers. Direct3D 10 supports 32-bit and 16-bit unsigned integers for indices. An example of using 32-bit indices and a reduced vertex count from the Primitives demo can be seen in Listing 3.5.

LISTING 3.5. DEFINING THE INDICES AND VERTICES FROM THE INDICES DEMO

```
unsigned int indices[] = { 0, 1, 2, 2, 3, 0 };

DX10_Vertex vertices[] =
{
```

```
    { D3DXVECTOR3( 0.5f,  0.5f, 0.5f) },
    { D3DXVECTOR3( 0.5f, -0.5f, 0.5f) },
    { D3DXVECTOR3(-0.5f, -0.5f, 0.5f) },
    { D3DXVECTOR3(-0.5f,  0.5f, 0.5f) },
};
```

The index list from Listing 3.5 specifies, using triangles as an example, that the first triangle is composed of the first, second, and third vertex points in the vertex list. The second triangle is specified by the third, fourth, and first vertex points. Each index in the index list is an array index that matches the data in the vertex list. You can specify the vertices in any order you want since vertex indices are used to specify each primitive, and not the vertex order, as was the case in the Primitives demo.

Earlier we mentioned that the `ID3D10Buffer` can be used to create a vertex buffer as well as other types of buffers and in the Indices demo it is also used to specify an index buffer. The index buffer is created in the same manner as the vertex buffer, with the exception of using the `D3D10_BIND_INDEX_BUFFER` flag for the `BindFlags` member of the `D3D10_BUFFER_DESC` structure. The creation of the index buffer, as well as the vertex buffer, from the Indices demo is shown in Listing 3.6.

LISTING 3.6. THE CREATION OF THE INDEX AND VERTEX BUFFER FROM THE INDICES DEMO

```
D3D10_BUFFER_DESC vbBuffDesc, ibBuffDesc;

vbBuffDesc.Usage = D3D10_USAGE_DEFAULT;
vbBuffDesc.ByteWidth = sizeof(DX10_Vertex) * 4;
vbBuffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
vbBuffDesc.CPUAccessFlags = 0;
vbBuffDesc.MiscFlags = 0;

ibBuffDesc.Usage = D3D10_USAGE_DEFAULT;
ibBuffDesc.ByteWidth = sizeof(unsigned int) * 6;
ibBuffDesc.BindFlags = D3D10_BIND_INDEX_BUFFER;
ibBuffDesc.CPUAccessFlags = 0;
ibBuffDesc.MiscFlags = 0;

D3D10_SUBRESOURCE_DATA vbResData, ibResData;
vbResData.pSysMem = vertices;
ibResData.pSysMem = indices;

hr = g_d3dDevice->CreateBuffer(&vbBuffDesc, &vbResData,
                               &g_vertexBuffer);

if(FAILED(hr))
   return false;
hr = g_d3dDevice->CreateBuffer(&ibBuffDesc, &ibResData,
                               &g_indexBuffer);
if(FAILED(hr))
   return false;
```

The index buffer must be set to the input assembler just as the vertex buffer was. This is done using the function `IASetIndexBuffer()`, and it takes as parameters the index

buffer, the format of the index buffer, and the offset. The offset can be used to specify which index you want to start with. The format can be either `DXGI_FORMAT_R32_UINT` for 32-bit unsigned integers or `DXGI_FORMAT_R16_UINT` for 16-bit unsigned integers. Setting the index buffer, along with the other input assembler objects, is shown in Listing 3.7 from the Indices demo.

LISTING 3.7. SETTING THE INDEX BUFFER WITH THE VERTEX BUFFER, INPUT LAYOUT, AND TOPOLOGY

```
g_d3dDevice->IASetInputLayout(g_layout);
g_d3dDevice->IASetVertexBuffers(0, 1, &g_vertexBuffer, &stride,
                                &offset);
g_d3dDevice->IASetIndexBuffer(g_indexBuffer,
DXGI_FORMAT_R32_UINT,
                              0);
g_d3dDevice->IASetPrimitiveTopology(
   D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

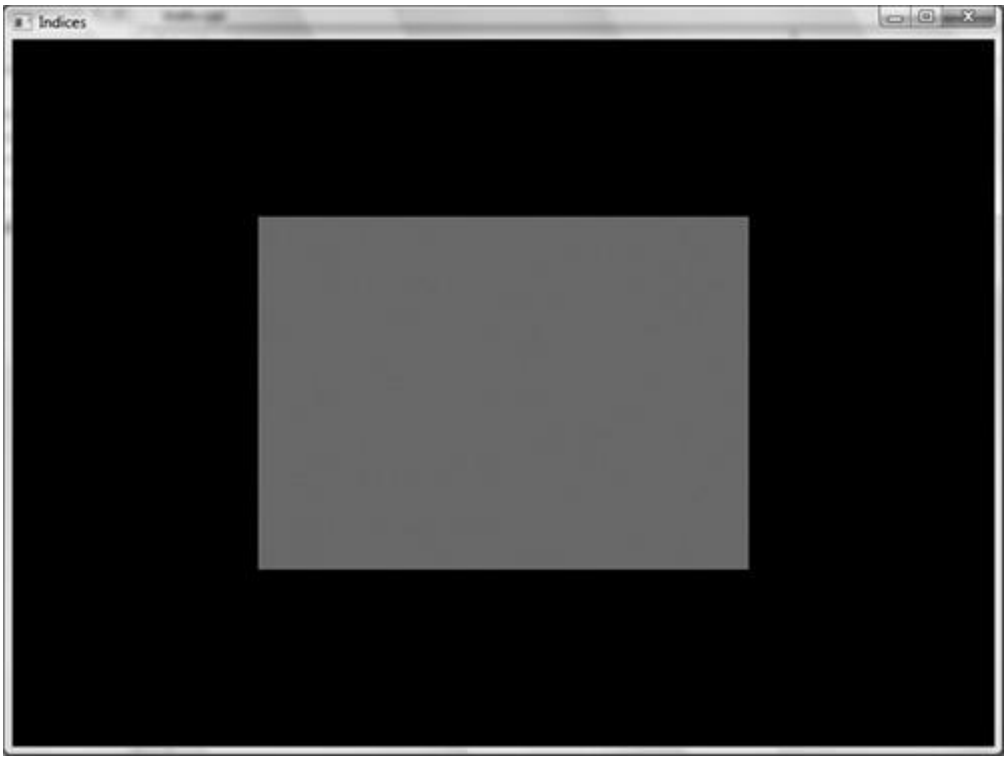
The last step is to render the data. Since we are using indexed geometry, we cannot use the `Draw()` function, which is for nonindexed geometry. To draw indexed geometry, we use `DrawIndexed()`, which takes as parameters the total number of indices, the starting index, and the starting vertex. The drawing method from the Indices demo application is shown in Listing 3.8.

LISTING 3.8. THE DRAWING METHOD FROM THE INDICES DEMO'S RENDERING FUNCTION

```
g_d3dDevice->DrawIndexed(6, 0, 0);
```

The remaining code from the indices demo is the same as the Primitives demo. A screenshot from the Indices demo is shown in Figure 3.11.

FIGURE 3.11. A SCREENSHOT FROM THE INDICES DEMO.



COLORS

The first property, aside from vertex positions, that we will discuss is vertex colors. In Direct3D a structure called D3DXCOLOR is used to represent colors. The D3DXCOLOR structure is an RGBA (which represents the Red, Green, Blue, and Alpha components of a color where Alpha is used for transparency) structure with floating-point members. A value of 0 represents the absence of color, while 1 represents full color. Anything in between is a percentage between the two extremes. A value of 1, or 100%, represents 255 when working with 8-bit color modes. The D3DXCOLOR structure is as follows.

```
typedef struct D3DXCOLOR {
    FLOAT r;
    FLOAT g;
    FLOAT b;
    FLOAT a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

COLORS DEMO

On the book's accompanying CD-ROM is a demo application called Colors in the Chapter 3 folder. This demo application builds off of the Primitives demo and adds a color property to the vertices that are rendered. In the Primitives demo the pixel shader used yellow for the rendering. In this demo the color is specified per vertex. The first change in the Colors demo is the addition of a D3DXCOLOR object to the vertex structure. The addition of colors to the vertex structure is shown in Listing 3.9.

LISTING 3.9. THE GLOBAL SECTION FROM THE COLORS DEMO

```
// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

// Effect objects.
ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_technique = NULL;

// Display object to store scene geometry.
ID3D10InputLayout *g_layout = NULL;
ID3D10Buffer *g_vertexBuffer = NULL;

// Structure used to represent a single vertex.
struct DX10_Vertex
{
    D3DXVECTOR3 pos;
    D3DXCOLOR col;
};
```

Since the vertex structure requires a color now, the data that makes up the object must be updated to reflect this. This can be seen in the initialize function for the Colors demo, where the input layout is given a different property and the list of vertices specifies a color for each vertex point. The initialization code that is specific to the Colors demo is shown in Listing 3.10. Note that the alignment offset (third to last property for each input element) is 12 for the color element. This is because the element that comes before it takes up 12 bytes, so to get to the color data, Direct3D must move past the first 12 bytes of each vertex to obtain the start of the color.

LISTING 3.10. THE INITIALIZE CODE SPECIFIC TO THE COLORS DEMO

```
bool InitializeDemo()
{
   // Load the shader.

   …


   // Create the geometry. The layout of each vertex is made up
   // of just a position so that is all we need.

   D3D10_INPUT_ELEMENT_DESC layout[] =
   {
     { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
     { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
   };


   …


   DX10_Vertex vertices[] =
   {
     { D3DXVECTOR3( 0.0f,  0.5f, 0.5f), D3DXCOLOR(1, 0, 0, 1) },
     { D3DXVECTOR3( 0.5f, -0.5f, 0.5f), D3DXCOLOR(0, 1, 0, 1) },
     { D3DXVECTOR3(-0.5f, -0.5f, 0.5f), D3DXCOLOR(0, 0, 1, 1) },
   };
   // Create the vertex buffer.

   D3D10_BUFFER_DESC buffDesc;

   buffDesc.Usage = D3D10_USAGE_DEFAULT;
   buffDesc.ByteWidth = sizeof(DX10_Vertex) * 3;
   buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
   buffDesc.CPUAccessFlags = 0;
   buffDesc.MiscFlags = 0;

   D3D10_SUBRESOURCE_DATA resData;
   resData.pSysMem = vertices;

   hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
                                  &g_vertexBuffer);
```

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

    return true;
}
```

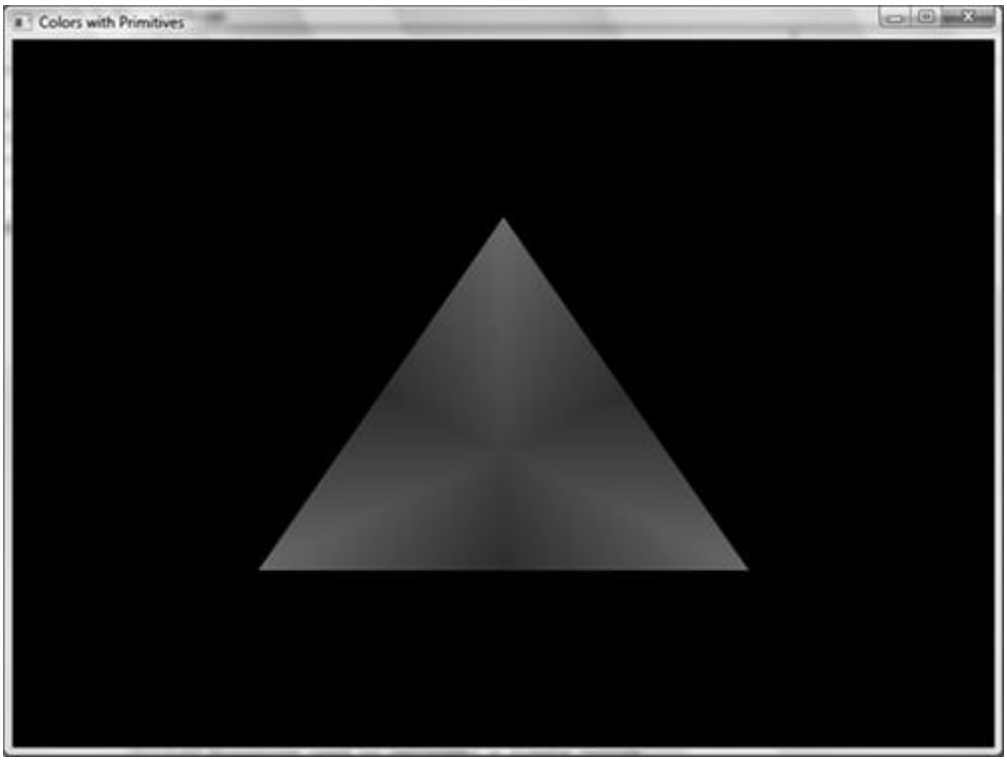
The rendering code from the Colors demo is the same as the Primitives demo, with the exception that we are rendering three vertices instead of six. The shader for the Colors demo is slightly different than the Primitives demo. We have added a color semantic to the vertex and pixel shader input structures. This occurs because now we are bringing in a vertex color from the application into the vertex shader input, and we are sending the color from the vertex shader into the pixel shader input. The pixel shader simply uses this incoming vertex color instead of specifying a hard-coded value like the Primitives demo did for its output color value. The shader from the Colors demo is shown in Listing 3.11. Figure 3.12 shows a screenshot from the Colors demo. To get a better appreciation of the demo in action it is recommended that you execute it.

LISTING 3.11. THE COLORS DEMO'S SHADER

```
struct VS_INPUT
{
  float4 Pos : POSITION;
  float4 Color : COLOR;
};


struct PS_INPUT
{
  float4 Pos : SV_POSITION;
  float4 Color : COLORO;
};


PS_INPUT VS(VS_INPUT input)
{
  PS_INPUT output = (PS_INPUT)0;

  output.Pos = input.Pos;
  output.Color = input.Color;

  return output;
}


float4 PS(PS_INPUT input) : SV_Target
{
  return input.Color;
}
technique10 PassThrough
{
  pass P0
  {
```

```
        SetVertexShader(CompileShader(vs_4_0, VS()));

        SetGeometryShader(NULL);

        SetPixelShader(CompileShader(ps_4_0, PS()));
    }
}
```

FIGURE 3.12. A SCREENSHOT FROM THE COLORS DEMO.



SUMMARY

Rendering primitives is important to all modern video games. In this chapter we discussed the basics of rendering in Direct3D 10. Because Direct3D 10 is shader only, a detailed discussion of programmable shaders is necessary. This discussion comes in the next chapter.

This chapter only covered the basics of rendering. Throughout this book we progress to more complicated and advanced topics. After rendering, we discuss other areas of games such as input and sound before moving on to the final section, the creation of a DirectX 10 video game.

The following elements were discussed in this chapter:

- Primitives
- Points
- Lines

- Triangles
- Indices
- Shaders
- Effects and techniques
- Colors

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. Define a primitive.
2. List three types of primitives that Direct3D supports.
3. List the different types of lines. Describe each one.
4. List the different types of triangles. Describe each one.
5. What is a vertex? What is a vector? How are the two similar?
6. What is an input layout?
7. What is a vertex buffer?
8. What is the input assembler, and how is it used to set up geometry in Direct3D?
9. What are indices, and how are they used in the rendering of geometry?
10. Describe techniques.
11. True or false: A vertex is a point's position.
12. True or false: There are eight bytes in a bit.
13. True or false: Indices are vertex index positions of each primitive's vertex points.
14. True or false: Rasterizer states control how Direct3D is set up.

15. True or false: The alpha channel is often a control for transparency.

16. True or false: HLSL is Direct3D's high level programmable shading language.

17. True or false: Back face culling is the ability to not draw polygons far away from the camera.

18. True or false: The fill mode controls how the surface is shaded.

19. True or false: A technique defined in an HLSL shader is an effect with one or more passes.

20. True or false: *Topology* is a term used to describe the primitive type of geometry.

4. SHADER MODEL 4

In This Chapter

- Shader Model 4
- Shaders in Direct3D 10

Real-time computer graphics have evolved over the generations. Today, game graphics are performed by executing custom-written code directly onto the graphics processing unit. These graphical code programs written for graphics hardware are known as programmable shaders. With programmable shaders just about any visual effect can be created in real time for a host of different applications. This allows developers a level of flexibility that was unseen before the turn of the millennium.

In this chapter we will discuss Direct3D's shading language, known as the High-Level Shading Language (HLSL). Direct3D 10 requires HLSL programmable graphics shaders to be written and used with the application programming interface (API), whereas in Direct3D 9, for example, their use was optional. It is essential that all graphics programmers know how to program the graphics processing unit, and the sooner they are exposed to this the better, since the industry is at a point where experience with and knowledge of shaders are essential. Although in Direct3D 10 newcomers have no choice but to learn shaders, OpenGL and XNA users still have the option to defer this learning until they are ready. XNA does not offer a fixed-function pipeline, but it does offer classes that are part of the framework and are used to implement basic effects without directly using HLSL in applications.

SHADER MODEL 4

Direct3D 10 uses Shader Model 4 and HLSL, which is a high-level language similar to the C programming language but for graphics hardware. In the early days of game graphics, graphical algorithms were implemented in graphics APIs as a series of states that could be enabled or disabled. This series of built-in algorithms and states is known as the fixed-function pipeline, and for many years it was the way hardware graphical effects were performed inside video game scenes.

For example, to use hardware lighting in a scene, a programmer could enable lighting in OpenGL by calling a function such as `glEnable(GL_LIGHTING)` and calling additional functions to set the specific states and properties of the scene's various lights. To use depth testing while rendering a scene, the API's depth-testing feature could be either enabled to activate its use it or disabled to turn it off, for example. Many features of the major graphics APIs worked in this manner.

The fixed-function pipeline hid the underlying implementation from the user. Therefore, anyone using, for example, OpenGL or Direct3D did not need to know the math or processes behind a lighting algorithm or how to program it because it was as simple as calling one or a few API functions. On the plus side, using the fixed-function pipeline was simple and often straightforward, but it did have its negatives, which for professional game developers far out-weighted its pros. Some of these negatives include the following issues.

- Developers had no control over what a graphics API offered and had to either use what was in the API or use clever tricks to attempt to create effects not supported by the tool.
- Developers had no direct control over when new features were added.
- Individuals could not modify the algorithms that make up the fixed-function pipeline.
- Individuals could not extend the graphical effects an API could perform.
- Some graphical effects were either very tricky or impossible to perform with the fixed-function pipeline.
- Having only a fixed-function pipeline limits what the API can do and what it is capable of graphically.
- It was not possible to perform new effects that were hardware accelerated unless the API somehow allowed for the effect to take place.
- Non-graphics-related calculations could not be performed on the GPU using a graphics API that supports only the fixed-function pipeline.

Shader Model 4 and HLSL, will be discussed in this chapter and used often throughout this book.

SHADER MODEL 4 VERSUS PRE-4

When graphics APIs and hardware started to embrace and support programmable shaders, it was originally done by programming directly to the GPU using low-level assembly language. The earlier graphics hardware that supported programmable shaders had very limited instruction counts, registers, texture reads, and so forth. Because low-level shaders used assembly language, they suffered from the same or similar drawbacks as applications created using assembly language. These issues include a few of the following.

- Assembly is harder to read than a high-level language such as C++.
- Assembly is harder to modify than a high-level language such as C++, especially by someone who did not originally write the software.
- Assembly is harder to maintain.
- Writing an application in assembly can take more time due to the number of instructions and registers that are used to obtain a meaningful result.

- It is harder to track down bugs and errors that are not syntax based in graphical shaders written in assembly.

- It is tricky to find ways to reduce the instruction count in an effort to do more within a shader.

NVIDIA's C for graphics (Cg) was the first high-level shading language that was available to developers, and HLSL soon followed. HLSL and Cg are similar because they were both developed by Microsoft (along with NVIDIA for Cg) at the same time. HLSL is to Direct3D what the OpenGL Shading Language (GLSL) is to the OpenGL standard.

A shader model is a version of shading technology; almost each generation of DirectX saw one or two shader models. Direct3D 10 brought forth Shader Model 4, Direct3D 9 saw Shader Model 2 and 3, and Direct3D 8 saw Shader Model 1. Many of the differences between the first three models include instruction and register count increases, while Model 4 completely changed shading technology, especially for the technical side in its implementation.

Shader Model 1 was the first shading version for programmable hardware. It was limited to 128 instructions, and the number of registers often depended on the hardware device because some cards can have more registers than others, meaning that if you write an effect that has to work on multiple pieces of hardware, you will have to write for the lower register count of the target hardware. Shader Model 2 added to the instruction and register count and added some new capabilities such as fetching from a texture multiple times. Shader Model 2 also has some very limited flow control that was improved up in Shader Model 3.

Shader Model 3 again increased the register and instruction counter of version 2 and added many new features (some of which have limitations and are improved upon in version 4) such as dynamic branching, better flow control, floating-point buffers, texture formats, and so forth. Dynamic branching in shaders allows for branching instructions such as conditional statements and loops. Shader Model 3 also allowed for the ability to create floating-point buffers that make it possible to create high dynamic range (HDR) images.

Shader Model 4 is a superset of version 3 but with new capabilities and a unified shader core for each of the shader types. This means each of the shader types has a unified instruction set, whereas previous versions did not. Unlike previous versions of Direct3D and shader models, version 4 for Direct3D can only be implemented using HLSL; previous versions could be implemented in either HLSL or in low-level assembly language. HLSL is compiled down into assembly, though, which can be viewed using the PIX tool from the DirectX software development kit (SDK). That means you can load compiled shaders in Direct3D at run-time or have them compiled at run-time by the application. Shader Model 4 greatly differs from the previous versions and offers the following features.

- Unified shader core (architecture)

- A new type of shader, the geometry shader

- Parallel processing and multi-core GPUs

- Improved version 3 features (e.g., improved dynamic branching, unlimited instruction count, etc.)

- The new constant and texture buffers instead of registers

- No restrictions in the instruction count of shaders

SHADER TYPES

In Shader Model 4 you can create three types of shaders: vertex shaders, pixel shaders (also known as fragment shaders), and geometry shaders. Prior to Shader Model 4 there were only vertex and pixel shaders. Direct3D 10 was the first graphics API to support geometry shaders and Shader Model 4.

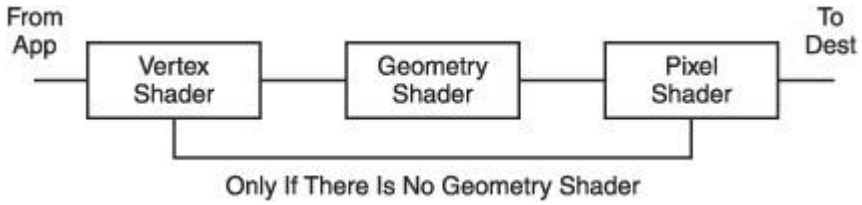
A vertex shader takes the incoming vertex of a surface and executes an algorithm on it to prepare it for later processing stages, including the geometry shader stage and the pixel shader stage. Usually this includes taking a vertex and transforming its position from local (object) space to screen space and calculating other properties that the vertex will need to possess for the later rendering stages. Spaces, transformations, and matrices will be discussed in Chapter 8, "Game Math." A vertex is literally a point that makes up the primitive being rendered. A primitive is a small piece of a larger model such as a single triangle. A vertex can have many properties, but the one property that is always required is a position. In the vertex shader this position is transformed between different spaces, and other calculations can be performed based on what the pixel or geometry shader needs to produce whatever graphical effects you are attempting to create. For example, the vertex shader can calculate the light vector that the pixel shader uses in diffuse lighting. Vectors will also be discussed in Chapter 8, while lighting will be discussed in Chapter 13, "Lighting."

A geometry shader is an optional shader that can be created and used. A geometry shader takes an incoming primitive, such as a triangle, and executes some algorithm on it. You might use the geometry shader to take an incoming triangle and split it up into smaller pieces to increase its geometric detail or to perform some other action that you need to occur on the primitive level. The geometry shader sits between the vertex shader and the pixel shader stages, which means the incoming geometry to the geometry shader is the transformed geometric output from the vertex shader, and its output goes to the pixel shader. The output from the geometry shader is zero or more primitives. This means the geometry shader can output no geometry or the incoming primitive it was given or even generate one or more pieces of new geometry and output that.

The pixel shader is responsible for shading the pixels that make up the surface of the primitive using the algorithm of your choice. A pixel shader can texture map the surface (i.e., place an image across the surface), shade the surface to a specified color, light the surface, bump map a surface, and much more. Pixel shaders output color values, which are stored on the screen for all visible and affected pixels. The pixel shader comes after the vertex shader and receives as input the vertex shader's output only if a geometry shader does not exist. If the geometry shader does exist, then that is the stage from which the pixel shader receives its input.

The three types of shader stages are shown in Figure 4.1.

FIGURE 4.1. VERTEX, GEOMETRY, AND PIXEL SHADER STAGES.



INTRINSIC FUNCTIONS

HLSL has a list of intrinsic functions available to use in effect shaders. Many of these functions perform mathematical operations, while a few of them are used for texture mapping. Some of these topics have yet to be discussed, such as texture mapping, vectors, and matrices, and will be discussed in later chapters in this book.

For now, use Table 4.1, which lists the intrinsic HLSL functions, as a reference and remember that each function that deals with a topic that is yet to be discussed will be reviewed in subsequent chapters.

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
`ret acos(x)`	Finds the arccosine of the input parameter. Takes as input a float, vector, or matrix with floating-point components and returns the same type as the input type.
`ret all(x)`	Returns true or false if all components of the input parameter are nonzero. Takes as input a float, vector, or matrix and returns a Boolean.
`ret any(x)`	Same as `ret all(x)` but returns true if any of the input components are nonzero.
`ret asfloat(x)`	Converts the input components to a floating-point value. This assumes the input components are integer-based.
`ret asin(x)`	Returns the arcsine value of the input parameter. Takes as input a float, vector, or matrix.
`ret asint(x)`	Takes the input parameter and converts its components to an integer. Assumes the components are originally a float or an unsigned integer.
`ret asuint(x)`	The same as `ret asint(x)` but converts components to an unsigned integer.
`ret atan(x)`	Returns the arctangent of the components of the input parameter. Takes as input a float, vector, or matrix.
`ret atan2(x, y)`	Same as `ret atan(x)` but returns the arctangent of (x, y).

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
`ret ceil(x)`	Returns the smallest integer value of the input parameter. For example, if the input is 0.3, then 0 is returned.
`ret clamp(x, min, max)`	Clamps the input x to the minimum and maximum range specified by the last two parameters.
`clip(x)`	Discards the current pixel if the value for this function's input parameter is less than 0. Used by the pixel shader only.
`ret cos(x)`	Returns the cosine value of the input parameter.
`ret cosh(x)`	Returns the hyperbolic cosine of the input parameter.
`ret cross(x, y)`	Returns the cross product of the two input parameters. The parameters must be vectors.
`ret D3DCOLORtoUBYTE4(x)`	Converts a `D3DCOLOR` parameter to a `UBYTE4`. `UBYTE4` has an integer component format.
`ret ddx(x)`	Returns the partial derivative with respect to the input parameter, which is the X axis of the screen coordinate.
`ret ddy(x)`	Same as `ret ddx(x)` but with respect to the Y axis of the screen coordinate.
`ret degrees(x)`	Converts the input parameter from radians to degrees, assuming the input is in radians.
`ret determinant(m)`	Returns the determinant of the input matrix parameter. The input type must be a matrix.
`ret distance(x, y)`	Returns the distance between two vectors specified in the x and y parameters. Their types must be a vector.
`ret dot(x, y)`	Returns the dot product between two input vectors specified in the x and y parameters. Their type must be a vector.

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
`ret exp(x)`	Returns the base-e exponential value of the input floating-point parameters (x).
`ret exp2(x)`	Same as `ret exp(x)` but returns the base-2 exponential value.
`ret faceforward(n, i, ng)`	Flips a surface normal (ng) to face the direction opposite specified by i. Returns the result in n. All parameters are vectors.
`ret floor(x)`	Same as `ret ceil(x)` but returns the largest integer component of the input parameter.
`ret fmod(x, y)`	Returns the floating-point remainder of x/y.
`ret frac(x)`	Returns the fractional (value after the decimal point) of the input parameter. The value is a floating-point number between 0 and 1.
`ret frexp(x, exp)`	Returns the mantissa and exponent of the x input parameter and stores the result in exp.
`ret fwidth(x)`	Returns the absolute value of the partial derivatives of the X screen coordinate. In other words abs(ddx(x)) + abs(ddy(x)). This function is only supported in the pixel shader.
`ret GetRenderTarget SamplePosition (index)`	Returns the (X, Y) location of the render target that is being sampled. The target is specified by the index parameter.
`UINT GetRenderTarget SampleCount()`	Returns the number of samples for a render target.
`ret isinfinite(x)`	Returns true or false if the floating-point parameter is infinite.
`ret isinf(x)`	The same as `ret isinfinite(x)` but returns true if the parameter is `+INF` or `?INF`. Returns false for all

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
	other values.
`ret isnan(x)`	Returns true if the parameter equals NAN or QNAN.
`ret ldexp(x, exp)`	Returns the x parameter multiplied by two and raised to the exponent specified by exp.
`ret length(x)`	Returns the length of the specified parameter, which has to be a vector. The length of a vector is also known as its magnitude.
`ret lerp(x, y, s)`	Performs linear interpolation between x and y using s, where s is between 0 and 1. The component types for the input parameters (which can be vectors, matrices, or floats) must be floating point.
`ret lit(n_dot_l, n_dot_h, m)`	Returns the lighting coefficient vector, where the ambient light is assumed to be 1, n_dot_l is the diffuse value, `n_dot_h` is the specular value, and m is the specular exponent.
`ret log(x)`	Returns the base-e logarithm of the input parameter.
`ret log2(x)`	Same as `ret log(x)` but returns the base-2 logarithm.
`ret log10(x)`	Same as `ret log2(x)` but returns the base-10 logarithm.
`ret max(x, y)`	Returns x if x is greater than y, or else it returns y.
`ret min(x, y)`	Same as `ret max(x, y)` but returns the parameter that is the smaller of the two.
`ret modf(x, ip)`	Spits a floating-point value specified by the x parameter into its integer and fractional parts. The integer part is returned to ip, and the fractional part is returned by the function.

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
`ret mul(x, y)`	Multiplies two matrices together and returns the result. The matrices must have the same number of rows and columns.
`ret noise(x)`	Returns a random value between ?1 and 1 based on x using the perlin noise algorithm.
`ret normalize(x)`	Returns the normalized vector of the input parameter x. This essentially equates to x/length(x).
`ret pow(x, y)`	Returns the value specified by x to the y power.
`ret radians(x)`	Converts the parameter x from degrees to radians.
`ret reflect(i, n)`	Returns the reflection vector based on the ray's direction (i) and the surface normal (n).
`ret refract(i, n, index)`	Returns the refraction vector based on the ray's direction (i), the surface normal (n), and the index of refraction (index).
`ret round(x)`	Rounds the value in x to the nearest integer.
`ret rsqrt(x)`	Returns the reciprocal of the square root of x. In other words 1/sqrt(x).
`ret saturate(x)`	Clamps the value of x to 0 and 1.
`ret sign(x)`	Returns ?1 if x is less than zero, 0 if x is zero, and 1 if x is greater than zero.
`ret sin(x)`	Returns the sine value of x.
`ret sincos(x, out s, out c)`	Returns the sine value of x in the s parameter and the cosine value of x in the c parameter.
`ret sinh(x)`	Returns the hyperbolic value of x.
`ret smoothstep(min,`	Returns the Hermite interpolation between min and max

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
`max, x)`	using x as a scalar. If x < min, then min is returned, else if x > max, then max is returned, or else some value between min and max.
`ret sqrt(x)`	Returns the square root of x.
`ret step(y, x)`	Compares the two parameters and returns 0 or 1 depending on which parameter is greater.
`ret tan(x)`	Returns the tangent of x.
`ret tanh(x)`	Returns the hyperbolic tangent of x.

`ret tex1D(s, t)`
`ret tex1Dbias(s, t)`
`ret tex1Dlod(s, t)`
`ret tex1Dgrad(s, t, ddx, ddy)`
`ret tex1Dproj(s, t)`
	Samples a 1D texture(s) using texture coordinates (t) and returns a color.

`ret tex2D(s, t)`
`ret tex2Dbias(s, t)`
`ret tex2Dlod(s, t)`
`ret tex2Dgrad(s, t, ddx, ddy)`
`ret tex2Dproj(s, t)`
	Samples a 2D texture(s) using texture coordinates (t) and returns a color.

`ret tex3D(s, t)`
`ret tex3Dbias(s, t)`
`ret tex3Dlod(s, t)`
`ret tex3Dgrad(s, t, ddx, ddy)`
`ret tex3Dproj(s, t)`
	Samples a 3D texture(s) using texture coordinates (t) and returns a color.

`ret texCUBE(s, t)`
`ret texCUBEbias(s, t)`
`ret texCUBElod(s, t)`
`ret texCUBEgrad(s, t, ddx, ddy)`
`ret texCUBEproj(s, t)`
	Samples a cube texture(s) using texture coordinates (t) and returns a color.

`ret transpose(x)`	Finds the transpose of the input parameter x, which has to be a matrix.

TABLE 4.1. HLSL INTRINSIC FUNCTIONS.

`ret abs(x)`	Finds the absolute value of the input parameter. Takes as input a float, vector, or matrix and returns those same types.
`ret trunc(x)`	Truncates a floating-point value to an integer.

SHADER CONSTANTS AND TEXTURE BUFFERS

In Shader Model 4 there are what are known as constant and texture buffers. Buffers are a way of optimizing data variables that are used by the shader program. A constant buffer is used to store constant variables, and a texture buffer is more suitable for texture images. By placing variables in buffers based on their usage and CPU access, Direct3D 10 can optimize how they are stored to improve performance and shader efficiency.

Creating a buffer is similar to creating a C structure and can take the following form:

```
BufferType[name] : register(b#)
{
   VariableDeclaration[packetoffset(c#.xyzw)];
}
```

The `BufferType` refers to `cbuffer` for constant buffer and `tbuffer` for texture buffer. The name refers to an optional unique name that you can give the buffer. The register is an optional piece of information used to manually pack constant data into specific register indexes. `VariableDeclaration` is a normal variable declaration in the shader file, and `packetoffset` is an optional piece of information used to manually pack constant data into specific registers. An example of a constant buffer minus the optional information is as follows.

```
cbuffer cbChangesEveryFrame
{
    float4 color;
};
```

In the example above `cbuffer` declares a constant buffer, `cbChangesEveryFrame` tells Direct3D that the variables in the constant buffer will be changed for each rendering frame (or set for each frame but not necessarily changed in value per se), and the variable inside the constant buffer is bound to that constant buffer. An example of using constant buffers in Direct3D 10 will be shown later in this chapter. Other types of usage types include `cbChangeOnResize` and `cbNeverChanges`.

VARIABLES AND DATA TYPES

HLSL offers a number of data types for variables. Some of these are the same as the C programming language and include the following scalar types.

- `bool`

- `int`
- `uint`
- `half`
- `float`
- `double`

There is also a string type in HLSL for storing strings of ASCII characters. HLSL also adds an additional data type known as `Buffer`, which is shown in the following example.

```
Buffer<float4> g_buffer;
float4 elementOne = g_buffer.Load(1);
```

A `Buffer` object is treated like an array in which you can store information in the object and read information from it using the `Load()` function. The index you send to the `Load()` function determines the array index element that is returned.

Other additional data types offered by HLSL include vector types, matrix types, sampler types, and texture types, and also structures. A structure in HLSL can be created in the same way a structure is created in C. For texture types they include the following.

- `texture` (which is untyped for backward compatibility)
- `Texture1D`
- `Texture1DArray`
- `Texture2D`
- `Texture2DArray`
- `Texture3D`
- `TextureCube`

Textures will be discussed in more detail in Chapter 6, "Shading and Surfaces," and Chapter 7, "Additional Texture Mapping," along with the sampler types. The vector types include between one and four component vectors specified by either the vector keyword followed by the number of components or by the data type followed by the number of components. Therefore, a vector can have float components, integer components, and any of the other scalar types. Examples of declaring three- and four-component vectors are as follows:

```
int3 vec3;                      // 3-component vector
float4 vec1;                    // 4-component vector
vector<float, 4> vec2;          // 4-component vector
```

The components of a vector can be accessed in the same way the member variables of a C structure can be accessed: by using the dot (.) operator followed by the name of the

component. Vectors in HLSL can have an X for 1D vectors; X and Y for 2D vectors; X, Y, and Z for 3D vectors; and X, Y, Z, and W for 4D vectors.

The matrix types are similar to the vector types, where a matrix is an object made up of rows and columns. You can have 3 × 3, 4 × 4, 3 × 4, and so on matrices in HLSL. Think of a matrix as a 2D table in which elements are accessed by row and column. Matrices can be created using any of the scalar types like vectors but they are declared using the keyword matrix or by defining two numbers for the rows and columns following the data type. An example of creating 3 × 4 and 4 × 4 matrices is as follows.

```
float3×4 matrix1;
float4×4 matrix2;
matrix<float, 4, 4> matrix3;
```

Vectors and matrices will be discussed in Chapter 8 in more detail.

Variable syntax includes specifying a storage class, data type, variable name, and semantic. The data types we've already discussed include the scalar types, vectors, and matrices. The storage class by default for global variables is uniform, so you don't have to actually use the uniform keyword before declaring variables in the global scope. Other storage classes include:

- `extern`, which cannot be combined with static and is used to mark a variable as being available for input externally (by the application) and is the default for global variables
- `static`, which is the same as it is for C/C++ and is used to keep the variable in scope even once the function it is in loses scope (cannot be used for globals)
- `nointerpolation`, which means to not interpolate the outputs used by the pixel shader
- `shared`, which marks a variable for being shared by different effects
- `uniform`, which is another default type that marks a variable's data as constant throughout the shader's execution
- `volatile`, which is used to mark a variable as being changed frequently

The semantic of a variable is an optional piece of information used to bind shader inputs and outputs by the HLSL compiler. This means we can use the semantic to bind variables to vertex attributes such as the position, color, direction (normal), and so on of a vertex point. Semantics are used for variables on the global scope only. Semantics of variables not part of the global scope are ignored. An example of a semantic follows, where the `pos` variable is given the semantic `POSITION`, which can be used by the application to store the incoming vertex positions in that variable as geometry is rendered.

```
float4 pos : POSITION
```

You can name a semantic whatever you like. Special reversed semantics are built into HLSL such as `SV_POSITION`. The `SV` at the beginning of the reserved semantics stands for system value. Other system-value semantics include the following.

- `SV_ClipDistance[n]:` Used for clip distance data
- `SV_CullDistance[n]:` Used for cull distance data
- `SV_Coverage:` The output coverage mask
- `SV_Depth:` The pixel's depth stored in the depth buffer
- `SV_IsFrontFace:` A flag indicating that the primitive is visible
- `SV_Position:` The vertex position
- `SV_RenderTargetArrayIndex:` The render target's array index that is currently being rendered to
- `SV_SampleIndex:` The sample frequency index data
- `SV_Target[n]:` A render target array (`SV_RenderTargetArrayIndex` is its index)
- `SV_ViewportArrayIndex:` The index to the current camera view-port being used
- `SV_InstanceID:` The per-instance identifier
- `SV_PrimitiveID:` The per-primitive identifier
- `SV_VertexID:` The per-vertex identifier

Semantics will be discussed more when we look at the first shader example for this chapter. Additional information that can be paired with a variable is an annotation that is used by the effects framework and ignored by HLSL, an initial value that simply sets a value to the variable at its declaration, a constant register index, and a packet offset.

FLOW CONTROL AND FUNCTIONS

Functions can be created in HLSL just like they can in C/C++. You can create whatever functions you want, and in the HLSL technique, you specify the main entry points for the vertex, geometry, and pixel shaders. This will be shown later during the discussion of techniques. The flow control statements that are supported by HLSL include the following:

- `if`
- `break`
- `continue`
- `discard`
- `do`

- `while`
- `for`
- `stop`
- `switch`

The `discard` statement is used to instruct the pixel shader to not output the results and to discard them. The `stop` statement is used to stop the execution of the current statement and return its output immediately.

SHADERS IN DIRECT3D 10

In the following sections we will examine three simple Direct3D 10 HLSL shaders: the Shader Example demo, the Constant Buffer demo, and the Uniform Variables demo. The Shader Example demo will demonstrate how to create a straightforward vertex, pixel, and geometry shader that is used to render geometry using a solid color. The Uniform Variables demo will demonstrate how to set the color of the geometry that is being rendered by using uniform variables. The Constant Buffer demo will demonstrate how to use constant buffers with uniform variables in HLSL. Each of these demos can be found on this book's accompanying CD-ROM in the Chapter 4 folder.

SHADER EXAMPLE DEMO

The goal of the Shader Example demo is to render a triangle using a color that is specified by the pixel shader. The demo will also create the most basic geometry shader that can be created, which will take the incoming geometry and pass it along to the pixel shader without doing anything to it.

The input for the vertex shader expects a single position for each vertex of the geometry that is being rendered. The geometry shader takes as input the output of the vertex shader, which also happens to be the position. In other words, the vertex shader passes along the position directly to the geometry shader. The pixel shader then shades any pixel between the vertex points with a purple color, which is created by using full red and full blue for the return color.

The body of the vertex shader takes the input and passes it directly to the output for the geometry shader. The pixel shader's output is a four-component color. The input from the geometry shader isn't used because the pixel shader does not need to calculate anything specific for the vertex attribute. On the other hand, algorithms such as lighting and other techniques that will be performed throughout this book need information from either the vertex or geometry shaders.

For the geometry shader, the function looks slightly different than for the vertex and pixel shaders. Before the start of the function, the geometry shader requires the `maxvertexcount` keyword followed by a number to tell HLSL the vertex count of what the shader will be writing out. Therefore, if we are outputting a single triangle, this value will be 3. The declaration of the geometry shader function starts with the primitive type it is to expect followed by the input geometry (which is the output from the vertex shader on the primitive level) and a stream object. The primitive type can be one of the following keywords, where `adj` stands for adjacent or index geometry (discussed in Chapter 8):

- point
- line
- triangle
- lineadj
- triangleadj

The triangle stream object is used to write out vertices that will be used by the pixel shader. The number used for the `maxvertexcount` should be the largest number of vertices that is outputted. A call to the stream object's function `Append()` is used to write a vertex. When declaring the triangle stream, the type the stream will write needs to be specified between angle brackets (`< >`). The HLSL shader from the Shader Example demo is shown in Listing 4.1. The inputs for each shader are represented by structures that store the various variables that make up the input, along with their semantics.

LISTING 4.1. THE SHADER EXAMPLE DEMO'S HLSL SHADER FILE

```
struct VS_INPUT
{
   float4 Pos : POSITION;
};

struct GS_INPUT
{
   float4 Pos : SV_POSITION;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
};


// VERTEX SHADER

GS_INPUT VS(VS_INPUT input)
{
   GS_INPUT output = (GS_INPUT)0;

   output.Pos = input.Pos;

   return output;
}
// GEOMETRY SHADER

[maxvertexcount(3)]
void GS(triangle GS_INPUT input[3],
        inout TriangleStream<PS_INPUT> triStream)
{
   PS_INPUT output = (PS_INPUT)0;
```

```
    output.Pos = input[0].Pos;
    triStream.Append(input[0]);

    output.Pos = input[1].Pos;
    triStream.Append(input[1]);

    output.Pos = input[2].Pos;
    triStream.Append(input[2]);

    triStream.RestartStrip();
}


// PIXEL SHADER

float4 PS(PS_INPUT input) : SV_Target
{
    return float4(1.0f, 0.0f, 1.0f, 1.0f);
}


technique10 PassThroughShader
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_4_0, VS()));
        SetGeometryShader(CompileShader(gs_4_0, GS()));
        SetPixelShader(CompileShader(ps_4_0, PS()));
    }
}
```

The application uses the `POSITION` semantic to bind the vertex position to the `Pos` variable of the vertex shader input on the application side, and `SV_POSITION` is the reserved output position semantic used by both the geometry and vertex shaders. In the geometry shader the function `RestartStrip()` of the triangle stream is used to mark the beginning of a new primitive. For a three-point triangle, this function should be called after each complete triangle (or after every three vertices) has been appended to the output stream.

At the end of the shader is a technique declaration. A technique in HLSL is used to specify an effect, and an effect file can have more than one technique inside of it. The keyword `technique10` is used for Direct3D 10 techniques and is followed by the name of the technique. This name can be anything you want it to be. Inside the technique you can specify one or more passes starting with P0 (and going to P1, P2, etc.). In each pass you must specify the name of the functions that mark the entry point for each shader type. If a geometry shader is not used, specify `NULL`.

The `CompileShader()` HLSL function compiles the functions that mark the entry point into the appropriate shader type, and `SetVertexShader()`, `SetGeometryShader`, and `SetPixelShader()` set the output of the `CompileShader()` function to the appropriate shader type. The `CompileShader()` function takes the shader version flag and entry function as parameters.

On the application side we need objects that represent the shader in an `ID3D10Effect` object, one that is used to bind to the effect technique that will be used in an object of the type `ID3D10EffectTechnqiue`, and an input layout in an object of type `ID3D10InputLayout`. The input layout is used to access the input data for the input assembler stage. This will be discussed later in this chapter. The global variables from the Shader Example demo are shown in Listing 4.2.

LISTING 4.2. GLOBALS FROM THE SHADER EXAMPLE DEMO

```
#include<d3d10.h>
#include<d3dx10.h>
#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Shader Example"
#define WINDOW_CLASS    "UPGCLASS"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600


// Global window handles.
HINSTANCE g_hInst = NULL;
HWND g_hwnd = NULL;


// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_passThroughTech = NULL;
ID3D10InputLayout *g_shaderInputLayout = NULL;
ID3D10Buffer *g_triVB = NULL;
```

The `InitializeDemo()` function starts by loading the shader file and then making sure it does not contain any errors. If any exist, they are displayed in a message box so the user knows what is wrong and they can be fixed afterwards. Errors are stored in an object of the type `ID3D10Blob`, and calling its `GetBufferPointer()` function allows for the retrieval of the text describing the errors in the shader file.

The technique `PassThroughShader` from the effect file is obtained, and the input layout is created after the successful loading of the effect file. Since the vertex shader requires a vertex with just a position, the input layout matches that and binds to the `POSITION` semantic that was seen in the vertex shader's input structure. The remainder of the function loads the triangle geometry and prepares it for rendering. The `InitializeDemo()` function is shown in Listing 4.3.

LISTING 4.3. THE SHADER EXAMPLE DEMO'S `INITIALIZEDEMO()` FUNCTION

```
bool InitializeDemo()
{
   // Load the shader.
```

```
   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   ID3D10Blob *errors = NULL;

   HRESULT hr = D3DX10CreateEffectFromFile("shader.fx", NULL,
NULL,
                                           "fx_4_0", shaderFlags,
0,
                                           g_d3dDevice, NULL,
NULL,
                                           &g_shader, &errors,
NULL);

   if(errors != NULL)
   {
      MessageBox(NULL, (LPCSTR)errors->GetBufferPointer(),
                 "Error in Shader!", MB_OK);

      errors->Release();
   }

   if(FAILED(hr))
      return false;

   g_passThroughTech = g_shader->GetTechniqueByName(
      "PassThroughShader");
    // Create the geometry.

    D3D10_INPUT_ELEMENT_DESC layout[] =
    {
       { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
          D3D10_INPUT_PER_VERTEX_DATA, 0 },
    };

    unsigned int numElements = sizeof(layout) /
sizeof(layout[0]);
    D3D10_PASS_DESC passDesc;

    g_passThroughTech->GetPassByIndex(0)->GetDesc(&passDesc);

    hr = g_d3dDevice->CreateInputLayout(layout, numElements,

passDesc.pIAInputSignature,

passDesc.IAInputSignatureSize,
                                       &g_shaderInputLayout);
```

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

    D3DXVECTOR3 vertices[] =
    {
       D3DXVECTOR3( 0.0f, 0.5f, 0.5f),
       D3DXVECTOR3( 0.5f, -0.5f, 0.5f),
       D3DXVECTOR3(-0.5f, -0.5f, 0.5f),
    };


    // Create the vertex buffer.

    D3D10_BUFFER_DESC buffDesc;
    buffDesc.Usage = D3D10_USAGE_DEFAULT;
    buffDesc.ByteWidth = sizeof(D3DXVECTOR3) * 3;
    buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
    buffDesc.CPUAccessFlags = 0;
    buffDesc.MiscFlags = 0;

    D3D10_SUBRESOURCE_DATA resData;
    resData.pSysMem = vertices;

    hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
&g_triVB);

    if(FAILED(hr))
       return false;

    return true;
}
```

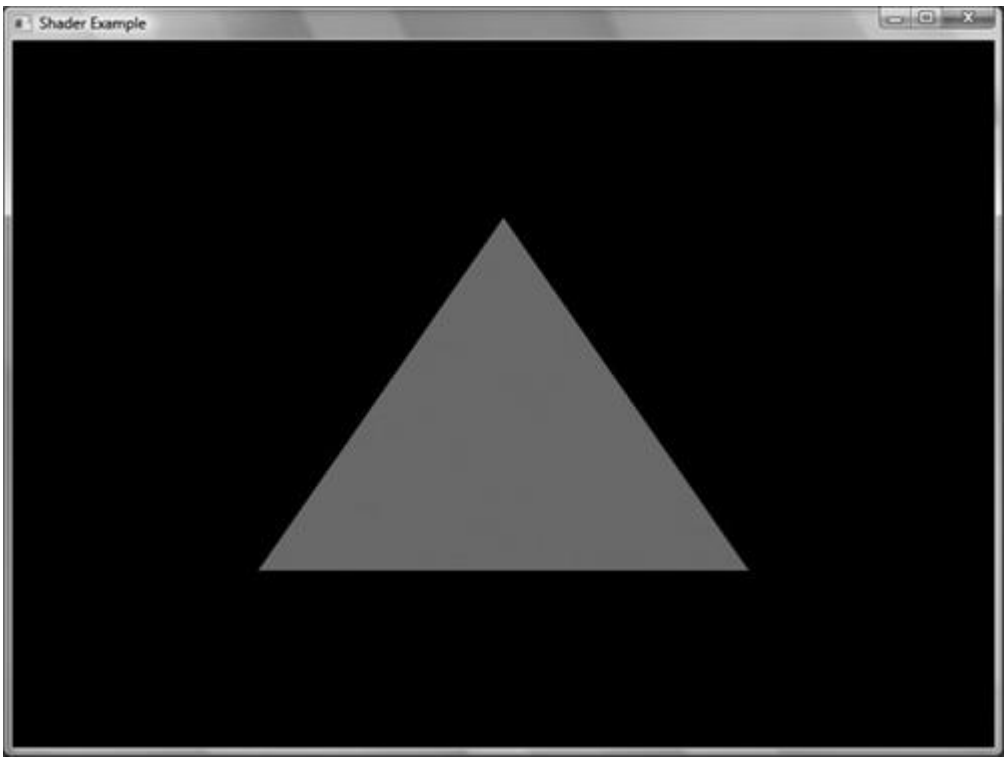
The last two functions in the Shader Example demo are the `RenderScene()` and `Shutdown()` functions. The rendering function starts by clearing the rendering target to black, applies the input layout, applies the vertex buffer, and applies the primitive type (primitive topology) to the input assembler. The function then gets the pass description from the techniques from the shader and uses this to loop through the passes specified by the effect file. For each pass specified the geometry is rendered using the `Draw()` function of the `Direct3D10` device. The rendering function is shown in Listing 4.4. The `Shutdown()` function releases each of the Direct3D 10 objects that were created when the application shut down and is also shown in Listing 4.4. A screenshot from the demo is shown in Figure 4.2.

LISTING 4.4. SHADER EXAMPLE DEMO'S `RENDERSCENE()` AND `SHUTDOWN()` FUNCTIONS

```
void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);
```

```
   unsigned int stride = sizeof(D3DXVECTOR3);
   unsigned int offset = 0;

   g_d3dDevice->IASetInputLayout(g_shaderInputLayout);

   g_d3dDevice->IASetVertexBuffers(0, 1, &g_triVB,
                                   &stride, &offset);

   g_d3dDevice->IASetPrimitiveTopology(
      D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   D3D10_TECHNIQUE_DESC techDesc;
   g_passThroughTech->GetDesc(&techDesc);

   for(unsigned int i = 0; i < techDesc.Passes; i++)
   {
      g_passThroughTech->GetPassByIndex(i)->Apply(0);
      g_d3dDevice->Draw(3, 0);
   }

   g_swapChain->Present(0, 0);
}


void Shutdown()
{
   if(g_d3dDevice) g_d3dDevice->ClearState();
   if(g_swapChain) g_swapChain->Release();
   if(g_renderTargetView) g_renderTargetView->Release();

   if(g_shader) g_shader->Release();
   if(g_shaderInputLayout) g_shaderInputLayout->Release();
   if(g_triVB) g_triVB->Release();

   if(g_d3dDevice) g_d3dDevice->Release();
}
```

FIGURE 4.2. SCREENSHOT FROM THE SHADER EXAMPLE DEMO.



UNIFORM VARIABLES SHADER DEMO

The Uniform Variables Shader demo builds off of the last demo and adds the ability to allow the color of the rendered geometry to be set through a uniform global variable. Because the default type for global variables is `extern` uniform, it is only necessary to specify the data type and its name. The Uniform Variables Shader demo's shader has the new global variable added to the top of the file, and the variable is used by the pixel shader to output the color, which is shown in Listing 4.5. The output of the pixel shader is a `float4`, which is essentially a four-component vector that can be used to store the red, green, blue, and alpha values of a color just like it can be used to store the position of a vector and so on.

LISTING 4.5. THE UNIFORM VARIABLES SHADER DEMO'S SHADER FILE

```
float4 color;


struct VS_INPUT
{
   float4 Pos : POSITION;
};

struct GS_INPUT
{
   float4 Pos : SV_POSITION;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
};
```

```
GS_INPUT VS(VS_INPUT input)
{
   GS_INPUT output = (GS_INPUT)0;
   output.Pos = input.Pos;

   return output;
}

[maxvertexcount(3)]
void GS(triangle GS_INPUT input[3],
        inout TriangleStream<PS_INPUT> triStream)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = input[0].Pos;
   triStream.Append(input[0]);

   output.Pos = input[1].Pos;
   triStream.Append(input[1]);

   output.Pos = input[2].Pos;
   triStream.Append(input[2]);
   triStream.RestartStrip();
}


float4 PS(PS_INPUT input) : SV_Target
{
   return color;
}


technique10 PassThroughShader
{
   pass P0
   {
      SetVertexShader(CompileShader(vs_4_0, VS()));
      SetGeometryShader(CompileShader(gs_4_0, GS()));
      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

To bind to the shader's new global variable, we use an object of type `ID3D10EffectVectorVariable` on the application side. This variable is used to bind to the shader's `extern` uniform variable and allows us to set its value from the application side before the shader is used to render any geometry. The data type used in the shader dictates the object type used by the application. So for a vector we will use `ID3D10EffectVectorVariable`, for a matrix we will use `ID3D10EffectMatrixVariable`, and so forth. The shader-related globals are shown in Listing 4.6. A list of effect variable interfaces is as follows:

- `ID3D10EffectVectorVariable`

- ID3D10EffectMatrixVariable
- ID3D10EffectBlendVariable
- ID3D10EffectDepthStencilVariable
- ID3D10EffectRasterizerVariable
- ID3D10EffectRenderTargetViewVariable
- ID3D10EffectSamplerVariable
- ID3D10EffectScalarVariable
- ID3D10EffectShaderResourceVariable
- ID3D10EffectShaderVariable
- ID3D10EffectStringVariable

The various effect variables such as `ID3D10EffectMatrixVariable` *and so forth will be discussed as they are encountered throughout this book. Each effect variable is used to access the corresponding variable in the shader's effect file.*

LISTING 4.6. THE SHADER-RELATED GLOBALS FROM THE UNIFORM VARIABLES SHADER DEMO

```
ID3D10Effect *g_shader = NULL;
ID3D10EffectVectorVariable *g_colorEffectVar = NULL;
ID3D10EffectTechnique *g_passThroughTech = NULL;
ID3D10InputLayout *g_shaderInputLayout = NULL;
```

To bind to the effect variable, we call the effect object's (of type `ID3D10Effect`) `GetVariableByName()` function. This function will return access to any variable in the shader that the application can access. The object that is returned is of type `ID3D10EffectVariable`, the base type for all effect variables. So, we call the base object's `AsVector()` function to return access to our `ID3D10EffectVectorVariable` object. Aside from getting a variable by name, we can also get it by index or by semantic. The modified and shader-specific code in the `InitializeDemo()` function for the Uniform Variables Shader demo is shown in Listing 4.7. The base (`ID3D10EffectVariable`) object that is returned has `As*()` functions for each of the effect variable types that was listed earlier in this section.

LISTING 4.7. MODIFIED `INITIALIZEDEMO()` FROM THE UNIFORM VARIABLES SHADER DEMO

```
bool InitializeDemo()
{
```

```
   // Load the shader.

   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   ID3D10Blob *errors = NULL;

   HRESULT hr = D3DX10CreateEffectFromFile("shader.fx", NULL,
NULL,
                                           "fx_4_0", shaderFlags,
0,
                                           g_d3dDevice, NULL,
NULL,
                                           &g_shader, &errors,
NULL);

   if(errors != NULL)
   {
      MessageBox(NULL, (LPCSTR)errors->GetBufferPointer(),
                 "Error in Shader!", MB_OK);

      errors->Release();
   }

   if(FAILED(hr))
      return false;

   g_passThroughTech = g_shader->GetTechniqueByName(
      "PassThroughShader");

   g_colorEffectVar = g_shader->GetVariableByName(
      "color")->AsVector();


   …
}
```

The last modified function from the Uniform Variables Shader demo is the `RenderScene()` function. In this function the only new code is the code that sets the uniform variable in the shader. This is done by calling the `ID3D10EffectVectorVariable` object's `SetFloatVector()`, which takes as a parameter a four-component floating-point array that has the values that are to be set to the shader's variable. In this demo the values that are used will set the color to red, where the first array element is the red (set to the max of 1.0), the second is the green, the third is the blue, and the last is the alpha component. The `RenderScene()` function is shown in Listing 4.8.

LISTING 4.8. THE UNIFORM VARIABLES SHADER DEMO'S `RENDERSCENE()` FUNCTION

```
void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

   unsigned int stride = sizeof(D3DXVECTOR3);
   unsigned int offset = 0;

   g_d3dDevice->IASetInputLayout(g_shaderInputLayout);

   g_d3dDevice->IASetVertexBuffers(0, 1, &g_triVB,
                                   &stride, &offset);

   g_d3dDevice->IASetPrimitiveTopology(
      D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   D3D10_TECHNIQUE_DESC techDesc;
   g_passThroughTech->GetDesc(&techDesc);

   float redCol[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
   g_colorEffectVar->SetFloatVector(redCol);

   for(unsigned int i = 0; i < techDesc.Passes; i++)
   {
      g_passThroughTech->GetPassByIndex(i)->Apply(0);
      g_d3dDevice->Draw(3, 0);
   }

   g_swapChain->Present(0, 0);
}
```

Each effect variable has its own set of functions for setting the type of variable in question. Using the vector variable, for example, there are the following get (retrieves the variable's value) and set (sets the variable's value) functions:

- `GetFloatVector()`
- `GetFloatVectorArray()`
- `SetFloatVector()`
- `SetFloatVectorArray()`
- `GetIntVector()`
- `GetIntVectorArray()`
- `SetIntVector()`
- `SetIntVectorArray()`

- `GetBoolVector()`
- `GetBoolVectorArray()`
- `SetBoolVector()`
- `SetBoolVectorArray()`

The array versions of these functions are used if you have an array of global variables that you want to set at one time. An example is `float4 colors[10]` for an array of 10 colors.

CONSTANT BUFFER SHADER DEMO

The Constant Buffer demo builds off of the Uniform Variables Shader demo and adds a few different features. The demo can be found on the CD-ROM in the Chapter 4 folder under the name Constant Buffer.

In the shader's source file, three matrices are added to the code for the world, view, and projection. Matrices will be discussed in more detail in Chapter 8. For now it is enough to know that the world matrix is used to position and orient geometry in the environment, the view matrix represents the virtual camera, and the projection matrix represents properties of the camera such as the field of view, aspect ratio, and so forth.

The projection matrix usually doesn't change much, and in most of the demos in this book it only changes upon the resizing of the application's window. Because of this, the projection matrix is placed in a constant buffer that is specified using the `cbChangeOnResize`, which tells HLSL that the contents in the constant buffer only change when the application's window resizes. The world and view matrices tend to change or at least be set often, and they are set in a constant buffer that is marked to change for every frame using `cbChangesEveryFrame`.

Matrices are used to transform a vertex from one coordinate space to another. In the vertex shader the vertex position is transformed from its local space to world space using the world matrix, then to view space using the view matrix, then to screen space using the projection matrix. Coordinate spaces and transformations will be discussed in more detail in Chapter 8. The Constant Buffer demo's shader is shown in Listing 4.9.

LISTING 4.9. THE CONSTANT BUFFER DEMO'S SHADER

```
float4 color;

cbuffer cbChangesEveryFrame
{
   matrix World;
   matrix View;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};
```

```
struct VS_INPUT
{
   float4 Pos : POSITION;
};

struct GS_INPUT
{
   float4 Pos : SV_POSITION;
};
struct PS_INPUT
{
   float4 Pos : SV_POSITION;
};

GS_INPUT VS(VS_INPUT input)
{
   GS_INPUT output = (GS_INPUT)0;

   float4 Pos = mul(input.Pos, World);
   Pos = mul(Pos, View);
   output.Pos = mul(Pos, Projection);

   return output;
}


[maxvertexcount(3)]
void GS(triangle GS_INPUT input[3],
        inout TriangleStream<PS_INPUT> triStream)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = input[0].Pos;
   triStream.Append(input[0]);

   output.Pos = input[1].Pos;
   triStream.Append(input[1]);

   output.Pos = input[2].Pos;
   triStream.Append(input[2]);
   triStream.RestartStrip();
}


float4 PS(PS_INPUT input) : SV_Target
{
   return color;
}

technique10 PassThroughShader
{
   pass P0
```

```
    {
        SetVertexShader(CompileShader(vs_4_0, VS()));
        SetGeometryShader(CompileShader(gs_4_0, GS()));
        SetPixelShader(CompileShader(ps_4_0, PS()));
    }
}
```

Since the Constant Buffer demo has three new effect variables, the application has to supply that information. Matrix variables are created using `ID3D10EffectMatrixVariable`. The global variables, including the new matrices, are shown in Listing 4.10.

LISTING 4.10. THE GLOBAL VARIABLES FROM THE CONSTANT BUFFER DEMO

```
ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_passThroughTech = NULL;

ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;
ID3D10EffectVectorVariable *g_colorEffectVar = NULL;

ID3D10InputLayout *g_shaderInputLayout = NULL;
```

We can bind the new matrix effect variables to the shader inputs. This is done by calling the `GetVariableByName()` function and calling the function `AsMatrix()` on the object that is returned. Remember to match the function call (such as `AsMatrix()` or `AsVector()`) with the proper variable type to correctly obtain and set the variable's data.

Once the application has access to the matrix variables, they can be set using the `SetMatrix()` function. This function takes a 16-element floating-point array, which can also be represented by a `D3DXMATRIX` object. To clear a matrix we must create what is known as an identity matrix, which is done by calling the function `D3DXMatrixIdentity()`. The modified `InitializeDemo()` function is shown in Listing 4.11. All of this will be reviewed in detail in Chapter 8.

LISTING 4.11. THE MODIFIED `INITIALIZEDEMO()` FUNCTION FOR THE CONSTANT BUFFER DEMO

```
bool InitializeDemo()
{
    // Load the shader.

    DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3D10_SHADER_DEBUG;
#endif

    ID3D10Blob *errors = NULL;

    HRESULT hr = D3DX10CreateEffectFromFile("shader.fx", NULL,
NULL,
```

```
      "fx_4_0", shaderFlags, 0, g_d3dDevice, NULL, NULL,
      &g_shader, &errors, NULL);

   if(errors != NULL)
   {
      MessageBox(NULL, (LPCSTR)errors->GetBufferPointer(),
                 "Error in Shader!", MB_OK);
       errors->Release();
    }

    if(FAILED(hr))
       return false;

    g_passThroughTech = g_shader->GetTechniqueByName(
       "PassThroughShader");

    g_worldEffectVar = g_shader->GetVariableByName(
       "World")->AsMatrix();

    g_viewEffectVar = g_shader->GetVariableByName(
       "View")->AsMatrix();

    g_projEffectVar = g_shader->GetVariableByName(
       "Projection")->AsMatrix();

    g_colorEffectVar = g_shader->GetVariableByName(
       "color")->AsVector();


    …

    D3DXMatrixIdentity(&g_worldMat);
    D3DXMatrixIdentity(&g_viewMat);
    g_projEffectVar->SetMatrix((float*)&g_projMat);

    return true;
}
```

The rendering function for the Constant Buffer demo adds functions that set the `extern` uniform variables for the world and view matrices. This is done by calling the function `SetMatrix()`. Since the world and view matrices in the demo are `D3DXMATRIX` objects, you can cast them to float to send to the `SetMatrix()` function. This is shown Listing 4.12.

LISTING 4.12. THE CONSTANT BUFFER DEMO'S RENDERING FUNCTION

```
void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);
```

```
    unsigned int stride = sizeof(D3DXVECTOR3);
    unsigned int offset = 0;

    g_d3dDevice->IASetInputLayout(g_shaderInputLayout);
    g_d3dDevice->IASetVertexBuffers(0, 1, &g_triVB,
                                    &stride, &offset);

    g_d3dDevice->IASetPrimitiveTopology(
       D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    D3D10_TECHNIQUE_DESC techDesc;
    g_passThroughTech->GetDesc(&techDesc);

    float redCol[4] = { 1.0f, 0.0f, 0.0f, 1.0f };

    g_colorEffectVar->SetFloatVector(redCol);
    g_worldEffectVar->SetMatrix((float*)&g_worldMat);
    g_viewEffectVar->SetMatrix((float*)&g_viewMat);

    for(unsigned int i = 0; i < techDesc.Passes; i++)
    {
       g_passThroughTech->GetPassByIndex(i)->Apply(0);
       g_d3dDevice->Draw(3, 0);
    }

    g_swapChain->Present(0, 0);
}
```

The last thing to note in this demo is that the projection matrix is calculated and set on a resize event. For now, know that the projection matrix controls the appearance of objects as they are rendered to the screen. Perspective projection adds perspective to the scene, which is what we usually want, while orthogonal projection does not use perspective. Orthogonal projection renders objects the same size regardless of how far away they are moving, while perspective naturally gives the impression that objects are getting smaller with distance.

To set the projection matrix to the shader effect, we set the projection effect variable we've created the same way we've done for the view and world matrices: using the `SetMatrix()` function. To create a perspective matrix we can use the function `D3DXMatrixPerspectiveFovLH()`, which is shown here but will be discussed in more detail in Chapter 8. The function prototype for `D3DXMatrixPerspectiveFovLH()` and the updated resize function for the demo that uses it are shown in Listing 4.13.

LISTING 4.13. UPDATED RESIZE FUNCTION FOR THE CONSTANT BUFFER DEMO

```
void ResizeD3D10Window(int width, int height)
{
    if(g_d3dDevice == NULL)
       return;

    D3D10_VIEWPORT vp;
    vp.Width = width;
```

```
   vp.Height = height;
   vp.MinDepth = 0.0f;
   vp.MaxDepth = 1.0f;
   vp.TopLeftX = 0;
   vp.TopLeftY = 0;

   g_d3dDevice->RSSetViewports(1, &vp);

   D3DXMatrixPerspectiveFovLH(&g_projMat, (float)D3DX_PI *
0.25f,
                              width/(FLOAT)height, 1.0f,
1000.0f);

   if(g_projEffectVar != NULL)
      g_projEffectVar->SetMatrix((float*)&g_projMat);
}
```

```
D3DXMATRIX * D3DXMatrixPerspectiveFovLH(
   D3DXMATRIX *pOut,
   FLOAT fovy,
   FLOAT Aspect,
   FLOAT zn,
   FLOAT zf
);
```

The `D3DXMatrixPerspectiveFovLH()` function returns the address to the projection matrix or can have it returned to the address of the first parameter and takes the field of view, the aspect ratio, the near plane (how close to the camera objects can be drawn), and the far plane (to what distance objects can be drawn) of the camera.

SUMMARY

Shaders are a very important part of computer graphics, and in the video games industry programmable shaders are a common and standard part of any game. Direct3D 10 requires at least a basic understanding of HLSL and shaders to display geometry properly to the screen. Throughout the remainder of this book shaders play a crucial role in the scenes that will be rendered.

In this chapter we took a look at Direct3D's HLSL along with features specific to Shader Model 4 such as geometry shaders. Throughout the remainder of this book we will be using shaders to create various effects that include but are not limited to the following.

- Per-pixel lighting
- Shadows
- Image filters as a post-processing effect
- Texture mapping
- Bump mapping

The following elements were discussed in this chapter.

- Shaders in general
- Direct3D's High-Level Shading Language
- Vertex shaders
- Pixel shaders
- Geometry shaders
- Constant buffers

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. Define programmable shaders.

2. What does HLSL stand for?

3. What does GLSL stand for?

4. What is a shader model? What are the versions discussed in this chapter?

5. What is the difference between low-level and high-level shaders?

6. List three of the issues discussed in this chapter that occur when working with low-level programmable shaders.

7. Define a vertex shader. What stage(s) accepts the vertex shader's output as input?

8. Define a geometry shader. What stage(s) accepts the geometry shader's output as input?

9. Define a pixel shader. What stage(s) accepts the pixel shader's output as input?

10. What is the input layout of the input assembler?

11. What data type is used for effect shaders in Direct3D 10?

12. What is the fixed-function pipeline?

13. List three of the limitations of the fixed-function pipeline that were discussed in this chapter.

14. What does it mean to have a unified shader core (architecture)?

15. Between which stages does the geometry shader sit?

16. List the various data types in the vector type.

17. List the various data types in the scalar type.

18. List the various data types in the matrix type.

19. List the seven data types in the sampler type.

20. What is a constant buffer, and what use does it have in shaders?

21. List and define the three constant buffer usages discussed in this chapter.

22. What is a texture buffer?

23. Define semantics.

24. What is the `Buffer` type used for in the HLSL syntax?

25. List and define four of the storage classes with which a variable can be defined.

26. Static cannot be used for what type of variables in an HLSL effect shader?

27. Define dynamic branching.

28. What does the `SV` in `SV_POSITION` stand for?

29. Define a perspective projection as discussed in this chapter.

30. Define an orthogonal projection as discussed in this chapter.

CHAPTER EXERCISES

Exercise 1: Change the output color to blue in the Shader Example demo.

Exercise 2: Add a uniform variable to the Uniform Variables Shader demo that is a single float that represents the color's brightness. Set this variable to a value between 0.0 and 1.0 and multiply it by the color in the pixel shader. The closer to 0.0 this new variable is, the darker the color should appear.

Exercise 3: Change the single color to an array of colors in the Uniform Variables Shader demo and add another uniform variable that will serve as an index into that array. In the application allow the index to be set for which color the user wants displayed and use that index to access a color value in the colors array. You can use any colors you want, but don't use two of the same colors when setting the colors array.

5. TRANSFORMATIONS

In This Chapter

- Projection Transformations
- World Transformations
- View Transformations
- Transformation Demo

As geometry is passed to the vertex shader, it is often operated on by various pieces of information to prepare it for its final rendering position. This information often includes adding projections to the scene, positioning objects throughout the 3D scene, and view or camera information.

The purpose of this chapter is to cover projection, world, and view transformations. These transformations often take place in the vertex shader and are a very important topic to discuss. All of these topics deal with matrices, which are discussed in detail in Chapter 8, "Game Math."

PROJECTION TRANSFORMATIONS

Projection transformations affect how a rendered scene looks when displayed to the screen. The two main types of projections are orthogonal projections and perspective projections, both of which are supported by Direct3D. A projection is a matrix that stores projection information. To apply the projection to the geometry in the scene, we multiply the projection matrix and vertices of the geometry, which is a process known as transformation. A projection is a representation of how objects are viewed when rendered. The second type, orthogonal projection, will be discussed next.

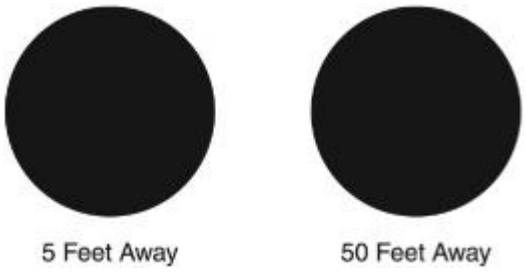
ORTHOGONAL PROJECTION

Orthogonal projection causes all objects to be rendered to the screen at the same size regardless of how far away an object is. In the real world, as objects move farther away from you, they appear smaller. An example of this is shown in Figure 5.1.

FIGURE 5.1. AN EXAMPLE OF OBJECTS GETTING SMALLER AS THEY MOVE FARTHER AWAY.

In orthogonal projection, the size of the objects does not change due to distance. Many times, this effect is desired, but for most 3D scenes in modern video games it is often important to have a different type of projection. Orthogonal projection can be a great type of projection for 2D elements such as menus, heads-up displays, and any other type of rendering where the geometry is not to change in size with distance. An example of orthogonal projection is shown in Figure 5.2.

FIGURE 5.2. AN EXAMPLE OF ORTHOGONAL PROJECTION.



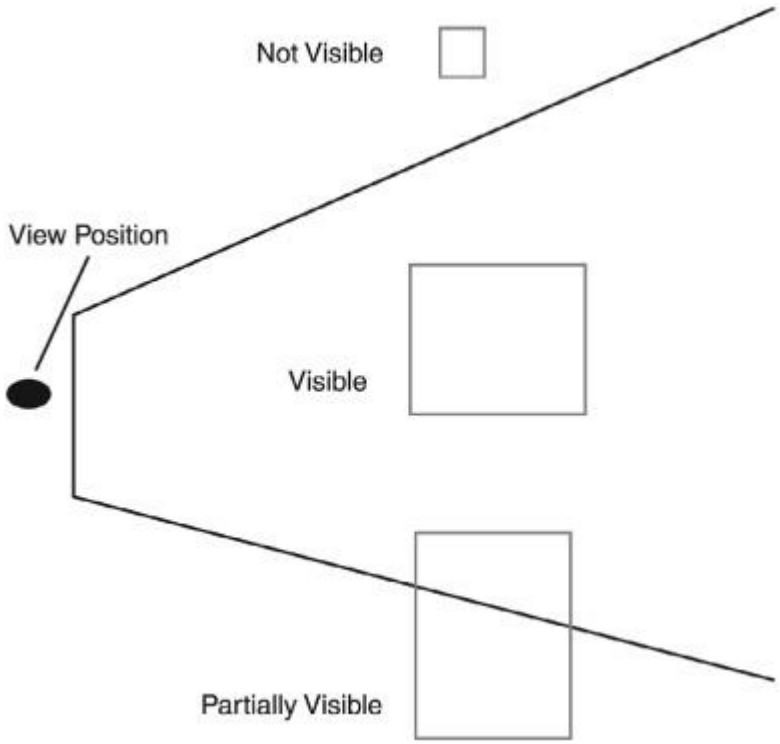
In Direct3D there are four different functions for creating an orthogonal matrix. The first two functions are `D3DXMatrixOrthoLH()` and `D3DXMatrixOrthoRH()`. The LH version creates a left-handed projection matrix, while RH creates a right-handed projection matrix. Their function prototypes are as follows.

```
D3DXMATRIX * D3DXMatrixOrthoLH(
    D3DXMATRIX *pOut,
    FLOAT w,
    FLOAT h,
    FLOAT zn,
    FLOAT zf
);

D3DXMATRIX * D3DXMatrixOrthoRH(
    D3DXMATRIX *pOut,
    FLOAT w,
    FLOAT h,
    FLOAT zn,
    FLOAT zf
);
```

The parameters of the functions start with the `D3DXMATRIX` object, which is the structure that represents matrices in Direct3D, which will be created from the function call, the width

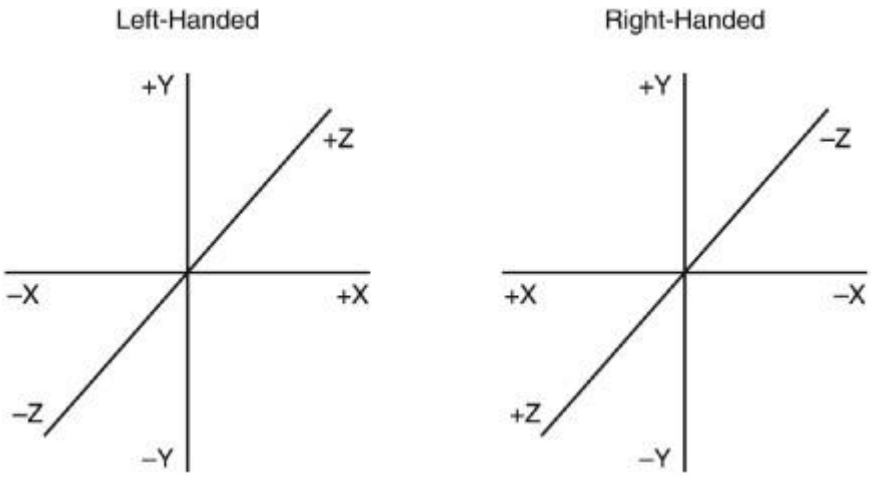
and height of the desired view volume, and the near and far plane. The near plane determines how close to the viewer an object can be before it is seen, while the far plane determines how far away an object can be before it disappears. The width and height, which is normally the width and height of the screen or rendering area, in addition to the near and far values collectively represent the view volume. The view volume is an area in which objects are visible. (see Figure 5.3)

FIGURE 5.3. AN EXAMPLE OF A VIEW VOLUME.



The left- and right-handedness of the functions refer to coordinate systems. A coordinate system essentially tells the graphics API which direction, left or right, the positive X axis travels and which direction, toward or away, the positive Z axis travels. An illustration is shown in Figure 5.4.

FIGURE 5.4. LEFT- AND RIGHT-HANDED COORDINATE SYSTEMS.



OpenGL uses a right-handed coordinate system, while Direct3D traditionally used a left-handed system.

Direct3D allows developers to use either left- or right-hand coordinates. Using a right-handed system allows developers to use the same data in OpenGL and Direct3D applications without modification of the geometry's X and Z axes. This is the reason behind the multiple versions of the orthogonal projection functions. The last two orthogonal projection functions are as follows.

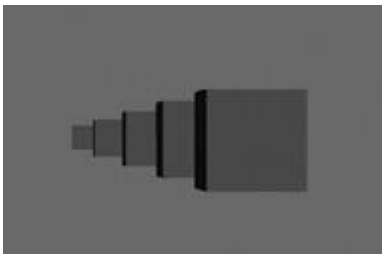
```
D3DXMATRIX * D3DXMatrixOrthoOffCenterLH(
    D3DXMATRIX *pOut,
    FLOAT l,
    FLOAT r,
    FLOAT b,
    FLOAT t,
    FLOAT zn,
    FLOAT zf
);
D3DXMATRIX * D3DXMatrixOrthoOffCenterRH(
    D3DXMATRIX *pOut,
    FLOAT l,
    FLOAT r,
    FLOAT b,
    FLOAT t,
    FLOAT zn,
    FLOAT zf
);
```

The `l` and `r` parameters represent the minimum and maximum width, while `b` and `t` represent the minimum and maximum height. `zn` and `zf` are the near and far values. The `D3DXMatrixOrthoLH()` and `D3DXMatrixOrthoRH()` functions are special cases of `D3DXMatrixOrthoOffCenterLH()` and `D3DXMatrixOrthoOffCenterRH()`. The off center functions allow more customizability than the other two seen earlier in this section.

PERSPECTIVE PROJECTION

The other type of projection is perspective projection. This type of projection adds perspective to scenes. Perspective projection allows objects to shrink as they move farther away from the viewer. Objects also distort as they are viewed at an angle. Figure 5.5 shows an example of perspective projection. Perspective projection is a type of projection that can be seen in all modern 3D video games.

FIGURE 5.5. PERSPECTIVE PROJECTION.



Perspective projection is the same as the idea behind perspective in drawing art.

The perspective projection matrix functions are as follows, where the parameters match those of the orthogonal counterparts.

```
D3DXMATRIX * D3DXMatrixPerspectiveLH(
   D3DXMATRIX *pOut,
   FLOAT w,
   FLOAT h,
   FLOAT zn,
   FLOAT zf
);

D3DXMATRIX * D3DXMatrixPerspectiveRH(
   D3DXMATRIX *pOut,
   FLOAT w,
   FLOAT h,
   FLOAT zn,
   FLOAT zf
);

D3DXMATRIX * D3DXMatrixPerspectiveOffCenterLH(
   D3DXMATRIX *pOut,
   FLOAT l,
   FLOAT r,
   FLOAT b,
   FLOAT t,
   FLOAT zn,
   FLOAT zf
);

D3DXMATRIX * D3DXMatrixPerspectiveOffCenterRH(
   D3DXMATRIX *pOut,
   FLOAT l,
   FLOAT r,
   FLOAT b,
   FLOAT t,
   FLOAT zn,
```

```
    FLOAT zf
);
```

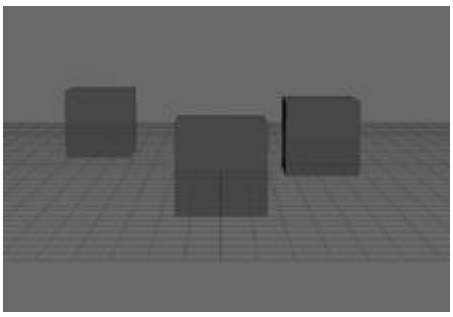
WORLD TRANSFORMATIONS

In modern video games many different 3D objects can be found throughout a scene. Some of these objects are dynamic, such as character models and vehicles, while others are static. Dynamic objects are objects that move either by player control, game physics, or artificial control. Static objects cannot move at all. A static object can be a building, the terrain, or some other object that the designer does not mean to be dynamic.

When objects are created in 3D modeling and animation applications, they are saved out and imported by the game. The vertex positions of these objects are not necessarily the positions they need to be in the game. For example, if a box is created around the origin (0, 0, 0) and that box needs to be in the second story of a virtual building, unless that position just happens to be around the origin, the data needs to be altered. If complex models are created and need to be rendered more than once—for example, having the same box appear 50 times in a level—then it would be inefficient to create the exact same geometry but at different positions again and again so that the boxes load in the correct spots in a game. To further complicate things, dynamic objects need to be moved on the fly. When it comes to character models, how and where these objects move are unpredictable.

The purpose of a world matrix transformation is to position and rotate objects in a 3D scene. This allows characters to move through the world, allows objects to be rendered at multiple locations, and so on. The world matrix represents an object's position in the game world. This means objects can be created in a modeling package such as 3D Studio Max and moved, rotated, and scaled as necessary so that they appear how the designer requires in a game. This is often done in a level editor, where objects can be positioned, rotated, and scaled. A visual of using world positions to place objects is shown in Figure 5.6.

FIGURE 5.6. DIFFERENT WORLD POSITIONS FOR THE SAME OBJECT.



Scaling refers to changing an object's size from its original size. To scale a matrix you call the function `D3DXMatrixScaling()`, which has the following prototype.

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

The D3DXMatrixScaling() function parameters are the matrix that is being scaled and the X, Y, and Z axis factors. To set the position, also known as translation, of a matrix, you use the function D3DXMatrixTranslation(). The function takes the matrix and the positions X, Y, and Z value of the scaling factors. The function prototype for the D3DXMatrixTranslation() can be seen as follows.

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

Rotations and other matrix-related topics and functions will be discussed in more detail in Chapter 8.

VIEW TRANSFORMATIONS

The view transformation matrix is a matrix that represents the viewer. The viewer is commonly known as the game's camera. A camera is represented by a position and a direction in which it is pointing. In OpenGL and Direct3D, camera matrices are often known as look-at matrices. In Direct3D a look-at matrix can be created using either a left- or right-handed function much like the projection matrix. These functions are called D3DXMatrixLookAtLH() and D3DXMatrixLookAtRH() and have the following prototypes.

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

D3DXMATRIX * D3DXMatrixLookAtRH(
   D3DXMATRIX *pOut,
   CONST D3DXVECTOR3 *pEye,
   CONST D3DXVECTOR3 *pAt,
   CONST D3DXVECTOR3 *pUp
);
```

The eye vector is the 3D position of the camera. The look-at vector is the position in 3D space at which you are looking. The position you are located at and the position you are looking at define the view's direction. The up vector is a vector that represents which direction is up. The up vector can be used to allow the camera to be rotated. In a first-person camera, this can be used to rotate the position you are looking at (look-at point) around the up vector to look around you from left to right.

We'll hold off on discussing the details about views, as well as matrices in general, until the mathematics chapter, Chapter 8. The purpose of this demo is to see how to transform

vectors in a vertex shader; the details and math behind it beyond what was discussed in this chapter come later.

The concatenation of the projection, world, and view matrices forms the model-view projection matrix. This is also known as the MVP matrix.

TRANSFORMATION DEMO

On the book's accompanying CD-ROM is a demo application called Transformations that demonstrates how to create a projection, view, and world matrix and how to use them in a vertex shader. The demo application builds off of the Primitives demo. If you want to follow along with the coding of this demo, you can use the Primitives demo's source code as a starting point.

The global section of the Transformations demo starts by adding matrices for the projection, view, and world. It also adds effect variables that allow these matrices to be bound to the shader. The global section from the Transformations demo is shown in Listing 5.1. Effect matrix variables are represented by the type `ID3D10EffectMatrixVariable`.

LISTING 5.1. THE GLOBAL SECTION FROM THE TRANSFORMATIONS DEMO

```
#include<windows.h>
#include<d3d10.h>
#include<d3dx10.h>

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Transformations"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600

// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

// Effect objects and variables.
ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_technique = NULL;
ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;

// Display object to store scene geometry.
ID3D10InputLayout *g_layout = NULL;
```

```
ID3D10Buffer *g_vertexBuffer = NULL;

// Structure used to represent a single vertex.
struct DX10_Vertex
{
   D3DXVECTOR3 pos;
};

// Projection, world, and view transformations.
D3DXMATRIX g_worldMat, g_viewMat, g_projMat;
```

The projection matrix depends on the window's dimensions. Because of this, the projection matrix is set in the demo's resizing function, `ResizeD3D10Window()`. This function in the Transformations demo adds a line of code at the end that creates a left-handed projection matrix using `D3DXMatrixPerspectiveFOVLH()`, which is shown in Listing 5.2.

LISTING 5.2. THE RESIZING FUNCTION FROM THE TRANSFORMATIONS DEMO

```
void ResizeD3D10Window(int width, int height)
{
   if(g_d3dDevice == NULL)
      return;

   D3D10_VIEWPORT vp;

   vp.Width = width;
   vp.Height = height;
   vp.MinDepth = 0.0f;
   vp.MaxDepth = 1.0f;
   vp.TopLeftX = 0;
   vp.TopLeftY = 0;

   g_d3dDevice->RSSetViewports(1, &vp);

   D3DXMatrixPerspectiveFovLH(&g_projMat, (float)D3DX_PI *
0.25f,
                              width/(FLOAT)height, 0.1f,
1000.0f);
}
```

The demo's initialization function adds code to bind the effect variables and to set the two remaining matrices. The projection matrix is set and updated by the `ResizeD3D10Window()` function, so it is not included in the initialize function. The world matrix is not set to anything, so it is cleared by calling `D3DXMatrixIdentity()`. The `D3DXMatrixIdentity()` function is essentially used to clear matrices, which is discussed mathematically in Chapter 8. The view matrix is set 15 units back, which places the view farther back in the scene than what it was in the Primitives demo. The square looks smaller in the scene because perspective projection is being used. There is no difference between the square geometry in the Transformations demo and that in the Primitives demo. The difference is in the camera's position. The demo's initialize function is shown in Listing 5.3. Effect variables are obtained by calling the `GetVariableByName()` function, which takes as a parameter the name used in the shader file for the variable.

LISTING 5.3. THE DEMO'S INITIALIZE FUNCTION

```
bool InitializeDemo()
{

   …

   g_technique = g_shader->GetTechniqueByName("PassThrough");
   g_worldEffectVar = g_shader->GetVariableByName(
      "World")->AsMatrix();
   g_viewEffectVar = g_shader->GetVariableByName(
      "View")->AsMatrix();
   g_projEffectVar = g_shader->GetVariableByName(
      "Proj")->AsMatrix();

   …
   // Clear the matrices.
   D3DXMatrixIdentity(&g_worldMat);
   D3DXMatrixIdentity(&g_viewMat);

   D3DXVECTOR3 eye(0, 0, -15);
   D3DXVECTOR3 lookAt(0, 0, 0);
   D3DXVECTOR3 up(0, 1, 0);

   D3DXMatrixLookAtLH(&g_viewMat, &eye, &lookAt, &up);

   return true;
}
```

In the rendering function, three added lines of code are used to set the matrices to the shader. Technically, it is only necessary to set effect variables when they change or when you switch shaders. The matrices are set by using the effect variable's function `SetMatrix()`, which takes as a parameter the matrix represented by a float array. This can be done by casting the matrix to a float pointer. The rendering function from the Transformations demo is shown in Listing 5.4.

LISTING 5.4. THE RENDERING FUNCTION FROM THE TRANSFORMATIONS DEMO

```
void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   // Clear the rendering destination to a specified color.

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

   unsigned int stride = sizeof(DX10_Vertex);
   unsigned int offset = 0;

   // Setup the geometry buffer that will be rendered.

   g_d3dDevice->IASetInputLayout(g_layout);
```

```
    g_d3dDevice->IASetVertexBuffers(0, 1, &g_vertexBuffer,
                                    &stride, &offset);
    g_d3dDevice->IASetPrimitiveTopology(
       D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    // Prepare the effect we will use to draw the geometry.

    g_viewEffectVar->SetMatrix((float*)&g_viewMat);
    g_projEffectVar->SetMatrix((float*)&g_projMat);
    g_worldEffectVar->SetMatrix((float*)&g_worldMat);

    D3D10_TECHNIQUE_DESC techDesc;
    g_technique->GetDesc(&techDesc);

    // Loop through each pass of the technique and draw.

    for(unsigned int i = 0; i < techDesc.Passes; i++)
    {
       g_technique->GetPassByIndex(i)->Apply(0);
       g_d3dDevice->Draw(6, 0);
    }

    // Display the results to the target window (swap chain).

    g_swapChain->Present(0, 0);
}
```

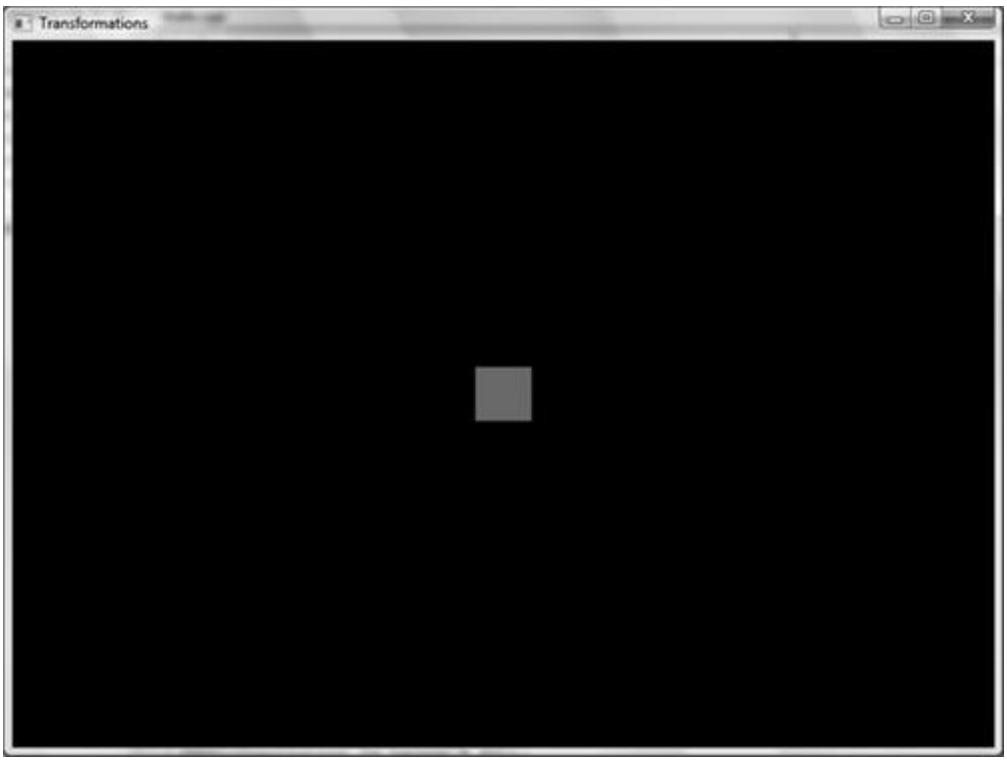
The last code that needs to be seen is the Transformations demo's shader. This shader starts by defining three variables (`World`, `View`, and `Proj`), each of which are obtained by the application by name. These variables are used by the vertex shader to multiply the matrices against the incoming vertex position. The vertex is transformed by the world matrix first, then the view, and then the projection. If the world, view, and projection matrices where concatenated into one, then one multiplication would be used. In this demo we'll see the longer form by multiplying against each in order. The rest of the shader is the same from the Primitives demo. The shader from the Transformation demo is shown in Listing 5.5. Figure 5.7 shows a screenshot of the demo.

LISTING 5.5. THE TRANSFORMATIONS DEMO'S SHADER

```
/*
  Chapter 5 - Transformations
  Ultimate Game Programming with DirectX 2nd Edition
  Created by Allen Sherrod
*/

matrix World;
matrix View;
matrix Proj;

struct VS_INPUT
{
  float4 Pos : POSITION;
};
```

```
struct PS_INPUT
{
  float4 Pos : SV_POSITION;
};

PS_INPUT VS(VS_INPUT input)
{
  PS_INPUT output = (PS_INPUT)0;

  output.Pos = mul(input.Pos, World);
  output.Pos = mul(output.Pos, View);
  output.Pos = mul(output.Pos, Proj);
  return output;
}

float4 PS(PS_INPUT input) : SV_Target
{
  return float4(1, 0, 1, 1);
}

technique10 PassThrough
{
  pass P0
  {
     SetVertexShader(CompileShader(vs_4_0, VS()));

     SetGeometryShader(NULL);

     SetPixelShader(CompileShader(ps_4_0, PS()));
  }
}
```

FIGURE 5.7. A SCREENSHOT FROM THE TRANSFORMATIONS DEMO.



SUMMARY

Transformations are very important in video game graphics. The vertex shader is often responsible for forming the transformation of vectors against various matrices. By transforming our data we are able to represent cameras, view projections, local orientations for both static and dynamic geometry, and more.

The following elements were discussed in this chapter:

- Transformations in general
- Projection transformations
- World transformations
- View transformations

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What are projection transformations?

2. Describe orthogonal projection.

3. Describe perspective projection.

4. In what shader does transformation often take place?

   A. Geometry

   B. Vertex

   C. Pixel

5. Multiplying a vector and a matrix together is known as what?

   A. Vector-matrix multiplication

   B. Vector transform

   C. Concatenation

   D. None of the above

6. What are the two types of coordinate systems? Describe each.

7. What is the purpose of a world matrix?

8. What is the purpose of the view matrix?

9. What is the name of the concatenation result of the projection, world, and view matrices?

10. What three elements can be used to build a view matrix in Direct3D?

11. True or false: Matrices can be concatenated together.

12. True or false: Vectors can be concatenated into a matrix.

13. True or false: There are generally three types of projections.

14. True or false: World and local are two different names for the same type of matrix.

15. True or false: The order in which matrices are multiplied matters.

CHAPTER EXERCISES

Exercise 1: Change from perspective projection in the Transformations demo to orthogonal projection.

Exercise 2: Place the information needed to represent a view in a class. Call this class a camera class.

Exercise 3: Build off of Exercise 2 and allow the view's position to be controlled by the keyboard. Use the Win32 function `GetAsyncKeyState()` to detect input from the keyboard, which has the following function prototype:

```
SHORT GetAsyncKeyState(int vKey);
```

Use the virtual key codes `VK_LEFT`, `VK_RIGHT`, `VK_UP`, and `VK_DOWN` for the parameter to detect input from the arrow keys.

6. SHADING AND SURFACES

In This Chapter

- Textures
- Types of Textures
- Textures in Direct3D 10
- Implementing Texture Mapping
- Additional Texturing Topics

The simulation and variation of detail is very important when creating 3D environments for players to experience. To realistically represent something such as a brick wall using nothing more than colored polygons would take such a tremendous amount of data that it would be not only impractical but impossible to represent the types of scenes we see in today's games in that manner. What is needed is a way to simulate detail in game objects and structures without overwhelming the user's hardware with millions upon millions of polygons just for one object.

In this chapter we will discuss a very important technique that is used in video game graphics to simulate this detail across surfaces. This technique is known as texture mapping, and it is the basis for a lot of effects in video games. When creating complex virtual scenes, the topic of texture mapping is unavoidable and is something that all graphics programmers will have to tackle early on in their education.

TEXTURES

A texture is data that is used during the shading process of surfaces to give them more detail. Textures are usually images that are loaded from image file formats such as .JPG, .TGA, .BMP, .DDS, and .PNG. The images themselves are usually created using an application such as Adobe Photoshop CS 3 or Microsoft Paint. Images that are not loaded from files can be created using mathematical algorithms, which are common for textures known as procedural textures. The topic of generating textures using algorithms is beyond the scope of this book, as they can become quite mathematically intense and require a lot of background knowledge.

The purpose of this chapter is to describe how to load textures into Direct3D 10 and how to apply them to surfaces in real time within our virtual environments. This is important

because textures and their various uses are critical to many of the graphical techniques commonly used in video games. Throughout this book, textures will come up repeatedly in the contexts of various topics, so a firm understanding of them is of the utmost importance before moving on.

TYPES OF TEXTURES

Several types of images can be loaded and used in Direct3D. Each of these texture types has its own purpose, each of which will be discussed in this section. The types of textures that can be used in Direct3D include the following texture types, but what each texture type is used for depends on what purpose the texture serves in the application, which can be different than storing color information of a surface:

- 1D textures
- 2D textures
- 3D textures
- Cube maps
- Sphere maps

1D TEXTURES

A 1D texture is akin to a 1D array of values. A 1D texture is often used as a look-up table in one or more shaders. A look-up table in this sense refers to sending an array of data to a shader so the shader can look up the values in the array to compute whatever value it is meant to compute. For example, let's say you filled in an array with 10 color values. You can then use some attribute of the vertex—for example, its height—as an array index into the color array to select the prespecified color. This can be done by taking the range of the height (where the height falls within the minimum and maximum possible) as a percentage and then multiplying that percentage by the size of the array. The integer result could then be used as an array index to select a color from the look-up table.

2D TEXTURES

2D textures are the most common types of textures that you will likely work with in your game projects. A 2D texture is essentially an ordinary image. Technically, anything can be stored in a texture that includes information not used for color, which we'll discuss later (e.g., the alpha mapping technique in Chapter 7, "Additional Texture Mapping," and bump mapping in Chapter 13, "Lighting").

The best way to understand a 2D texture as an image is to open up any image file you have on your computer that has a width and a height. In a 1D texture, the data can be thought of as rows without columns or widths without heights, but a 2D texture has both a width and a height. An example of a 2D texture created in Adobe Photoshop is shown in Figure 6.1.

FIGURE 6.1. A 2D TEXTURE.

In this chapter we will focus on the loading and rendering of these 2D texture images. The textures loaded in this chapter will be used to color a surface to increase the surface details. Textures used in this manner are known as color maps or decal maps.

3D TEXTURES

3D textures are also called volume textures. A 3D texture is a texture that has a width, a height, and a depth. The depth part of the 3D texture is what makes it a volume rather than a flat slice like a 2D texture image. 3D textures can be used for the following graphical effects:

- Volumetric fog
- Volumetric clouds
- Various other environmental effects such as 3D textures used to light an environment
- Visualization for scientific analysis instruments

3D textures have traditionally required a lot of processing power and memory in applications such as video games. Therefore, not many games use many 3D textures. For example, a 2D texture of 128 × 128 is relatively small, especially considering that many textures in today's games exceed the resolution of 1024 × 1024. A 128 × 128 2D texture has 16,384 pixels. If each pixel is three bytes in size, then a 128 × 128 2D texture is 49,152 bytes, or almost 50 kilobytes. If you had a 3D texture with the same size across all dimensions, such as 128 wide, 128 high, and 128 deep, you would end up with a texture that has 2,097,152 pixels and would be 6,291,456 bytes in size assuming 3 bytes per pixel.

In other words, a 3D texture with the resolution of 128 × 128 × 128 would be 6 megabytes in size for a single, relatively low-resolution 3D texture. Imagine how big a 512 × 512 × 512 3D texture would be. Even at a cubic size of 128, the difference between 6 megabytes and 50 kilobytes is beyond tremendous. If you had a 3D texture with a cubic size of 512, more memory (over to 380 megabytes, assuming 24 bits) would be required for that one texture than some games have for entire game levels. Even if you assumed a 600-megabyte budget of texture data for each game level in your game, a single 512 resolution 3D texture would consume more than half that budget.

CUBE AND SPHERE MAPS

A cube map is a collection of six 2D texture images that together often represent the view of an environment from a point in space. A 3D texture is a volume, but a cube map is just six 2D images that create not a volume but rather what is known as an environment map. An environment map can be created dynamically by placing the camera in the game world and saving the view's render as a texture six times for the forward, backward, up, down, left, and right directions.

Technically the data in a cube map can be used however you choose, but cube maps are commonly used for storing the scene's environment so that other graphical techniques such as reflection mapping (simulating reflections using textures) can be performed on objects. An example of a cube map is shown in Figure 6.2. A cube map can be six separate images or one large image. Later in this chapter you'll see how to create and load cube maps in Direct3D 10.

FIGURE 6.2. A CUBE MAP.

A sphere map is a 2D texture image in which the contents are spherical. When sampling from the texture, specific equations are used to retrieve the information in a way that allows the sphere map to be used to texture a sphere or some other object with volume. Sphere maps and cube maps tend to serve the same purpose, which is to store an environment's information in an image. An example of a sphere map is shown in Figure 6.3.

FIGURE 6.3. A SPHERE MAP.

TEXTURES IN DIRECT3D 10

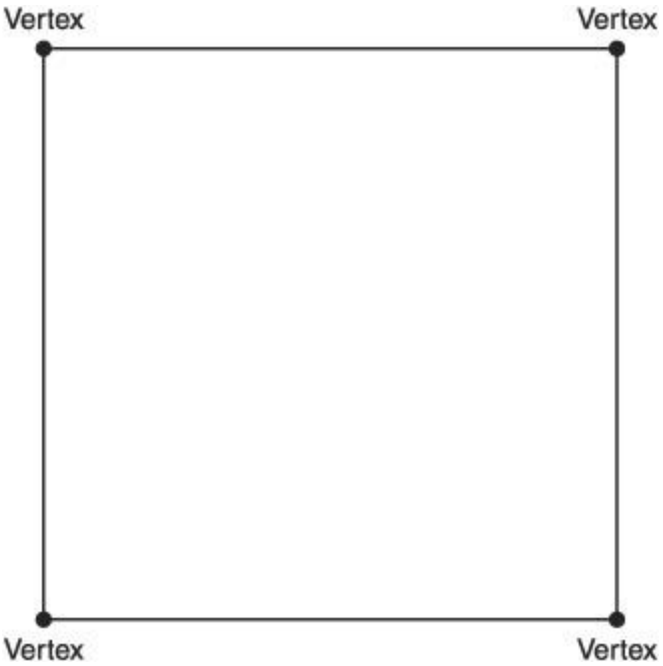
In this chapter we will load texture images that were created from tools such as Adobe Photoshop and display them on surfaces in Direct3D 10. The topic of actually creating texture images is beyond the scope of this book and is more suited for a text that deals with digital art. Since this book focuses on programming, we will focus on the code necessary to load and display textures. All of the sample demos in this chapter come with one or more texture images on the CD-ROM.

TEXTURES COORDINATES

To properly display textures on surfaces, we use texture coordinates. A texture coordinate is an attribute of the vertex in the same way the position is an attribute that is used to specify how textures are to be mapped onto surfaces. When performing texture mapping, we must specify, along with the positions of each vertex, the texture coordinates.

In Figure 6.4, four vertices form the shape of the square. The position property of the vertex specifies how the shape appears in the 3D virtual world. The texture coordinates for each vertex, on the other hand, specify how the texture image is displayed on the rendered surface.

FIGURE 6.4. A SQUARE DISPLAYING VERTICES.



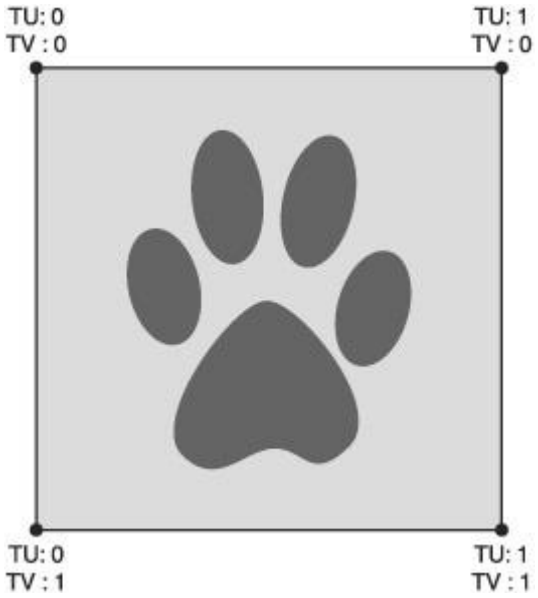
A 1D texture has a single value for the texture coordinate of a vertex, a 2D texture uses two values (one for the width and one for the height), and a 3D texture uses three values. Cube maps use three values since the six images make up a single cube texture map whereas a cube map is made up of six 2D textures, and sphere maps use two since sphere maps are just 2D images.

A texture coordinate is essentially a percentage and is defined using floating-point data types. Using 2D texture coordinates as an example, the first value in the texture coordinate

is the percentage from 0.0 to 1.0 (in other words, 0% to 100%) of how far along the width this vertex is mapped onto the image, and the second value is the percentage for the height. These are the S and T texture coordinates for 2D textures. They are also known as the U and V or the TU and TV texture coordinates. For 3D textures, you have S, T, and Q or TU, TV, and TW. The value of a texture coordinate can go below 0.0 or above 1.0, which can be useful for tiling a texture so that it appears across a surface repeatedly.

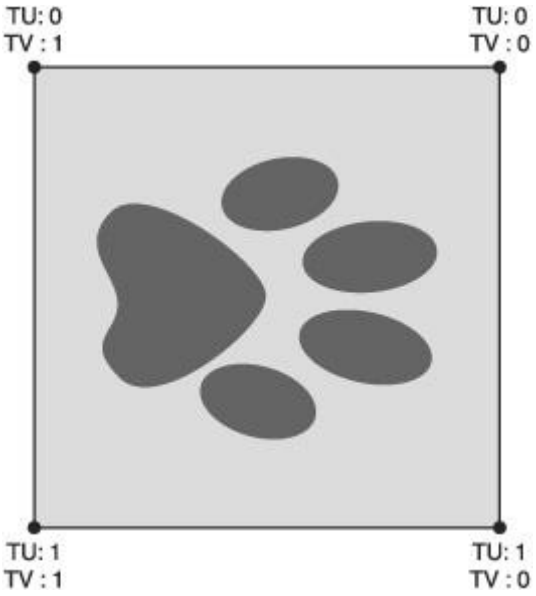
In Figure 6.5 the upper-left vertex point has a texture coordinate of 0.0 for the S and 0.0 for the T. This tells graphics APIs such as Direct3D where in the image the mapping should be for that vertex. The upper-right vertex has 1.0 and 0.0 for the coordinates; that is, the upper-right vertex should have the upper-right portion of the textured image mapped to it (i.e., 100% of the width but 0% of the height).

FIGURE 6.5. TEXTURE COORDINATES FOR A SURFACE'S VERTICES.



When a primitive is mapped, the relationship of all the vertices of the shape determines how the object will look. So using the example in Figure 6.5, the image is displayed on the surface as if it was opened up normally in an image editor. However, in Figure 6.6, you can see that changing the texture coordinates will alter how the image is mapped unto the surface. How the image appears mapped on the surface is solely dependent on the relationship of all the vertices.

FIGURE 6.6. CHANGING THE TEXTURE COORDINATES ALTERS THE OUTPUT.



In Direct3D, 0.0 for the S (TU) texture coordinate represents the left-most part of the image, and 1.0 is the right-most part. For the T (TV) coordinate 0.0 is the top-most part and 1.0 is the bottom-most part. Any value between 0.0 and 1.0 falls within that range.

Up to this point we've specified vertices using three floating-point values for the position. From here on, whenever texture mapping is used, we will need five floating-point values for 2D textures, where the first three are for the position and the last two values are the S and T texture coordinates. Throughout the remainder of this book, we will refer to the texture coordinates for a 2D texture as the TU and TV pair since many books use that terminology.

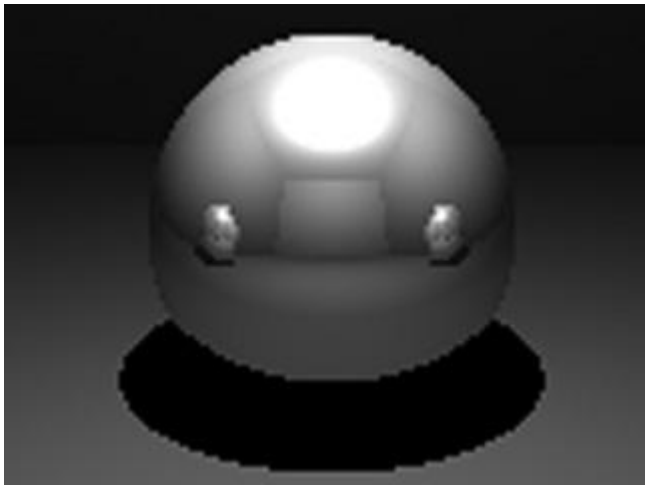
TEXTURE FILTERING

Texture filtering is algorithms used by graphics hardware that affect how the mapped image's contents appear on the surface. They are commonly supported by graphics hardware. By using the right texture filtering, you can improve the quality of textured mapped surfaces. Because images are not displayed to the rendered screen at a 1:1 ratio, artifacts can appear on textured mapped surfaces as objects move away from the camera, close to the camera, or are tilted at an angle. This means that every screen pixel is not shaded with an individual pixel in the image. As the surface with the texture moves away, a single screen pixel can actually have multiple image pixels fall within it. This can cause the images to distort slightly and display artifacts that can damage the rendered look. Among these are aliasing artifacts, which will be discussed in more detail later in this book.

The fastest filtering algorithm is called nearest-neighbor interpolation filtering, also commonly referred to as point-filtering. Point-filtering essentially selects the closest pixel to the point being sampled and uses that as the color. This is the fastest type of filtering because no additional equations need to be solved to compute the color. It simply uses the texture coordinates to find the closest pixel.

The second-fastest type of filtering is called bilinear interpolation, and the third-fastest is called trilinear interpolation. Bilinear interpolation averages each pixel with four surrounding pixels and displays the average instead of the original. This has the effect of slightly softening the images by using a very simple blur and helps reduce various artifacts such as aliasing artifacts, as shown in Figure 6.7. Trilinear interpolation does the same, but it also includes interpolation between mip maps (multi-resolution maps). Mip maps will be discussed in more detail in the upcoming section. We'll discuss how to specify hardware filtering later on in this chapter.

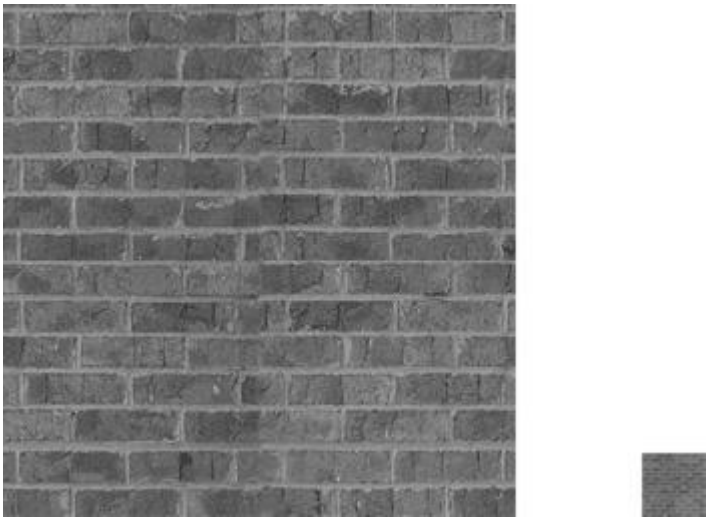
FIGURE 6.7. ALIASING ARTIFACTS.



MULTI-RESOLUTION MAPS

A mip map is a multi-resolution map of a texture image. Let's say you have a 2D texture that is 512 × 512. In computer graphics you can load the same texture at smaller resolutions and store them all in a single texture object so that the graphics hardware can choose which resolution of the image (hence multi-resolution map) to use. The reasoning for this is fairly straightforward. As objects move away from the camera, their high-resolution detail is not as visible as if you opened the image in an editor such as Adobe Photoshop or if you viewed the image up close. Therefore, if you have an image that is 1024 × 1024, and if that texture is being displayed on a surface that is so far away that the texture looks the way it would if it was 128 × 128, then why send the 1024 × 1024 image down the rendering pipeline? The larger the textures, the larger the bottleneck on the graphics hardware, along with other factors such as the amount of geometry. Figure 6.8 shows how detail from a far-away texture cannot be seen as easily.

FIGURE 6.8. DETAIL IS LOST AS THE OBJECT MOVES FAR AWAY.

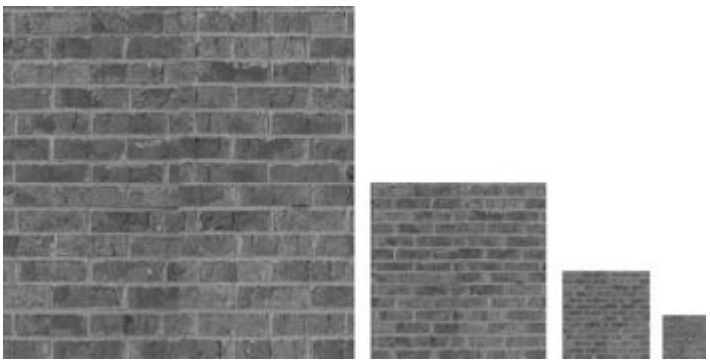


The purpose of mip maps is to reduce the amount of texture data that is processed and passed down the graphics hardware when rendering surfaces at a distance. When mip maps

are enabled, the graphics hardware performs all operations and chooses the best resolution to display surfaces that are rendered in a scene. So when using an API such as OpenGL or Direct3D, all you have to do is supply the API with the mip maps or tell the API to generate them if you don't have multiple resolutions. Reducing the amount of data that is passed down the graphics hardware can lead to better performance, and that is the purpose of mip maps.

Mip maps can be created by choosing a texture format that saves mip maps such as the .DDS image file format. The mip maps can be manually loading from individual images one at a time into a texture object, or they can be generated by the graphics hardware, which is an option OpenGL and Direct3D offer. Mip maps have resolutions that are a factor of 2, and every level of resolution is half the size as the one before it. So if you had a 1024 × 1024 texture and four levels of mip maps, the highest level would be 1024 × 1024, the second level would be 512 × 512, the third level would be 256 × 256, and the fourth level would be 128 × 128. An example of mip maps as they would look side-by-side is shown in Figure 6.9.

FIGURE 6.9. EXAMPLE OF MIP MAPS (512 × 512, 256 × 256, 128 × 128, 64 × 64).



LOADING TEXTURES

There are three main ways to create a texture in Direct3D 10. Programmers have the option of loading a texture from a file, loading a texture from memory, or generating one in a pixel shader. The generation of a texture in a shader is called procedural texture generation, and these textures are created using an algorithm.

Direct3D 10 offers a number of texture-related functions, each of which we'll briefly look at in this chapter. Throughout the rest of this book we focus on the function `D3DX10CreateShaderResourceViewFromFile()`. We will use this function to load 2D textures, but it also works on the other texture types discussed in the beginning of this chapter. The texture functions Direct3D offers include the following.

- `D3DX10CreateShaderResourceViewFromFile()`
- `D3DX10CreateShaderResourceViewFromMemory()`
- `D3DX10CreateShaderResourceViewFromResource()`
- `D3DX10CreateTextureFromFile()`
- `D3DX10CreateTextureFromMemory()`

- D3DX10CreateTextureFromResource()
- D3DX10LoadTextureFromTexture()
- D3DX10GetImageInfoFromFile()
- D3DX10GetImageInfoFromMemory()
- D3DX10GetImageInfoFromResource()
- D3DX10CreateAsyncTextureInfoProcessor()
- D3DX10CreateAsyncTextureProcessor()
- D3DX10FilterTexture()
- D3DX10ComputeNormalMap()
- D3DX10SaveTextureToFile()
- D3DX10SaveTextureToMemory()

To use textures we need a texture object and a shader resource view. The texture object is the texture itself, and the shader resource view is used to allow the shader to access the resource. The D3DX10CreateShaderResourceViewFromFile(), D3DX10CreateShaderResourceViewFromMemory(), and D3DX10CreateShaderResourceViewFromResource() functions are used to create a shader resource view along with a texture object. The result of calling this function is the shader resource view object that has the type ID3D10ShaderResourceView. If you use one of these three functions, you don't have to create and load the texture object (of type ID3D10Texture2D for 2D textures) separately because these functions will do all of the work of preparing the shader resource view and texture object for you. Later, when we free the texture, we will see that when using this function, we must use the shader resource view to obtain a pointer to the texture so that the texture can be freed from the shader resource view.

The D3DX10CreateShaderResourceViewFromFile() function takes as parameters the Direct3D device object, the name of the texture file to load, an optional D3DX10_IMAGE_LOAD_INFO structure that is used to specify the characteristics of the texture, a thread pump used by multi-threading applications (beyond the scope of this book), the address to the shader resource view that will be created as a result of this function, and an optional address that will store the return value of the function in a multi-threading application if the thread pump was created. The function's memory has slightly different parameters, offering the size and image pixel data instead of the file name. The resource version has a parameter that represents the resource's name rather than a texture file name. The D3DX10CreateShaderResourceViewFromFile() function is the most commonly used function in this book.

The D3DX10CreateTextureFromFile(), D3DX10CreateTextureFromMemory(), and D3DX10CreateTextureFromResource() functions are used to create a texture object but do not create the shader resource view object. In contrast, other functions, including the D3DX10CreateShaderResourceViewFromFile() function, create the shader resource view and internally set the texture object to it. If you call one of the shader resource view creation functions, you do not need to call any of these functions that directly

create the texture object since it is done automatically. These functions essentially create the texture without the shader resource view.

The `D3DX10LoadTextureFromTexture()` function is used to load a new texture from an existing texture and takes as parameters the source texture, a `D3DX10_TEXTURE_LOAD_INFO` descriptor, and an address at which to store the new texture.

The `D3DX10GetImageInfoFromFile()`, `D3DX10GetImageInfoFromMemory()`, and `D3DX10GetImageInfoFromResource()` functions are used to retrieve image information from a texture. This information is stored in a `D3DX10_IMAGE_INFO` object and includes the image's width, height, depth, size in bytes, total mip map levels, file format, color format, dimensions, and miscellaneous flags. The resource dimensions can be any of the following:

- `D3D10_RESOURCE_DIMENSION_UNKNOWN`
- `D3D10_RESOURCE_DIMENSION_BUFFER`
- `D3D10_RESOURCE_DIMENSION_TEXTURE1D`
- `D3D10_RESOURCE_DIMENSION_TEXTURE2D`
- `D3D10_RESOURCE_DIMENSION_TEXTURE3D`

The format can be any of the `D3D10FORMAT` types discussed in Chapter 3, "Rendering Geometry," and the miscellaneous flags can be `D3D10_RESOURCE_MISC_GENERATE_MIPS`, `D3D10_RESOURCE_MISC_SHARED`, or `D3D10_RESOURCE_MISC_TEXTURECUBE`.

The `D3DX10CreateAsyncTextureInfoProcessor()` and `D3DX10CreateAsyncTextureProcessor()` functions are used to create data processors that are used in multi-threaded applications to load a texture. This is an advanced topic that requires an understanding of multi-threading, which is beyond the scope of this book.

The next two texture functions are used to manipulate a texture that is already loaded. The `D3DX10FilterTexture()` function takes as parameters the texture object to filter, the mip map level of the original texture being filtered, and flags for the filter. The flags can be one of the following values:

- `D3DX10_DEFAULT`
- `D3DX10_FILTER_NONE`
- `D3DX10_FILTER_POINT` (nearest neighbor filtering)
- `D3DX10_FILTER_LINEAR` (bilinear filtering)
- `D3DX10_FILTER_TRIANGLE`
- `D3DX10_FILTER_BOX`
- `D3DX10_FILTER_MIRROR_U`

- D3DX10_FILTER_MIRROR_V
- D3DX10_FILTER_MIRROR_W
- D3DX10_FILTER_MIRROR (same as D3DX10_FILTER_MIRROR_U / D3DX10_FILTER_MIRROR_V / D3DX10_FILTER_MIRROR_W)
- D3DX10_FILTER_DITHER
- D3DX10_FILTER_DITHER_DIFFUSION
- D3DX10_FILTER_SRGB_IN
- D3DX10_FILTER_SRGB_OUT
- D3DX10_FILTER_SRGB (same as D3DX10_FILTER_SRGB_IN / D3DX10_FILTER_SRGB_OUT)

The second function is the D3DXComputeNormalMap(), which is used to take a texture and to convert it to a normal map image. This function will be discussed in more detail in Chapter 13 when bump and normal mapping is covered.

The last texture functions are used to save a texture to a file or to memory. The D3DX10SaveTextureToFile() function takes as parameters the texture object that is to be saved, a D3DX10_IMAGE_FILE_FORMAT description object, and the name of the file that will be created. The second texture-saving function is called D3DX10SaveTextureToMemory(), and it takes as parameters the texture object to be saved, the image format description, an out address to a ID3D10BLOG object that will store the texture in memory, and optional flags.

If you want to create the texture and shader resource view separately, you can call D3DX10CreateTextureFromFile() to create the texture, or you can call one of the other texture creation functions—for example, CreateShaderResourceView()—to create only the resource view. The CreateShaderResourceView() takes as parameters the resource (such as the ID3D10Texture2D texture that is created by calling the D3DX10CreateTextureFromFile() function), the D3D10_SHADER_RESOURCE_VIEW_DESC, which is the descriptor object that specifies the characteristics of the shader resource view object is being created, and the address at which to store the shader resource view as an ID3D10ShaderResourceView object.

APPLYING TEXTURES

Objects and surfaces are rendered in Direct3D 10 using effect shaders as discussed in Chapter 4, "Shader Model 4." The effect shaders themselves are objects of the ID3D10Effect type. Inside each shader there can be one or more techniques, which are essentially implementations of rendering effects.

To apply a texture to a technique so that a shader bound to that technique can access it, we need to create an ID3D10EffectShaderResourceVariable object. This variable will bind the application to the shader so that a value can be stored inside it and be accessed by the shaders. As discussed in Chapter 4, this is done by calling the technique object's GetVariableByName() function (or an equivalent access function) and calling AsShaderResource() on the returned object to gain access to the

ID3D10EffectShaderResourceVariable object that will be used to set the variable or in this case the texture.

Once access to the shader variable is obtained, we set the texture by calling the SetResource() function on the ID3D10EffectShaderResourceVariable object. For example, if the ID3D10EffectShaderResourceVariable object was named g_decalEffectVar, and if we had a texture object named g_decal, we could set it like so:

```
g_decalEffectVar->SetResource(g_decal);
```

When initially creating the ID3D10EffectShaderResourceVariable object, we create it after the shader has been initially loaded like so:

```
HRESULT hr = D3DX10CreateEffectFromFile("TextureMap.fx",
   NULL, NULL, "fx_4_0", shaderFlags, 0, g_d3dDevice, NULL,
NULL,
   &g_shader, NULL, NULL);

if(FAILED(hr))
   return false;

g_effect = g_shader->GetTechniqueByName("TextureMapping");

g_decalEffectVar = g_shader->GetVariableByName(
   "decal")->AsShaderResource();
```

When you access the shader variable, the name passed into GetVariableByName() must match the variable name that is defined inside the shader.

FREEING TEXTURES

To free a texture, you can call Release() on the ID3D10Texture2D object. This will work if you manually created the texture object directly, but if you loaded the texture from a file using a Direct3D 10 helper function such as D3DX10CreateShaderResourceViewFromFile(), then instead of an ID3D10Texture2D object, you would have an ID3D10ShaderResource object. Even with the shader resource view object, you still have to release the ID3D10Texture2D object it contains. To do this, if you don't have a pointer to the texture but have one to the shader resource view, you call the GetResource() function of the ID3D10ShaderResource object and pass to it an ID3D10Resource pointer that will point to the texture resource object.

Since ID3D10Resource is a base class of ID3D10Texture2D, you can call Release() on this returned base object to release the texture asset. You can then call Release() on the ID3D10ShaderResource object to release the shader resource view from memory. Keep in mind that releasing only the shader resource view will not release the texture object to which the shader resource view is attached, so this must be done as a separate task. An example of this is shown next, where g_decal is assumed to be the shader resource view that was created as a result of calling the D3DX10CreateShaderResourceViewFromFile() function.

```
if(g_decal)
{
    ID3D10Resource *pRes;
    g_decal->GetResource(&pRes);

    pRes->Release();
    g_decal->Release();
}
```

You only need to call `GetResource()` *to obtain a pointer to the texture object if you don't already have it. If you already have the* `ID3D10Texture2D` *object, you can call* `Release()` *on that and* `Release()` *on any shader resource view that uses it.*

IMPLEMENTING TEXTURE MAPPING

In this section we will discuss the chapter demos that perform texture mapping. The first demo is called Texture Mapping, and the second is called Multi Texture.

You can find both of these demos on the accompanying CD-ROM in the Chapter 6 folder.

2D TEXTURE MAPPING DEMO

The Texture Mapping demo builds off of the Transformations demo source code from Chapter 5, "Transformations," and adds the ability to texture-map the rendered surface. The demo starts by declaring a 2D vector in the vertex structure that will be used to hold the per-vertex texture coordinates. Also added to the global section is the `ID3D10ShaderResourceView` object, which will hold the texture, and an `ID3D10EffectShaderResourceVariable` object that will be used to allow the texture to be bound to a variable in the shader file. The remainder of the global section is comparable to the global section from the Transformations demo and is shown in Listing 6.1.

LISTING 6.1. THE GLOBAL SECTION FROM THE TEXTURE MAPPING DEMO

```
#include<d3d10.h>
#include<d3dx10.h>

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Texture Mapping"
#define WINDOW_CLASS    "UPGCLASS"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600
```

```
// Global window handles.
HINSTANCE g_hInst = NULL;
HWND g_hwnd = NULL;

// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

struct DX10Vertex
{
   D3DXVECTOR3 pos;
   D3DXVECTOR2 tex0;
};

ID3D10InputLayout *g_layout = NULL;
ID3D10Buffer *g_squareVB = NULL;
ID3D10ShaderResourceView *g_squareDecal = NULL;

ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_textureMapTech = NULL;
ID3D10EffectShaderResourceVariable *g_decalEffectVar = NULL;
ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;

D3DXMATRIX g_worldMat, g_viewMat, g_projMat;
```

In the `InitializeDemo()` function we add code to load the texture from a file using the Direct3DX function `D3DX10CreateShaderResourceViewFromFile()`, which loads the texture and creates a shader resource view in one call. Access to the shader variable is obtained by calling `GetVariableByName()` and sending to it the name of the texture in the shader file, which in this demo is `decal`. The result of the `GetVariableByName()` function call returns a base object that we can call `AsShaderResource()` to get a pointer to the object using the correct object type. A partial look at the top half of the `InitializeDemo()` function is shown in Listing 6.2, where only a few lines were added to support loading the texture image. Listing 6.3 shows the remainder of the `InitializeDemo()` function, where the surface was modified from a triangle shape to a square. Also, each vertex has a texture coordinate set attached to it.

LISTING 6.2. THE FIRST HALF OF THE `INITIALIZEDEMO()` FUNCTION

```
bool InitializeDemo()
{
   // Load the shader.

   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG )  || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif
```

```
   ID3D10Blob *errors = NULL;

  HRESULT hr = D3DX10CreateEffectFromFile(
     "TextureMapDemoEffects.fx", NULL, NULL, "fx_4_0",
     shaderFlags, 0, g_d3dDevice, NULL, NULL, &g_shader,
&errors,
     NULL);

   if(errors != NULL)
   {
      MessageBox(NULL, (LPCSTR)errors->GetBufferPointer(),
                 "Error in Shader!", MB_OK);

      errors->Release();
   }

   if(FAILED(hr))
      return false;

   g_textureMapTech = g_shader->GetTechniqueByName(
     "TextureMapping");

   g_worldEffectVar = g_shader->GetVariableByName(
     "World")->AsMatrix();

   g_viewEffectVar = g_shader->GetVariableByName(
     "View")->AsMatrix();

   g_projEffectVar = g_shader->GetVariableByName(
     "Projection")->AsMatrix();

   g_decalEffectVar = g_shader->GetVariableByName(
     "decal")->AsShaderResource();


   // Load the texture.
   hr = D3DX10CreateShaderResourceViewFromFile(g_d3dDevice,
      "brick.dds", NULL, NULL, &g_squareDecal, NULL);

   if(FAILED(hr))
      return false;

   …
}
```

LISTING 6.3. THE SECOND HALF OF THE `INITIALIZEDEMO()` FUNCTION

```
bool InitializeDemo()
{
   …
```

```
   // Create the geometry.

   D3D10_INPUT_ELEMENT_DESC layout[] =
{
   { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
     D3D10_INPUT_PER_VERTEX_DATA, 0 },
   { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
     D3D10_INPUT_PER_VERTEX_DATA, 0 },
};

unsigned int numElements = sizeof(layout) / sizeof(layout[0]);
D3D10_PASS_DESC passDesc;

g_textureMapTech->GetPassByIndex(0)->GetDesc(&passDesc);

hr = g_d3dDevice->CreateInputLayout(layout, numElements,
   passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
   &g_layout);

if(FAILED(hr))
   return false;

DX10Vertex vertices[] =
{
   { D3DXVECTOR3( 0.5f,  0.5f, 1.5f), D3DXVECTOR2(1.0f, 0.0f) },
   { D3DXVECTOR3( 0.5f, -0.5f, 1.5f), D3DXVECTOR2(1.0f, 1.0f) },
   { D3DXVECTOR3(-0.5f, -0.5f, 1.5f), D3DXVECTOR2(0.0f, 1.0f) },
   { D3DXVECTOR3(-0.5f, -0.5f, 1.5f), D3DXVECTOR2(0.0f, 1.0f) },
   { D3DXVECTOR3(-0.5f,  0.5f, 1.5f), D3DXVECTOR2(0.0f, 0.0f) },
   { D3DXVECTOR3( 0.5f,  0.5f, 1.5f), D3DXVECTOR2(1.0f, 0.0f) }
};


// Create the vertex buffer.
   D3D10_BUFFER_DESC buffDesc;
   buffDesc.Usage = D3D10_USAGE_DEFAULT;
   buffDesc.ByteWidth = sizeof(DX10Vertex) * 6;
   buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
   buffDesc.CPUAccessFlags = 0;
   buffDesc.MiscFlags = 0;

   D3D10_SUBRESOURCE_DATA resData;
   resData.pSysMem = vertices;

   hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
      &g_squareVB);

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

```
    // Set the shader matrix variables that won't change once
here.
    D3DXMatrixIdentity(&g_worldMat);
    D3DXMatrixIdentity(&g_viewMat);

    g_viewEffectVar->SetMatrix((float*)&g_viewMat);
    g_projEffectVar->SetMatrix((float*)&g_projMat);

    return true;
}
```

In the `RenderScene()` function, one line of code was added to bind the texture object to the shader variable. This is done by calling the `SetResource()` function of the `ID3D10EffectShaderResourceVariable` object that is bound to the variable and passing to its parameter the shader resource view texture object. The texture must be set before rendering occurs so that the shaders that use the texture have access to it and its image contents. The `RenderScene()` function is shown in Listing 6.4.

LISTING 6.4. THE `RENDERSCENE()` FUNCTION FROM THE TEXTURE MAPPING DEMO

```
void RenderScene()
{
    float col[4] = { 0, 0, 0, 1 };

    g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

    g_worldEffectVar->SetMatrix((float*)&g_worldMat);
    g_decalEffectVar->SetResource(g_squareDecal);

    unsigned int stride = sizeof(DX10Vertex);
    unsigned int offset = 0;

    g_d3dDevice->IASetInputLayout(g_layout);
    g_d3dDevice->IASetVertexBuffers(0, 1, &g_squareVB, &stride,
                                    &offset);
    g_d3dDevice->IASetPrimitiveTopology(
       D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    D3D10_TECHNIQUE_DESC techDesc;
    g_textureMapTech->GetDesc(&techDesc);

    for(unsigned int i = 0; i < techDesc.Passes; i++)
    {
       g_textureMapTech->GetPassByIndex(i)->Apply(0);
       g_d3dDevice->Draw(6, 0);
    }

    g_swapChain->Present(0, 0);
}
```

In the `Shutdown()` function, to free the texture, we call the `GetResource()` function on the shader resource view to gain access to the `ID3D10Texture2D` object. With a pointer to this object, we can free its contents by calling its `Release()` function. Once the texture is released, we can release the shader resource view object as well. This is shown in Listing 6.5.

LISTING 6.5. THE `SHUTDOWN()` FUNCTION FROM THE TEXTURE MAPPING DEMO

```
void Shutdown()
{
   if(g_d3dDevice) g_d3dDevice->ClearState();
   if(g_swapChain) g_swapChain->Release();
   if(g_renderTargetView) g_renderTargetView->Release();

   if(g_shader) g_shader->Release();
   if(g_layout) g_layout->Release();
   if(g_squareVB) g_squareVB->Release();

   if(g_squareDecal)
   {
      ID3D10Resource *pRes;
      g_squareDecal->GetResource(&pRes);

      pRes->Release();
      g_squareDecal->Release();
   }

   if(g_d3dDevice) g_d3dDevice->Release();
}
```

The last file to look at is the HLSL effect file for the Texture Mapping demo. In this file a 2D texture is defined in the global section so that the shaders can have access to it. This object has the HLSL data type `Texture2D`. A sampler state object of type `SamplerState` is used to sample a pixel from the texture. When creating a `SamplerState` object, you can specify the filtering mode, address mode (should it repeat, not repeat, etc.), border color, minimum and maximum level of detail in the image, and maximum anisotropy, which deals with filtering quality.

In the Texture Mapping demo we set the `Filter` state to `MIN_MAG_MIP_LINEAR`, which sets the min (surfaces that are minimized), mag (surfaces that are magnified), and mip map layers to linear. Linear interpolation applied to all three of these areas is known as trilinear filtering. The min filter is the filter used on the image when the image is drawn on a surface that is smaller than the original size of the image (i.e., the image is not drawn to scale). The mag filter is used when images are drawn on surfaces that are larger than the image's original size and thus need to be magnified. Mip filtering occurs on the mip map levels. Remember, trilinear filtering is bilinear filtering with an additional interpolation taking place between the mip maps.

The other states that are set are the `AddressU` and `AddressV` states, and they are set to wrap. Wrap means that if the texture coordinate for the U or V is over 1.0 or under 0.0, the texture will wrap around the surface.

The entire shader file used in the Texture Mapping demo is shown in Listing 6.6. In the vertex shader we pass along the texture coordinates to the output without performing any

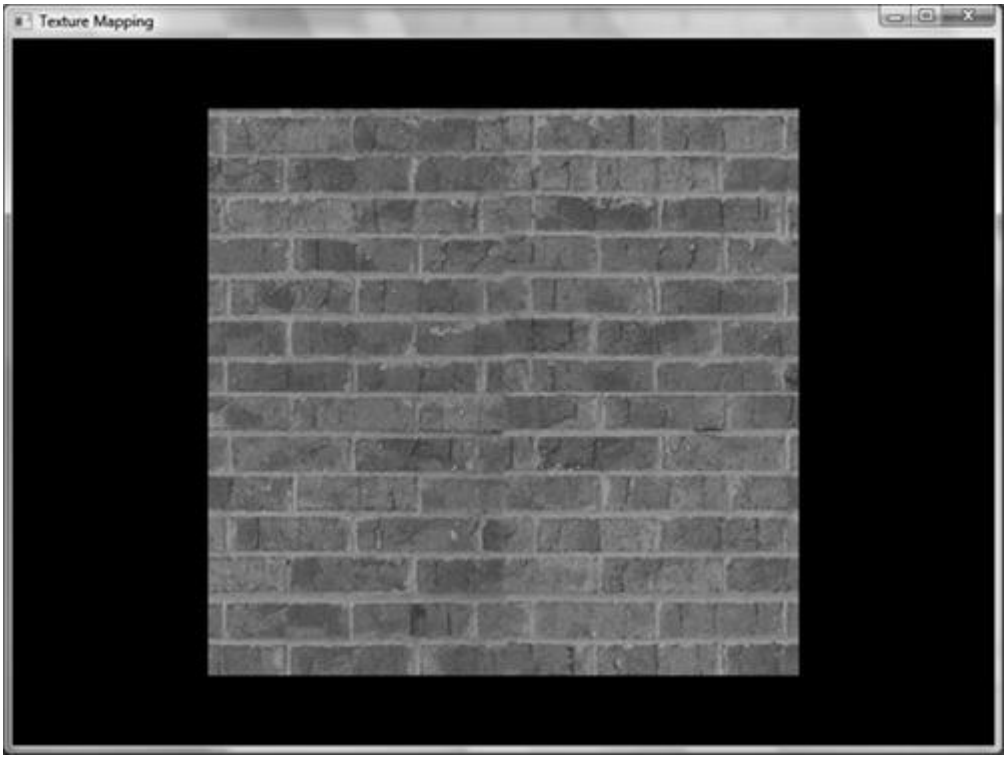
additional work on them because nothing needs to be done to prepare the texture coordinates for the pixel shader. In the pixel shader we call the `Sample()` function on the 2D texture object, and we send to it the sampler state and the vertex's texture coordinate. Keep in mind that the sampler state tells the graphics hardware what properties you want to have on the texture (e.g., filtering etc.) when it is sampled. The return result is a color value that we can use as the output of the pixel shader. The final result is a textured surface as shown in the screenshot in Figure 6.10.

LISTING 6.6. THE TEXTURE MAPPING DEMO'S SHADER FILE

```
Texture2D decal;

SamplerState DecalSampler
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};

cbuffer cbChangesEveryFrame
{
   matrix World;
   matrix View;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};

struct VS_INPUT
{
   float4 Pos : POSITION;
   float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
   float2 Tex : TEXCOORD0;
};


PS_INPUT VS(VS_INPUT input)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = mul(input.Pos, World);
   output.Pos = mul(output.Pos, View);
   output.Pos = mul(output.Pos, Projection);

   output.Tex = input.Tex;

   return output;
```

```
}
float4 PS(PS_INPUT input)  : SV_Target
{
   return decal.Sample(DecalSampler, input.Tex);
}


technique10 TextureMapping
{
   pass P0
   {
      SetVertexShader(CompileShader(vs_4_0, VS()));
      SetGeometryShader(NULL);
      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

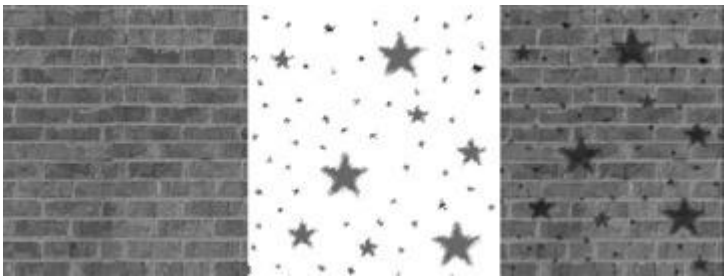
FIGURE 6.10. SCREENSHOT FROM THE TEXTURE MAPPING DEMO.



MULTI TEXTURE DEMO

Multi-texturing is a technique that is used to display multiple textures on one surface. In Direct3D 10 this can be done by loading a second texture image and sending it to the pixel shader. Inside the pixel shader both shaders are sampled, and the results are blended together. How you blend them is up to you. You can multiply the colors together, add them, interpolate between them, and so forth. An example of multi-texturing is shown in Figure 6.11. The demo that performs multi-texturing on the CD-ROM is called the Multi Texture demo and can be found in the Chapter 6 folder.

FIGURE 6.11. TEXTURE A (LEFT), TEXTURE B (MIDDLE), AND A AND B MULTI-TEXTURED (RIGHT).



In the Multi Texture demo's shader file we declare a second `Texture2D` object. Inside the pixel shader we sample from both shaders using the same sampler state and the same texture coordinates. This is shown in Listing 6.7. If you wanted, you could use different sampler states and even different texture coordinates. In the pixel shader we combine colors by multiplying them together. Since colors are in the range of 0.0 to 1.0, two white colors will be white, two black colors will be black, and anything in between will be a blend. We could also have added them together to get a different effect, subtracted, linearly interpolated, and so forth.

LISTING 6.7. THE SHADER FROM THE MULTI TEXTURE DEMO

```
Texture2D decal1;
Texture2D decal2;

SamplerState DecalSampler
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};

cbuffer cbChangesEveryFrame
{
   matrix World;
   matrix View;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};

struct VS_INPUT
{
   float4 Pos : POSITION;
   float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
   float2 Tex : TEXCOORD0;
};
```

```
PS_INPUT VS(VS_INPUT input)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = mul(input.Pos, World);
   output.Pos = mul(output.Pos, View);
   output.Pos = mul(output.Pos, Projection);

   output.Tex = input.Tex;

   return output;
}

float4 PS(PS_INPUT input)  : SV_Target
{
   return decal1.Sample(DecalSampler, input.Tex) *
          decal2.Sample(DecalSampler, input.Tex);
}


technique10 MultiTextureMapping
{
   pass P0
   {
      SetVertexShader(CompileShader(vs_4_0, VS()));
      SetGeometryShader(NULL);
      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

The source code from the Multi Texture demo builds directly off of the code from the Texture Mapping demo. In the global section another shader resource view and shader resource variable were added for the second texture. This addition is shown in Listing 6.8. In the `InitializeDemo()` function the second texture is loaded along with binding to the shader variable `decal2`, which is shown in Listing 6.9.

LISTING 6.8. THE ALTERED GLOBALS IN THE MULTI TEXTURE DEMO FROM TEXTURE MAPPING

```
ID3D10ShaderResourceView *g_squareDecal1 = NULL;
ID3D10ShaderResourceView *g_squareDecal2 = NULL;

ID3D10EffectShaderResourceVariable *g_decalEffectVar1 = NULL;
ID3D10EffectShaderResourceVariable *g_decalEffectVar2 = NULL;
```

LISTING 6.9. THE ALTERED `INITIALIZEDEMO()` FUNCTION FROM THE MULTI TEXTURE DEMO.

```
bool InitializeDemo()
{
   …
```

```
    g_textureMapTech = g_shader->GetTechniqueByName(
       "MultiTextureMapping");

    g_worldEffectVar = g_shader->GetVariableByName(
       World")->AsMatrix();

    g_viewEffectVar = g_shader->GetVariableByName(
       "View")->AsMatrix();

    g_projEffectVar = g_shader->GetVariableByName(
       "Projection")->AsMatrix();

    g_decalEffectVar1 = g_shader->GetVariableByName(
       "decal1")->AsShaderResource();

    g_decalEffectVar2 = g_shader->GetVariableByName(
       "decal2")->AsShaderResource();
    // Load the textures.
    hr = D3DX10CreateShaderResourceViewFromFile(g_d3dDevice,
      "brick.dds", NULL, NULL, &g_squareDecal1, NULL);

    if(FAILED(hr))
       return false;

    hr = D3DX10CreateShaderResourceViewFromFile(g_d3dDevice,
       "wavystars.dds", NULL, NULL, &g_squareDecal2, NULL);

    if(FAILED(hr))
       return false;

    …
}
```

The last modified code in the Multi Texture demo from the Texture Mapping demo can be seen in the `RenderScene()` and `Shutdown()` functions. In the `RenderScene()` function a new line of code is added to send the second texture to the `decal2` shader variable. In the `Shutdown()` function the second shader resource view and its texture object are released from memory directly under the first texture. Both of these functions are shown in Listing 6.10. A screenshot of the application is shown in Figure 6.12.

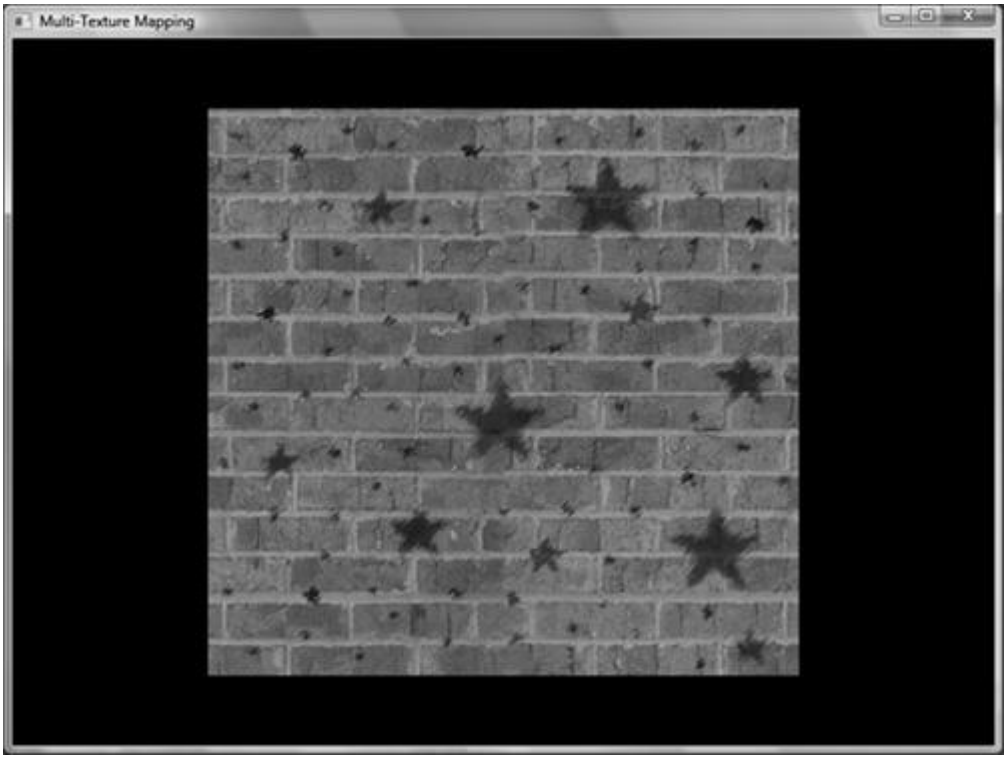
LISTING 6.10. THE `RENDERSCENE()` AND `SHUTDOWN()` FUNCTIONS FROM THE MULTI TEXTURE DEMO

```
void RenderScene()
{
    float col[4] = { 0, 0, 0, 1 };

    g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

    g_worldEffectVar->SetMatrix((float*)&g_worldMat);
    g_decalEffectVar1->SetResource(g_squareDecal1);
```

```
   g_decalEffectVar2->SetResource(g_squareDecal2);
   unsigned int stride = sizeof(DX10Vertex);
   unsigned int offset = 0;

   g_d3dDevice->IASetInputLayout(g_layout);
   g_d3dDevice->IASetVertexBuffers(0, 1, &g_squareVB,
      &stride, &offset);
   g_d3dDevice->IASetPrimitiveTopology(
      D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   D3D10_TECHNIQUE_DESC techDesc;
   g_textureMapTech->GetDesc(&techDesc);

   for(unsigned int i = 0; i < techDesc.Passes; i++)
   {
      g_textureMapTech->GetPassByIndex(i)->Apply(0);
      g_d3dDevice->Draw(6, 0);
   }

   g_swapChain->Present(0, 0);
}


void Shutdown()
{
   if(g_d3dDevice) g_d3dDevice->ClearState();
   if(g_swapChain) g_swapChain->Release();
   if(g_renderTargetView) g_renderTargetView->Release();

   if(g_shader) g_shader->Release();
   if(g_layout) g_layout->Release();
   if(g_squareVB) g_squareVB->Release();

   if(g_squareDecal1)
   {
      ID3D10Resource *pRes;
      g_squareDecal1->GetResource(&pRes);

      pRes->Release();
      g_squareDecal1->Release();
   }

   if(g_squareDecal2)
   {
      ID3D10Resource *pRes;
      g_squareDecal2->GetResource(&pRes);

      pRes->Release();
      g_squareDecal2->Release();
   }

   if(g_d3dDevice) g_d3dDevice->Release();
```

```
}
```

FIGURE 6.12. SCREENSHOT FROM THE MULTI TEXTURE DEMO.



ADDITIONAL TEXTURING TOPICS

There are a few additional texturing techniques to discuss in this chapter. A tremendous number of texture-based techniques can be performed in computer graphics. In this section we will examine a few very common techniques to give you an idea of what else can be done with textures. Throughout this book we are looking at additional techniques such as bump mapping, shadow mapping, and so forth.

MANUALLY LOADING AND GENERATING TEXTURES

There might come a time when you do not want to use Direct3D functions to load your textures, but instead you want to do so manually. Loading your own data into a texture object is fairly straightforward, and in this section you will see how to do it by calling the `CreateTexture2D()` function to create the 2D texture object and the `CreateShaderResourceView()` to create the shader resource view. This discussion will be kept brief since manually loading textures essentially requires two function calls.

Keep in mind that this information and the information throughout this chapter apply to all types of textures. We are using 2D textures simply as examples, but it doesn't matter what type of texture you are working with.

So far, we have loaded images from files using D3DX utility functions. If you wanted to manually place color data into an ID3D10Texture2D object, you would need a texture description of the type D3D10_TEXTURE2D_DESC and a subresource description of type D3D10_SUBRESOURCE_DATA. Once you've created the ID3D10Texture2D object, you can create the shader resource view and use the texture as normal. The D3D10_TEXTURE2D_DESC object represents the characteristics of the texture being created such as its width, height, and size in bytes. An example of creating and filling in such an object is as follows.

```
D3D10_TEXTURE2D_DESC textureDesc;

textureDesc.ArraySize = 1;
textureDesc.BindFlags = D3D10_BIND_SHADER_RESOURCE;
textureDesc.Usage = D3D10_USAGE_DYNAMIC;
textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.Width = image_width;
textureDesc.Height = image_height;
textureDesc.MipLevels = 1;
textureDesc.SampleDesc.Count = 1;
textureDesc.SampleDesc.Quality = 0;
textureDesc.CPUAccessFlags = D3D10_CPU_ACCESS_WRITE;
textureDesc.MiscFlags = 0;
```

The width and height from the texture description is the image's resolution. MipLevels is the number of mip maps contained in the data. ArraySize is the number of textures created by the CreateTexture() function call. Format is the color format of the image. SampleDesc describes the image's multi-sampling (discussed later in this chapter). Usage describes how the image will be read or written to. BindFlags are flags describing how the data will be used in the rendering pipeline. CPUAccessFlags determines if the CPU can read or write to the texture. MiscFlags can be any of the miscellaneous flags that were discussed earlier in this chapter.

The CPUAccessFlags can be D3D10_CPU_ACCESS_READ, D3D10_CPU_ACCESS-WRITE, or both using the logical OR operator.

The UsageFlags can be D3D10_USAGE_DEFAULT (the resource uses read and write operations), D3D10_USAGE_IMMUTABLE (the resource can only be read by the GPU), D3D10_USAGE_DYNAMIC (the resource can be read by the GPU and written by the CPU), and D3D10_USAGE_STAGING (the resource can have data transfer from the GPU to the CPU). You can combine flags using the logical OR operator as long as the flags do not conflict with each other. For example, you cannot use D3D10_USUAGE_IMMUTABLE, which says the only read access can occur using the GPU, with another type that allows writing by the GPU or read/write by the CPU.

The BindFlags can be any of the following:

- D3D10_BIND_VERTEX_BUFFER if we are creating a vertex buffer
- D3D10_BIND_INDEX_BUFFER if it's an index buffer
- D3D10_BIND_CONSTANT_BUFFER for constant buffers

- `D3D10_BIND_SHADER_RESOURCE` for shader resources
- `D3D10_BIND_STREAM_OUTPUT` if the object is to be used for output
- `D3D10_BIND_RENDER_TARGET` for rendering targets
- `D3D10_BIND_DEPTH_STENCIL` for binding a texture as a depth and stencil buffer output

With the texture description object filled in, the next object you will need is the subresource description. In the `D3D10_SUBRESOURCE_DATA` structure we can set the `pSysMem` variable, which will hold the actual image data, the `SysMemPitch` variable, which represents the number of bytes for a row of image data (width × the number of components, where RGB would be 3 and RGBA would be 4), and the `SysMemSlicePitch` variable, which is the depth of the image multiplied by the number of components for each pixel. The `SysMemSlicePitch` variable is only used for 3D textures. An example of creating and filling in a `D3D10_SUBRESOURCE_DATA` object and calling `CreateTexture2D()` to create a 2D texture is shown as follows. `image_data` is assumed to be an array of bytes, and `image_width` is the width of the image.

```
D3D10_SUBRESOURCE_DATA resData;
resData.pSysMem = (void*)image_data;
resData.SysMemPitch = image_width * 4;
resData.SysMemSlicePitch = 0;

ID3D10Texture2D *texture;

hr = g_d3dDevice->CreateTexture2D(&textureDesc, &resData,
&texture);
```

Keep in mind that the number of components depends on the format type you've specified when creating the texture descriptor. With the `ID3D10Texture2D` object created, you can then create a shader resource view by calling `CreateShaderResourceView()`. Once you have the shader resource view, you can use the texture as normal. An example of creating a shader resource view is shown as follows, where `Format` is the texture's format, `MipLevels` is the total number of mip maps, `MostDetailedMip` is the mip map level with the largest resolution, and `ViewDimension` describes the resource type:

```
D3D10_SHADER_RESOURCE_VIEW_DESC svDesc;

svDesc.Format = textureDesc.Format;
svDesc.Texture2D.MipLevels = 1;
svDesc.Texture2D.MostDetailedMip = 0;
svDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURE2D;

ID3D10ShaderResourceView *shaderResourceView = NULL;
g_d3dDevice->CreateShaderResourceView(texture, &svDesc,
                                      &shaderResourceView);
```

COMPRESSED TEXTURES

Compression is a term used to refer to algorithms that take a source data and represent it using fewer bytes. The two main types of compression are lossless and lossy.

Lossless compression does not affect the original quality. If you view something with lossless compression, it not only has the same quality as the original, but re-compression shouldn't degrade the quality of the data.

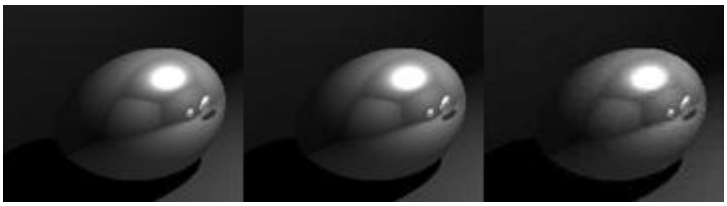
For a simple example of lossless compression, let's say you have an image with 100 pixels made up of only 10 unique colors. The original image would be 300 bytes if you assume 3 bytes per-pixel. If you place those 10 colors in an array of RGB values, you need 30 bytes to store the unique colors. If you used 1-byte array indexes for each pixel of the image, you would need 130 bytes to store the image (100 for the pixel indexes and 30 for the array of unique colors). This is a difference of 170 bytes. If you used 4 bits for the array index instead of a byte (where four bits can store a range from 0 to 16, which would be more than enough for a 0 to 9 array index), it would take 30 bytes for the unique colors array and 40 bytes for the image. This means you could have taken a 300-byte image and reduced it to 70 bytes using lossless compression.

You can perfectly re-create the original image using this data, which means quality wasn't reduced. If you used fewer bits for the indexes, that would not be enough to represent the entire index range. For example, three bits (0 to 7) caused you to have to choose to drop some colors since the number of bits cannot be used to index every element in the array. You could then choose to clamp the colors where the highest index in the array is used for all pixels that originally referenced colors above that element. For example, you can drop the last two colors, and any image pixels that use those values can use the eighth (index 7) color instead. This is lossy compression: the data is reduced in a way that cannot be decompressed perfectly to match the original source since those values were dropped and replaced. The replaced data can change the image so that under close examination it is clearly not accurate.

Lossy compression compresses data by altering it so that compressed data does not retain its original quality. Lossy compression works by sacrificing quality and accuracy for file size. If you recompress data that is already lossy compressed, the quality will suffer even more because the data being compressed is not the original data but is data that already has reduced quality. Therefore, if you need to recom-press data, it is best to only compress the original data rather than trying to compress data that is already lossy compressed.

Take a JPEG image, for example, which uses lossy compression. The quality decreases as the compression ratio increases. The higher the compression ratio that is used, the smaller the file size and thus the worse the image quality for JPEG images. This is illustrated in Figure 6.13, which shows the original image, the compressed version, and the image recompressed from an already compressed version.

FIGURE 6.13. ORIGINAL (LEFT), COMPRESSED (MIDDLE), AND RE-COMPRESSED FROM AN ALREADY COMPRESSED IMAGE (RIGHT).



Compression is an extremely important topic in video game development. For example, if you are able to compress all textures by a factor of four to one, making the textures 25% the size of the originals, you can reduce the disk space and memory requirements for your

game. This means that the amount of data that needs to be processed by the rendering pipeline is reduced, disk space is reduced, loading times are reduced since there is not as much information to load, texture streaming technology can work faster, and so forth.

Looking at the issue strictly from a graphics rendering perspective, this also means that the reduced data size allows developers to use four times as many textures in a game level (assuming all textures are the same size and you compressed them all by 25%). It could also mean that developers have more room to use textures that are four times larger in resolution since you reduced all textures by one-fourth.

All major graphics hardware for the past few generations has supported compressed textures. To use compressed textures in Direct3D, you do not need to enable anything or provide any special code. The only thing you need to do is save your images using a compressed file format. When Direct3D 10 loads the compressed images that the hardware supports, they are used directly in the graphics hardware. The graphics hardware itself handles the decompressing and returning the correct color value when a compressed texture is sampled. To create compressed textures you can save your textures in a tool such as Adobe Photoshop, or you can use the DirectX Texture tool that comes with the DirectX SDK.

Modern graphics hardware supports the S3 Texture Compression (S3TC) algorithms. There are five versions of these algorithms, DXT1 through DXT5. To save your images to one of these formats, you can save your .DDS images using Photoshop, the DirectX Texture tool, and so on and specifying one of these format types.

The DXT1 format uses four bits for each pixel in the image and is the format that can give the largest compression and the worst quality, depending on the image. The DXT1 format did not originally support an alpha channel, but there is an extended version that does. DXT1 offers eight-to-one compression without the alpha channel and six-to-one with the alpha channel.

The DXT2 and DXT3 formats use an additional four bits for an alpha channel. The DXT2 format's alpha is premultiplied with the color, while the DXT3 format has an explicit alpha that is not premultiplied. DXT2 and DXT3 offer a four-to-one compression ratio.

DXT4 uses an interpolated alpha channel that is premultiplied with the color. DTX5 is similar to DXT4 but without the premultiplication. DXT4 and DXT5 also offer a four-to-one compression ratio.

The DXT compression formats are great for color images but are not necessarily the best option for normal map images. In Chapter 13 we'll look at normal map compression formats such as 3Dc, which is used to retain much more quality than the DXT formats. We'll discuss why the DXT formats are not good to use for normal maps and why 3Dc is a great alternative.

When looking at compressed textures, keep in mind that there is nothing you need to do with Direct3D 10 to use them other than to save your textures using DXT1 through DXT5. Some images will look better with some formats than others, and usually you can experiment when saving textures to see which formats give you the best compression while retaining an acceptable level of quality. The 3Dc and S3TC (DXT) compression algorithms use lossy compression.

MULTI-SAMPLING

Multi-sampling is any technique where a texture is sampled more than once to find the averages of colors and decrease blocky artifacts in images. You can enable multi-sampling in the texture description so that when a texture is sampled in the pixel shader, instead of sampling a single color around the pixel to use, the hardware sends an average color value

based on the surrounding pixels. For example, if 2 × 2 multi-sampling is used to sample the pixel above, below, and to each side of each pixel, the color that is fetched is not the center (original) pixel but the average of all pixels that surround it plus itself.

Multi-sampling aims to smooth out color differences of nearby pixels. For example, in Figure 6.14 the color from one pixel to the next can change quite sharply. By averaging the neighboring pixels, you can blur these sharp edges so that the blocky staircase rendering artifacts do not show up or at least are not as noticeable. This is simply a blurring operation and in some cases can help improve the quality of textured surfaces.

FIGURE 6.14. ORIGINAL (TOP) CENTER PIXEL VERSUS AVERAGED (BOTTOM) CENTER PIXEL.

Multi-sampling is a hardware-accelerated technique, and to use it you simply enable it when creating textures in the texture description. Texture filtering such as bilinear, trilinear, and so on also averages pixels to create this effect of smoothing out sharp variations. You can also multi-sample the rendering targets so that the entire rendered scene is smoothed out a little.

Adaptive super-sampling is essentially multi-sampling that only occurs on pixels that require it, rather than all pixels in the image.

SUMMARY

Texture mapping is one of the most important topics in computer graphics. In this chapter we saw how to perform basic texture mapping in Direct3D 10 by defining texture coordinates to vertices, how to load a texture image from a file or manually place data in a texture object, how to make the texture accessible to the pixel shader, and how to free the texture resources once we are done with it.

Throughout the remainder of this book textures will be used in nearly every demo. Most of these demos will load a 2D texture from a file and display that image on various surfaces in the rendered scenes.

The following elements were discussed in this chapter:

- Types of textures

- Texture mapping in Direct3D 10
- Texture coordinates
- Texture filtering
- Mip maps
- Multi-texturing
- Compressed textures
- Multi-sampling

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What is the purpose of texture mapping?

2. List three ways textures can be used in computer graphics.

3. What are texture coordinates, and why are they necessary for texture mapping?

4. What is the purpose of texture filtering?

5. What is the min texture filter?

6. What is the mag texture filter?

7. What are mip maps?

8. Describe point filtering.

9. Describe bilinear filtering.

10. Describe trilinear filtering.

11. What are 1D textures?

12. What are 2D textures?

13. What are 3D textures?

14. What are cube maps? How do they differ from 3D textures?

15. What are sphere maps? How do they differ from 2D textures?

16. What is multi-texturing?

17. What is multi-sampling?

18. What are the S3TC texture compression formats? How do they differ from each other?

19. What is the difference between lossy and lossless compression?

20. Why is it a bad idea to compress data that is already compressed when using a lossy algorithm?

CHAPTER EXERCISES

Exercise 1: Modify the Multi Texture demo to add the textures instead of multiplying them.

Exercise 2: Modify the Multi Texture demo and add a third texture.

Exercise 3: Modify the Multi Texture demo and add another set of texture coordinates so that all of the textures use a different set when sampling in the pixel shader.

7. ADDITIONAL TEXTURE MAPPING

In This Chapter

- Alpha Mapping
- Sprites
- Image Filters

Texture mapping is a very important part of video game graphics. It is used to simulate complex detail on surfaces in real time. Texture mapping is used in many ways in computer graphics, including some of the following.

- Color maps
- Lighting (bump and normal mapping will be discussed in Chapter 13, "Lighting")
- Detail preservation (normal mapping)

- Look-up tables (i.e., storing any type of value for any purpose in a texture so that it can be passed to the graphics hardware and retrieved by a shader using texture sample functions)
- Gloss mapping (i.e., specular lighting using textures, also discussed in Chapter 13)
- Alpha mapping
- Environment maps (reflection mapping, refraction mapping, etc.)
- Displacement maps

The purpose of this chapter is to examine and implement a few additional texture-based graphical effects. So many graphical effects exist in computer graphics that the subject could span multiple books. In this chapter we will look at a few to give you an idea of the different ways textures can be used inside graphical applications such as video games that go beyond simply coloring a surface.

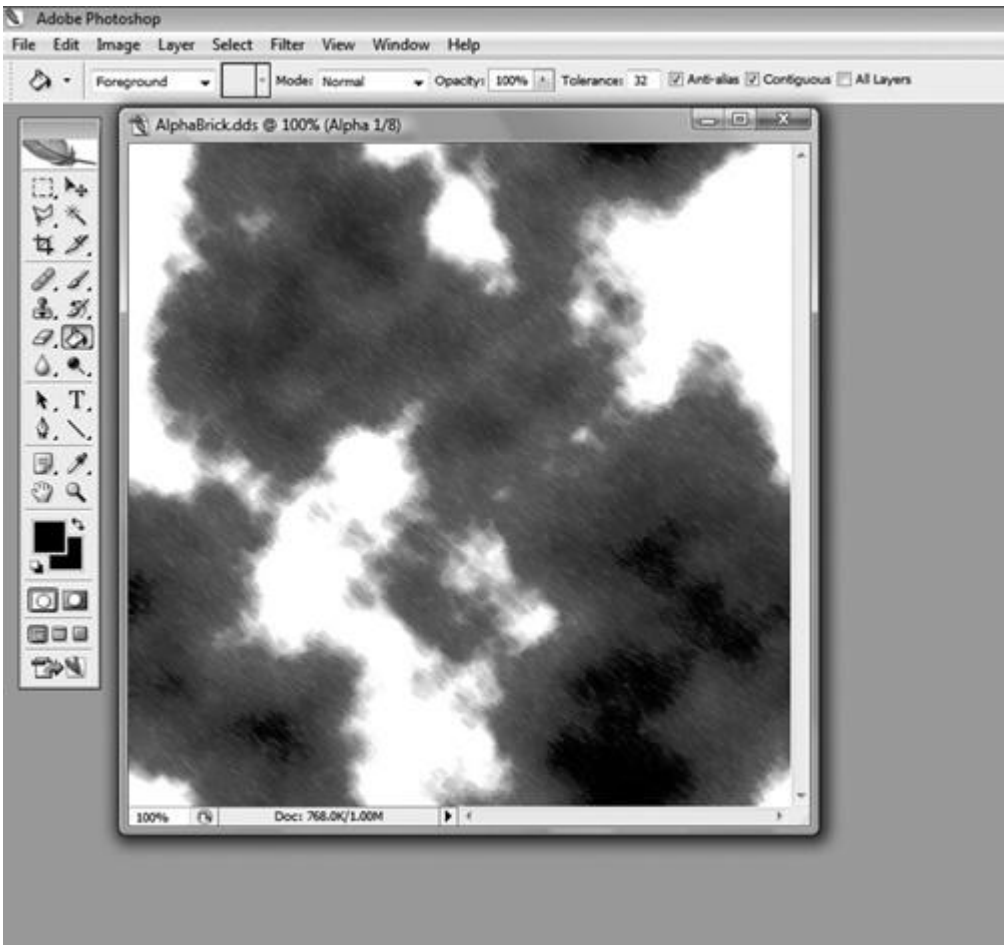
ALPHA MAPPING

Alpha mapping is a technique in which you use another image or the alpha channel of the decal image to specify areas of the image that are to be visible, transparent, and semitransparent. Therefore, an alpha map is used to specify the pixel-level transparency of a surface, while the color map specifies the pixel-level color values of a surface.

CREATING ALPHA MAPS

There are two ways to specify per-pixel alpha values. The first is to use the alpha channel of a 32-bit RGBA image. This can be done in any image editor, and the alpha values typically range between 0 for transparent and 255 for visible. Anything between 0 and 255 represents a level of transparency between the two extremes. In fact, you can consider 0 to be 0% and 255 to be 100% visible. An example of an alpha channel from a texture created in Adobe Photoshop is shown in Figure 7.1. If this alpha map was used, there would be many semi-transparent areas (any gray area in the alpha map), only a few fully visible areas, and even fewer fully transparent areas.

FIGURE 7.1. THE ALPHA MAP CHANNEL IN ADOBE PHOTOSHOP.



Another way to create an alpha map is to create a separate texture image. If you use an RGB image, these images are typically grayscale images (black and white). More specifically, it means the red, green, and blue components have equal values (e.g., R:200 G:200 B:200, R:123 G:123 B:123, etc.). When you use a grayscale image, it does not matter which color channel you use for the alpha value since it is the same value across the channels. If the image is not in black and white, it would be difficult to determine how the alpha map would affect the rendered object unless you were looking solely at one component at a time.

Creating an alpha map using a 24-bit RGB image is wasteful memory-wise if you are only using one component because the alpha channel is one byte; having RGBA images just to use that one channel means that the other channels (the RGB channels) are not used, which results in three wasted bytes per pixel. Therefore, it is better to use either a one-component image or the alpha channel of an RGBA image. That way each pixel is 1 byte instead of 3. If semitransparent pixels are not needed, you can replace that 1 byte per pixel with 1 bit, where 0 represents invisible and 1 represents visible (i.e., true or false).

ALPHA MAPPING DEMO

On the CD-ROM, in the Chapter 7 folder, is a demo called Alpha Mapping that demonstrates how to perform alpha mapping. The demo places the alpha values in the alpha channel of the image to avoid having to load a separate image, and the alpha map used is the one from Figure 7.1. The image in Figure 7.1 was copied into the alpha channel of the decal image in Photoshop and saved as an RGBA DXT5 texture.

Alpha mapping is done by using alpha blending. In Direct3D 10 this can be set on the application side or through HLSL. Throughout this chapter we will set states in HLSL, but if you were to do it on the application side, you would create a `D3D10_BLEND_DESC` object, set each of its member states (variables), and send it to Direct3D by calling `OMSetBlendState()`.

In HLSL a blend state can be created by creating a `BlendState` descriptor and naming it whatever you like. In the Alpha Mapping demo for this chapter, the `BlendState` descriptor is called `AlphaBlending`. Inside the descriptor we can set a host of related states that include the following.

- `AlphaToCoverageEnable`
- `BlendEnable`
- `SrcBlend`
- `DstBlend`
- `BlendOp`
- `SrcBlendAlpha`
- `DstBlendAlpha`
- `BlendOpAlpha`
- `RenderTargetWriteMask[n]`

Alpha to coverage is a term in computer graphics that deals with multisampling and refers to the way alpha-mapped surfaces are rendered in scenes that have many overlapping polygons. Alpha to coverage can be used even if the application is not using multisampling, in which case it is used to draw overlapping polygons that have transparency without the need for you to render them in a specific order to get the correct results.

The keyword `BlendEnable` enables or disables the blending feature, while `SrcBlend`, `SrcBlendAlpha`, `DstBlend`, and `DstBlendAlpha` set the blend options of the source 1 (SRC) and source 2 (DST) colors and alphas. The blend options can be one of the following. Note that the HLSL keywords are the same minus the `D3D10_BLEND_ part` of each option.

- `D3D10_BLEND_ZERO:` The data source is black.
- `D3D10_BLEND_ONE:` The data source is white.
- `D3D10_BLEND_SRC_COLOR:` The data source is the color from the pixel shader.
- `D3D10_BLEND_INV_SRC_COLOR:` The data source is 1 minus the color from the pixel shader.
- `D3D10_BLEND_SRC_ALPHA:` The data source is the alpha value from the pixel shader.

- `D3D10_BLEND_INV_SRC_ALPHA`: The data source is 1 minus the alpha value from the pixel shader.
- `D3D10_BLEND_DEST_ALPHA`: The data source is the alpha value from the rendering target.
- `D3D10_BLEND_INV_DEST_ALPHA`: The data source is 1 minus the alpha value from the rendering target.
- `D3D10_BLEND_DEST_COLOR`: The data source is the color value that is already stored in the rendering target.
- `D3D10_BLEND_INV_DEST_COLOR`: The data source is 1 minus the render target's color value.
- `D3D10_BLEND_SRC_ALPHA_SAT`: The data source is the clamped alpha value from the pixel shader.
- `D3D10_BLEND_BLEND_FACTOR`: The data source is the blend factor that was set by the Direct3D function `OMSetBlendState()` in the second parameter (the first parameter is the blend descriptor, and the last is the blend mask).
- `D3D10_BLEND_INV_BLEND_FACTOR`: The data source is 1 minus the blend factor set by the Direct3D function `OMSetBlendState()`.
- `D3D10_BLEND_SRC1_COLOR`: The data source is both color values outputted by the pixel shader (used in dual-source color blending).
- `D3D10_BLEND_INV_SRC1_COLOR`: The data source is 1 minus the color values from the pixel shader.
- `D3D10_BLEND_SRC1_ALPHA`: The same as `D3D10_BLEND_SRC1_COLOR` but using the alpha values from the pixel shader.
- `D3D10_BLEND_INV_SRC1_ALPHA`: One minus `D3D10_BLEND_SRC1_ALPHA`.

`BlendOp` and `BlendOpAlpha` from the blend state are used to set the blend operation. This means the two sources can be added using `D3D10_BLEND_OP_ADD`, subtracted using `D3D10_BLEND_OP_SUBTRACT` (or `D3D10_BLEND_OP_REV_SUB-TRACT`, which subtracts source 2 from source 1), set to the minimum of the two sources using `D3D10_BLEND_OP_MIN`, or set to the maximum of the two sources using `D3D10_BLEND_OP_MAX`.

The last member of the blend description, `RenderTargetWriteMask[n]`, is used to set the per-pixel write mask that is used to determine which components of the rendering target can be written to during rendering. A value of 0×0F can be used to specify that all components are to be written to. The value 0×0F is essentially the logical OR result of `D3D10_COLOR_WRITE_ENABLE_RED | D3D10_COLOR_WRITE_ENABLE_GREEN | D3D10_COLOR_WRITE_ENABLE_BLUE | D3D10_COLOR_WRITE_ENABLE_ALPHA`.

The shader from the Alpha Mapping demo is shown in Listing 7.1. The only difference between this shader and the Texture Mapping shader in Chapter 6, "Shading and Surfaces," is the addition of a `BlendState` descriptor. This object is set by calling the HLSL function `SetBlendState()`, which is the same as calling the Direct3D function

`OMSetBlendState()` on the application side. The function takes the blend state description, a blend factor, and a blend mask. The mask is set to 0xFFFFFF to allow for all components. Note that you can set the blend state either in HLSL, which is what we are doing in this demo, or on the application side using `OMSetBlendState()`. You don't have to do both.

LISTING 7.1. THE ALPHA MAPPING DEMO'S SHADER

```
   /*
      Alpha Mapping Demo's HLSL Shader
      Ultimate Game Programming with DirectX 2nd Edition
      Created by Allen Sherrod
   */


   Texture2D decal;

   SamplerState DecalSampler
   {
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU = Wrap;
      AddressV = Wrap;
   };


   BlendState AlphaBlending
   {
       AlphaToCoverageEnable = FALSE;
       BlendEnable[0] = TRUE;
       SrcBlend = SRC_ALPHA;
       DestBlend = INV_SRC_ALPHA;
       BlendOp = ADD;
       SrcBlendAlpha = ZERO;
       DestBlendAlpha = ZERO;
       BlendOpAlpha = ADD;
       RenderTargetWriteMask[0] = 0×0F;
   };

cbuffer cbChangesEveryFrame
   {
      matrix World;
      matrix View;
   };

   cbuffer cbChangeOnResize
   {
      matrix Projection;
   };

   struct VS_INPUT
   {
      float4 Pos : POSITION;
      float2 Tex : TEXCOORD;
```

```
   };

   struct PS_INPUT
   {
      float4 Pos : SV_POSITION;
      float2 Tex : TEXCOORD0;
   };


   PS_INPUT VS(VS_INPUT input)
   {
      PS_INPUT output = (PS_INPUT)0;

      output.Pos = mul(input.Pos, World);
      output.Pos = mul(output.Pos, View);
      output.Pos = mul(output.Pos, Projection);

      output.Tex = input.Tex;

      return output;
   }
float4 PS(PS_INPUT input) : SV_Target
{
   return decal.Sample(DecalSampler, input.Tex);
}

technique10 AlphaMapping
{
   pass P0
    {
      SetBlendState(AlphaBlending, float4(0.0f, 0.0f, 0.0f,
0.0f),
                    0xFFFFFFFF);

        SetVertexShader(CompileShader(vs_4_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, PS()));
    }
}
```

The Alpha Mapping demo builds off of the Texture Mapping demo in Chapter 6. In the Alpha Mapping demo the only difference is in the HLSL effect shader. The main source file from the demo is the same except for a few object name changes. The relevant code form the Alpha Mapping demo's main source file is shown in Listing 7.2. Figure 7.2 shows a screenshot from the demo.

LISTING 7.2. THE RELEVANT CODE IN THE ALPHA MAPPING DEMO'S SOURCE FILE

```
/*
   Alpha Mapping
   Ultimate Game Programming with DirectX 2nd Edition
   Created by Allen Sherrod
```

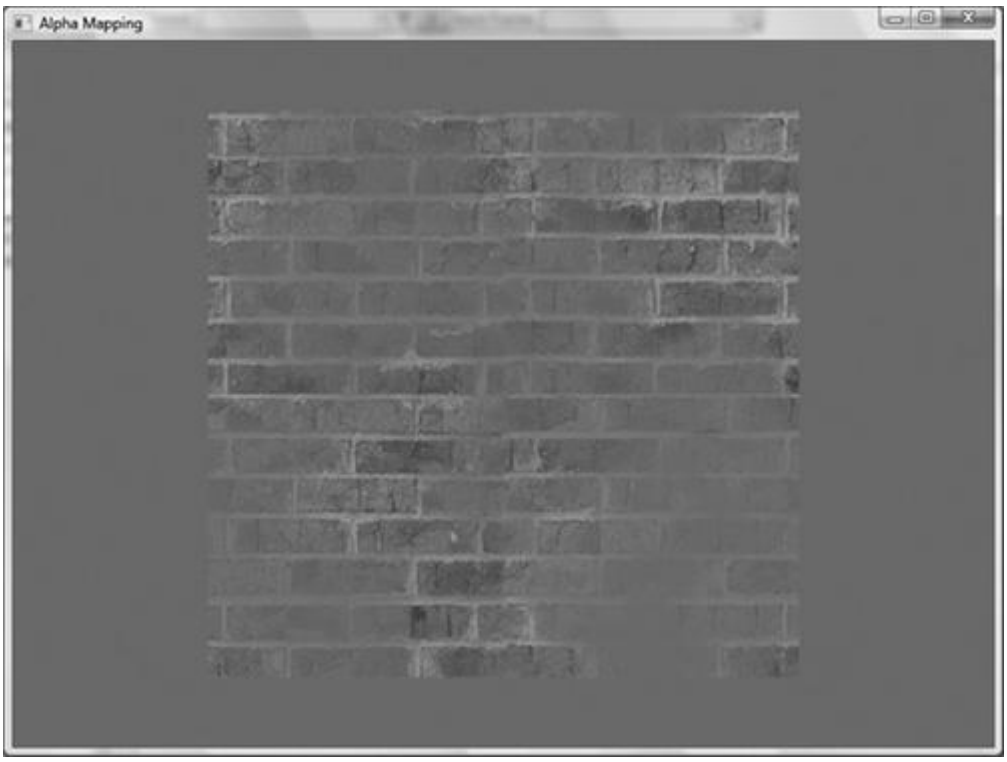
```
*/


#include<d3d10.h>
#include<d3dx10.h>
#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Alpha Mapping"
#define WINDOW_CLASS    "UPGCLASS"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600

// Global window handles.
HINSTANCE g_hInst = NULL;
HWND g_hwnd = NULL;

// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

struct DX10Vertex
{
   D3DXVECTOR3 pos;
   D3DXVECTOR2 tex0;
};

ID3D10InputLayout *g_layout = NULL;
ID3D10Buffer *g_squareVB = NULL;
ID3D10ShaderResourceView *g_squareDecal = NULL;

ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_alphaMapTech = NULL;
ID3D10EffectShaderResourceVariable *g_decalEffectVar = NULL;
ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;
D3DXMATRIX g_worldMat, g_viewMat, g_projMat;


bool InitializDemo()
{
   // Load the shader.

   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   HRESULT hr = D3DX10CreateEffectFromFile(
```

```
      "AlphaMapDemoEffects.fx", NULL, NULL, "fx_4_0",
shaderFlags,
      0, g_d3dDevice, NULL, NULL, &g_shader, NULL, NULL);

   if(FAILED(hr))
      return false;

   g_alphaMapTech = g_shader-
>GetTechniqueByName("AlphaMapping");

   g_worldEffectVar = g_shader->GetVariableByName(
      "World")->AsMatrix();

   g_viewEffectVar = g_shader->GetVariableByName(
      "View")->AsMatrix();

   g_projEffectVar = g_shader->GetVariableByName(
      "Projection")->AsMatrix();

   g_decalEffectVar = g_shader->GetVariableByName(
      "decal")->AsShaderResource();

   // Load the texture.
   hr = D3DX10CreateShaderResourceViewFromFile(g_d3dDevice,
      "AlphaBrick.dds", NULL, NULL, &g_squareDecal, NULL);

  if(FAILED(hr))
     return false;


  // Create the geometry.

  D3D10_INPUT_ELEMENT_DESC layout[] =
  {
     { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
       D3D10_INPUT_PER_VERTEX_DATA, 0 },
     { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
       D3D10_INPUT_PER_VERTEX_DATA, 0 },
  };

  unsigned int numElements = sizeof(layout) / sizeof(layout[0]);
  D3D10_PASS_DESC passDesc;

  g_alphaMapTech->GetPassByIndex(0)->GetDesc(&passDesc);

  hr = g_d3dDevice->CreateInputLayout(layout, numElements,
     passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
     &g_layout);

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

```
  DX10Vertex vertices[] =
  {
     { D3DXVECTOR3( 0.5f, 0.5f, 1.5f), D3DXVECTOR2(1.0f, 0.0f)
},
     { D3DXVECTOR3( 0.5f, -0.5f, 1.5f), D3DXVECTOR2(1.0f, 1.0f)
},
     { D3DXVECTOR3(-0.5f, -0.5f, 1.5f), D3DXVECTOR2(0.0f, 1.0f)
},
     { D3DXVECTOR3(-0.5f, -0.5f, 1.5f), D3DXVECTOR2(0.0f, 1.0f)
},
     { D3DXVECTOR3(-0.5f, 0.5f, 1.5f), D3DXVECTOR2(0.0f, 0.0f)
},
     { D3DXVECTOR3( 0.5f, 0.5f, 1.5f), D3DXVECTOR2(1.0f, 0.0f) }
  };

  // Create the vertex buffer.

  D3D10_BUFFER_DESC buffDesc;
  buffDesc.Usage = D3D10_USAGE_DEFAULT;
  buffDesc.ByteWidth = sizeof(DX10Vertex) * 6;
  buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
  buffDesc.CPUAccessFlags = 0;
  buffDesc.MiscFlags = 0;

  D3D10_SUBRESOURCE_DATA resData;
  resData.pSysMem = vertices;

  hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
                                 &g_squareVB);

  if(FAILED(hr))
     return false;

  // Set the shader matrix variables that won't change once
here.
  D3DXMatrixIdentity(&g_worldMat);
  D3DXMatrixIdentity(&g_viewMat);

  g_viewEffectVar->SetMatrix((float*)&g_viewMat);
  g_projEffectVar->SetMatrix((float*)&g_projMat);

  return true;
}
void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

   g_worldEffectVar->SetMatrix((float*)&g_worldMat);
   g_decalEffectVar->SetResource(g_squareDecal);
```

```
   unsigned int stride = sizeof(DX10Vertex);
   unsigned int offset = 0;

   g_d3dDevice->IASetInputLayout(g_layout);

   g_d3dDevice->IASetVertexBuffers(0, 1, &g_squareVB, &stride,
                                   &offset);

   g_d3dDevice->IASetPrimitiveTopology(
     D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   D3D10_TECHNIQUE_DESC techDesc;
   g_alphaMapTech->GetDesc(&techDesc);

   for(unsigned int i = 0; i < techDesc.Passes; i++)
   {
      g_alphaMapTech->GetPassByIndex(i)->Apply(0);
      g_d3dDevice->Draw(6, 0);
   }

   g_swapChain->Present(0, 0);
}

void Shutdown()
{
   if(g_d3dDevice) g_d3dDevice->ClearState();
   if(g_swapChain) g_swapChain->Release();
   if(g_renderTargetView) g_renderTargetView->Release();

   if(g_shader) g_shader->Release();
   if(g_layout) g_layout->Release();
   if(g_squareVB) g_squareVB->Release();

   if(g_squareDecal)
   {
      ID3D10Resource *pRes;
      g_squareDecal->GetResource(&pRes);

      pRes->Release()
      g_squareDecal->Release()
   }

   if(g_d3dDevice) g_d3dDevice->Release()
}
```

FIGURE 7.2. SCREENSHOT FROM THE ALPHA MAPPING DEMO.



SPRITES

A sprite is a 2D image displayed on the screen. Sprites are more common in 2D games, where the images act solely as the various game objects, characters, and environments. In 3D we usually have 3D geometry with texture images applied to them to simulate detail where none exists. In some 3D games, sprites can be seen as screen interface graphics such as health bars, shield bars, and so on.

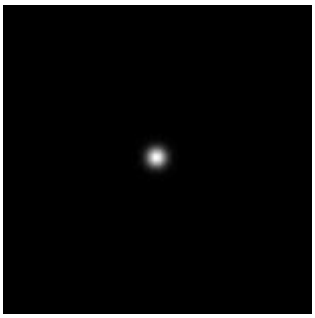
In the early days of 3D video games, sprites were used more like they are for 2D games, where sprite characters and objects populate the game world. This can be seen in early 3D games such as Id Software's *Doom, Duke Nuke 'Em 3D*, and many more. Today sprites are used mostly for particle effects such as snow, rain, smoke, dust, fire, weapon effects, and so forth.

TYPES OF SPRITES

There are two main types of sprites: sprites that can orient themselves in 3D space and those that cannot. The sprites that cannot orient in 3D are known as billboard sprites. In other words, a billboard sprite always faces the camera but can be positioned anywhere. Characters in the game *Doom* were billboard sprites; no matter how the player was oriented, the enemies and many world objects would always face the player.

The decision to use billboard sprites or not depends on the application. For most particle systems the particles are so small that changing their rotation might have a negative effect on the scene or, possibly, no effect at all. In 3D you can draw a textured square to act as a sprite. In this chapter we will create billboard sprites in Direct3D 10 using the sprite image in Figure 7.3 since learning that will give us something new to cover rather than drawing yet another textured square.

FIGURE 7.3. AN EXAMPLE OF A SPRITE.



To create a sprite you simply create a 2D image. If any part of the image is to be transparent or semitransparent, then you can create a 32-bit RGBA image to represent the object. In the image in Figure 7.3 the black areas are set to transparent in the alpha channel in Adobe Photoshop, while the orb itself is the only visible part of the texture image.

POINT SPRITES

Direct3D 9 and OpenGL have what are known as point sprites. A point sprite is essentially a sprite that is generated from a point and, in the case of Direct3D 9 and OpenGL, is also hardware accelerated. This means the hardware handles keeping the sprite facing the camera, and all the programmer has to do is enable the point sprite feature, specify the properties (such as size etc.), and supply a list of points to the graphics hardware.

Direct3D 10, however, does not have this point sprite feature. To use hardware-accelerated point sprites in Direct3D 10, we must use the geometry shader to create the sprites from a list of points. Since the geometry shader is doing the work, the effect is technically GPU hardware accelerated since no CPU processing is used for the creation of the geometry.

SPRITES DEMO

On the CD-ROM, in the Chapter 7 folder, is a demo called Billboard Sprites. The Billboard Sprites demo builds off of the Alpha Mapping demo and makes several modifications to the original source code.

In the shader effect file, a new global uniform variable is created to represent the size of the sprite. Also added to the effect file is a geometry shader. The geometry shader is set up to accept an input of a single point, and it outputs six vertices because it creates two triangles (three points each) that form the square shape. The body of the geometry shader will use the input point's position as the center position and will use the global size to create four points around the center point. It outputs each triangle using the stream object's `Append()` function, and once a triangle has been fully outputted, the `RestartStrip()` function is called to mark the start of the next triangle. Keep in mind that each triangle is separated by a `RestartStrip()` call in a geometry shader.

To ensure that the triangles are always facing the camera, we use the billboard technique. This is done by taking the right and up vectors (i.e., directions) of the view matrix. These vectors act as directions, where -right is left, right is right, up is up, and -up is down. Therefore, to generate the point that goes in the upper left, we add the center position to the -right + up vectors (e.g., pos + [-right + up]) to move the center position to the upper left. We can then pass that position to the triangle stream output using `Append()`. We do this for the upper-left, upper-right, lower-left, and lower-right positions of the square that will act as the sprite.

Vectors and matrices will be discussed in more detail in the next chapter. The view matrix represents the camera. By taking the first column (right vector) and the first row (up vector), we can use those directions in 3D space to move the center point around in a way that gives the impression that the resulting square is facing the camera. Mathematically, what is happening is that we determine which directions are considered right of and up from the camera, and we generate a square in those directions so that no matter where the camera is facing, we can always generate geometry that looks directly at the camera. You can see this by holding up a pencil and moving it to the left and up (upper left) of where you are facing. Then move it right and up from its original position and so forth. The view matrix uses the same idea. As long as you move the object with respect to what the view (your eyes) considers right and up or down and left, the resulting geometry is always facing you.

The HLSL shader from the Billboard Sprite demo is shown in Listing 7.3. Note that the geometry shader also generates texture coordinates so that the square can be textured properly. Also, since the size is set every frame, it is added to the constant buffer that is marked for change on a frame-by-frame basis. Since the geometry coming into the geometry shader is the transformed data from the vertex shader, we must use the original transformed W of each point for the pixel shader to get the correct data. Once the data has been transformed by the vertex shader, we don't have to alter it in a way that changes its meaning and therefore alters our output in ways we don't expect.

LISTING 7.3. THE BILLBOARD SPRITE'S HLSL EFFECT SHADER

```
/*
  Billboard Sprite Demo's Shader
  Ultimate Game Programming with DirectX 2nd Edition
  Created by Allen Sherrod
*/

uniform Texture2D decal;

SamplerState DecalSampler
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};


BlendState AlphaBlending
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = TRUE;
    SrcBlend = SRC_ALPHA;
    DestBlend = INV_SRC_ALPHA;
    BlendOp = ADD;
    SrcBlendAlpha = ZERO;
    DestBlendAlpha = ZERO;
    BlendOpAlpha = ADD;
    RenderTargetWriteMask[0] = 0×0F;
};


cbuffer cbChangesEveryFrame
```

```
{
   matrix World;
   matrix View;
   float size;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};

struct VS_INPUT
{
   float4 Pos : POSITION;
};

struct GS_INPUT
{
   float4 Pos : SV_POSITION;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
   float2 Tex : TEXCOORD0;
};

GS_INPUT VS(VS_INPUT input)
{
   GS_INPUT output = (GS_INPUT)0;

   output.Pos = mul(input.Pos, World);
   output.Pos = mul(output.Pos, View);
   output.Pos = mul(output.Pos, Projection);

   return output;
}


[maxvertexcount(6)]
void GS(point GS_INPUT input[1],
        inout TriangleStream<PS_INPUT> triStream)
{
   PS_INPUT output = (PS_INPUT)0;

   matrix modelView = World * View;

   // Used for generating billboard geometry aligned to the
camera.

   float3 right = float3(modelView._m00, modelView._m10,
                         modelView._m20);
```

```
   float3 up = float3(modelView._m01, modelView._m11,
                      modelView._m21);

   float3 pos = input[0].Pos.xyz;


   // Must use transformed vertex W. Start first triangle.

   output.Pos = float4(pos + (-right + up) * size,
input[0].Pos.w);
   output.Tex = float2(0.0, 1.0);
   triStream.Append(output);

   output.Pos = float4(pos + (right + up) * size,
input[0].Pos.w);
   output.Tex = float2(0.0, 0.0);
   triStream.Append(output);

   output.Pos = float4(pos + (right - up) * size,
input[0].Pos.w);
   output.Tex = float2(1.0, 0.0);
   triStream.Append(output);


   // Start next triangle.

   triStream.RestartStrip();
   output.Pos = float4(pos + (right - up) * size,
input[0].Pos.w);
   output.Tex = float2(1.0, 0.0);
   triStream.Append(output);

   output.Pos = float4(pos + (-right - up) * size,
input[0].Pos.w);
   output.Tex = float2(1.0, 1.0);
   triStream.Append(output);

   output.Pos = float4(pos + (-right + up) * size,
input[0].Pos.w);
   output.Tex = float2(0.0, 1.0);
   triStream.Append(output);

   triStream.RestartStrip();
}

float4 PS(PS_INPUT input) : SV_Target
{
   return decal.Sample(DecalSampler, input.Tex);
}

technique10 Billboard
```

```
{
   pass P0
   {
      SetBlendState(AlphaBlending, float4(0.0f, 0.0f, 0.0f,
0.0f),
                    0xFFFFFFFF);

      SetVertexShader(CompileShader(vs_4_0, VS()));
      SetGeometryShader(CompileShader(gs_4_0, GS()));
      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

On the application side, the Billboard Sprite demo sends four points to Direct3D 10. Each of these points acts as the center of a sprite, and, when rendered, the geometry shader generates squares around each of these points. Since the geometry shader also generates the texture coordinates, the only thing we need to render is a list of positions. The global section of the demo's main.cpp source file is shown in Listing 7.4, where an effect variable for the sprite's size has been added to the mix. Also in Listing 7.4, you can see the modified `InitializeDemo()` function and `RenderScene()` function. Notice that in the `RenderScene()` function the flag `D3D10_PRIMITIVE_TOPOLOGY_POINTLIST` is used to specify that we are sending points, not triangles, to the Direct3D 10 API. Even though the geometry shader generates triangles, technically, we are rendering points with this function call, and therefore Direct3D 10 should expect points as the input, not triangle primitives. Listing 7.4 shows the code that was modified to create the Billboard Sprite demo and was built from the Alpha Mapping demo. To avoid listing too much redundant code, we have listed the `InitializeDemo()` function up until the point at which the vertex buffer is created.

LISTING 7.4. THE MODIFIED CODE OF THE BILLBOARD SPRITE DEMO'S MAIN SOURCE FILE

```
/*
    Billboard Sprites
    Ultimate Game Programming with DirectX 2nd Edition
    Created by Allen Sherrod
*/

#include<d3d10.h>
#include<d3dx10.h>

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Billboard Sprites"
#define WINDOW_CLASS    "UPGCLASS"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600
// Global window handles.
HINSTANCE g_hInst = NULL;
HWND g_hwnd = NULL;

// Direct3D 10 objects.
```

```
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;

struct DX10Vertex
{
   D3DXVECTOR3 pos;
};

ID3D10InputLayout *g_layout = NULL;
ID3D10Buffer *g_pointsVB = NULL;
ID3D10ShaderResourceView *g_spriteDecal = NULL;

ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_billboardTech = NULL;
ID3D10EffectShaderResourceVariable *g_decalEffectVar = NULL;
ID3D10EffectScalarVariable *g_pointSpriteSize = NULL;
ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;

D3DXMATRIX g_worldMat, g_viewMat, g_projMat;


bool InitializeDemo()
{
   // Load the shader.
   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   HRESULT hr = D3DX10CreateEffectFromFile(
      "BillboardSpritesShader.fx",
      NULL, NULL, "fx_4_0", shaderFlags, 0, g_d3dDevice, NULL,
      NULL, &g_shader, NULL, NULL);

   if(FAILED(hr))
      return false;

   g_billboardTech = g_shader->GetTechniqueByName("Billboard");
   g_worldEffectVar = g_shader->GetVariableByName(
      "World")->AsMatrix();

   g_viewEffectVar = g_shader->GetVariableByName(
      "View")->AsMatrix();

   g_projEffectVar = g_shader->GetVariableByName(
      "Projection")->AsMatrix();

   g_decalEffectVar = g_shader->GetVariableByName(
```

```
      "decal")->AsShaderResource();

   g_pointSpriteSize = g_shader->GetVariableByName(
      "size")->AsScalar();


   // Load the texture.
   hr = D3DX10CreateShaderResourceViewFromFile(g_d3dDevice,
      "sprite.dds", NULL, NULL, &g_spriteDecal, NULL);
   if(FAILED(hr))
       return false;


   // Create the geometry.

   D3D10_INPUT_ELEMENT_DESC layout[] =
   {
      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
   };

   unsigned int numElements = sizeof(layout) /
sizeof(layout[0]);
   D3D10_PASS_DESC passDesc;

   g_billboardTech->GetPassByIndex(0)->GetDesc(&passDesc);

   hr = g_d3dDevice->CreateInputLayout(layout, numElements,
      passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
      &g_layout);

   if(FAILED(hr))
      return false;

   DX10Vertex vertices[] =
   {
      { D3DXVECTOR3( 0.5f, 0.5f, 3.0f) },
      { D3DXVECTOR3( 0.5f, -0.5f, 3.0f) },
      { D3DXVECTOR3(-0.5f, -0.5f, 3.0f) },
      { D3DXVECTOR3(-0.5f, -0.5f, 3.0f) },
      { D3DXVECTOR3(-0.5f, 0.5f, 3.0f) },
      { D3DXVECTOR3( 0.5f, 0.5f, 3.0f) }
   };
   …

   return true;
}

void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };
```

```
    g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);

    g_worldEffectVar->SetMatrix((float*)&g_worldMat);
    g_decalEffectVar->SetResource(g_spriteDecal);
    g_pointSpriteSize->SetFloat(0.3f);

    unsigned int stride = sizeof(DX10Vertex);
    unsigned int offset = 0;

    g_d3dDevice->IASetInputLayout(g_layout);

    g_d3dDevice->IASetVertexBuffers(0, 1, &g_pointsVB, &stride,
                                    &offset);

    g_d3dDevice->IASetPrimitiveTopology(
       D3D10_PRIMITIVE_TOPOLOGY_POINTLIST);

    D3D10_TECHNIQUE_DESC techDesc;
    g_billboardTech->GetDesc(&techDesc);

    for(unsigned int i = 0; i < techDesc.Passes; i++)
    {
       g_billboardTech->GetPassByIndex(i)->Apply(0);
       g_d3dDevice->Draw(6, 0);
    }

    g_swapChain->Present(0, 0);
}
```

Figure 7.4 shows a screenshot of the Billboard Sprite demo.

FIGURE 7.4. SCREENSHOT FROM THE BILLBOARD SPRITE DEMO.

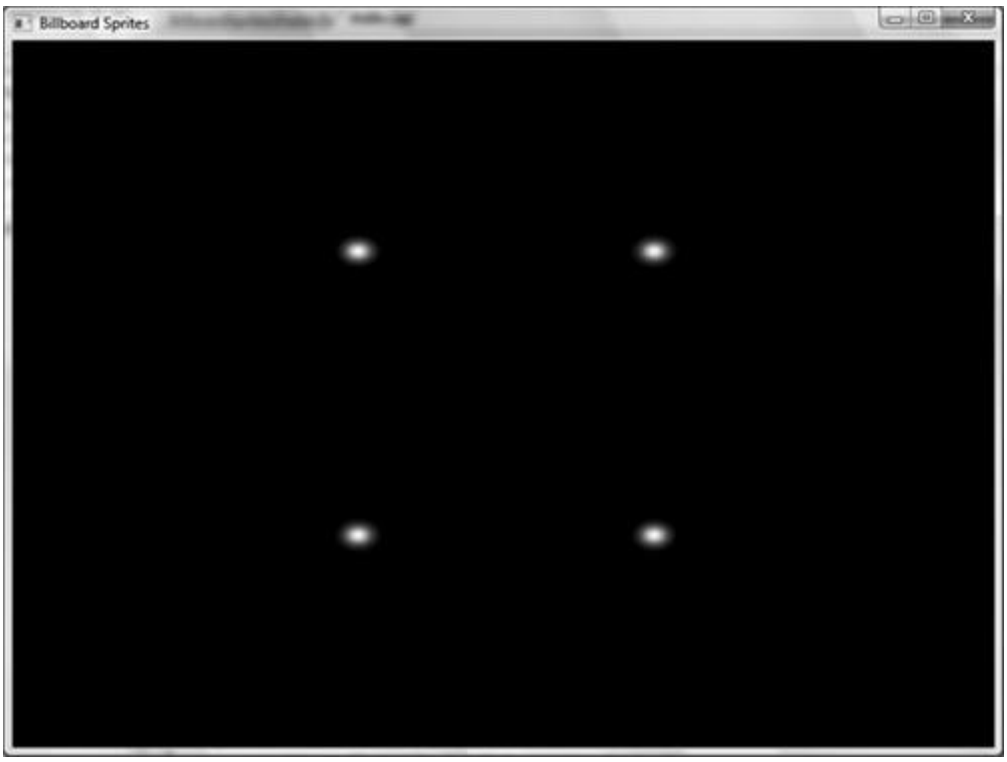


IMAGE FILTERS

The next topic we will discuss is image filters. An image filter is an algorithm that is executed to convert a picture from its original input to some type of output, which is dependent on the algorithm itself. Examples of some image filters are color inversion filters, black-and-white (luminance) filters, sepia filters, edge detection filters, blur filters, and so forth.

In this section we will look at a few simple image filters and apply them to textures being rendered on square surfaces to keep the code short and simple. Each of these image-filtering demos is a modified version of the Texture Mapping demo in Chapter 6, so the only new code is in the effect files.

COLOR INVERSION

Color inversion is a simple effect to create when it comes to image filters. To perform this effect we only have to use 1 minus the color in the pixel shader. This makes colors that are 1 equal to 0 and colors that are 0 equal to 1. In other words, it flips the colors around, so dark becomes light and light becomes dark.

On the CD-ROM, in the Chapter 7 folder, is a demo called Color Inversion. The effect shader file from the demo is shown in Listing 7.5. Figure 7.5 shows a screenshot of the effect.

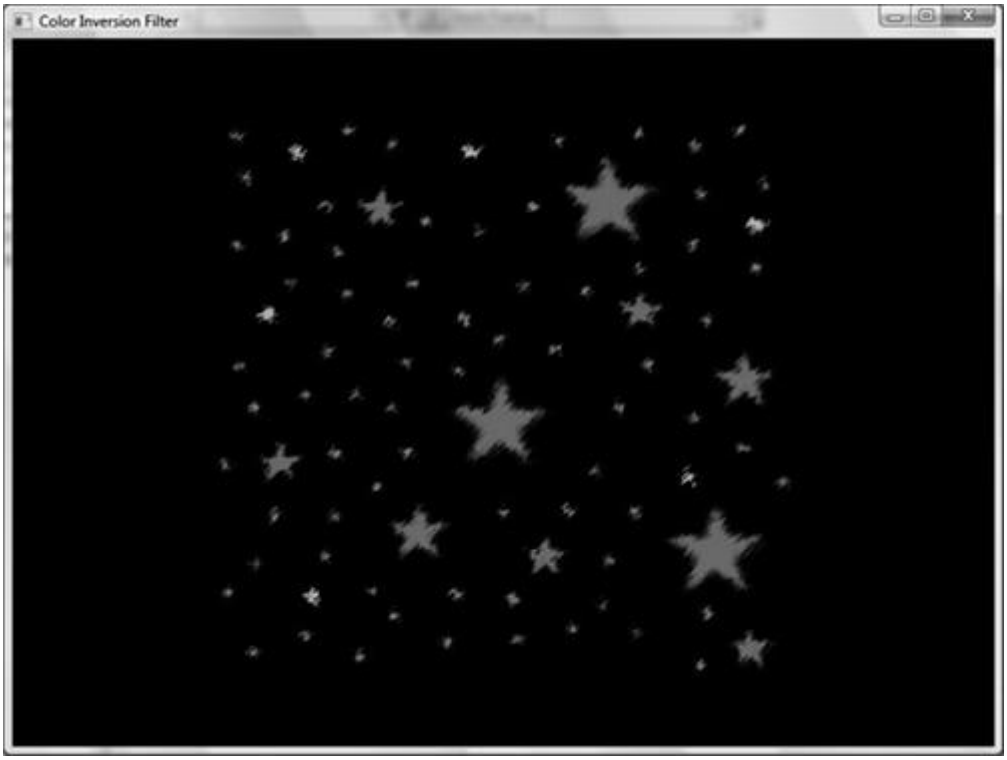
LISTING 7.5. COLOR INVERSION DEMO'S EFFECT SHADER FILE

```
/*
     Color Inversion Filter Demo's HLSL Shader
```

```
     Ultimate Game Programming with DirectX 2nd Edition
     Created by Allen Sherrod
*/

Texture2D decal;

SamplerState DecalSampler
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};

cbuffer cbChangesEveryFrame
{
    matrix World;
    matrix View;
};

cbuffer cbChangeOnResize
{
    matrix Projection;
};

struct VS_INPUT
{
   float4 Pos : POSITION;
   float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
   float2 Tex : TEXCOORD0;
};

PS_INPUT VS(VS_INPUT input)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = mul(input.Pos, World);
   output.Pos = mul(output.Pos, View);
   output.Pos = mul(output.Pos, Projection);

   output.Tex = input.Tex;

   return output;
}

float4 PS(PS_INPUT input) : SV_Target
{
   return 1 - decal.Sample(DecalSampler, input.Tex);
```

```
}
technique10 ColorInversion
{
   pass P0
   {
      SetVertexShader(CompileShader(vs_4_0, VS()));
      SetGeometryShader(NULL);
      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

FIGURE 7.5. SCREENSHOT FROM THE COLOR INVERSION DEMO.



LUMINANCE FILTER

The next filter is known as a luminance filter. This effect is used to convert a color picture to black and white. You can find the Luminance demo on the CD-ROM in the Chapter 7 folder.

The algorithm for the luminance filter works by taking each color in the pixel shader and multiplying each component by the luminance constant. You then add the result of each multiplied component together to get the value that will act as the grayscale pixel output. The luminance constant is 0.30 for the red, 0.59 for the green, and 0.11 for the blue components. Multiplying the color's red, green, and blue components by this luminance constant and then adding the results of all components will create a result that appears black and white for all pixels. The operations are shown as follows, where color is the original pixel color and lum is the luminance constant (0.30, 0.59, 0.11).

> Luminance = color.red * lum.red + color.green * lum.green + color.blue * lum.blue

This operation is essentially the dot product of two vectors, or in this case two colors. You can use the HLSL `dot()` function to perform this operation, which will be discussed in more detail in the next chapter for vectors. For now, know that the `dot()` function takes each component from the two vectors (color and the luminance constant in this example) and multiplies them together and adds up the results. The resulting value is used for the output color. The Luminance demo's effect shader file is shown in Listing 7.6. Figure 7.6 shows a screenshot of the demo in action.

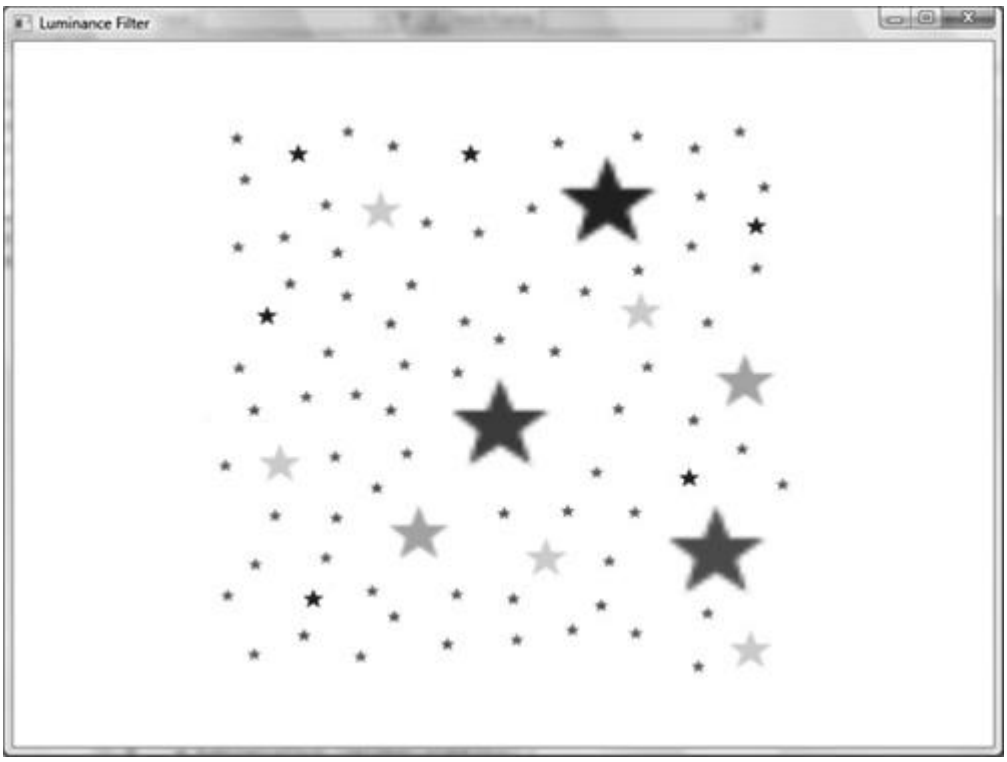
LISTING 7.6. THE LUMINANCE DEMO'S EFFECT SHADER FILE

```
/*
    Luminance Filter
    Ultimate Game Programming with DirectX 2nd Edition
    Created by Allen Sherrod
*/

Texture2D decal;

SamplerState DecalSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};

cbuffer cbChangesEveryFrame
{
    matrix World;
    matrix View;
};

cbuffer cbChangeOnResize
{
    matrix Projection;
};

struct VS_INPUT
{
   float4 Pos : POSITION;
   float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float2 Tex : TEXCOORD0;
};

PS_INPUT VS(VS_INPUT input)
{
   PS_INPUT output = (PS_INPUT)0;
   output.Pos = mul(input.Pos, World);
   output.Pos = mul(output.Pos, View);
```

```
    output.Pos = mul(output.Pos, Projection);

    output.Tex = input.Tex;

    return output;
}

float4 PS(PS_INPUT input) : SV_Target
{
     float3 lumConst = float3(0.30, 0.59, 0.11);

     float3 color = decal.Sample(DecalSampler, input.Tex);
     float dp = dot(color, lumConst);

     return float4(dp, dp, dp, 1);
}


technique10 LuminanceFilter
{
    pass P0
    {
        SetVertexShader(CompileShader(vs_4_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, PS()));
    }
}
```

The luminance filter is also known as the black-and-white transfer function.

FIGURE 7.6. A SCREENSHOT FROM THE LUMINANCE DEMO.



SEPIA TONE FILTER

The sepia tone effect is used to color a black-and-white image with a brownish tone to give it the appearance of and old photograph. It does this by taking the original color and a brown tone, and using the two to color a black-and-white version of the image to create the effect. As the description implies, the sepia tone builds off of the luminance filter. The Sepia demo can be found on the accompanying CD-ROM in the Chapter 7 folder.

The sepia tone filter works in the following steps.

1. Convert the image to black and white.

2. Use the luminance (black-and-white) value as a percent between a light and dark set of colors to create the sepia constant, where 0% (0.0) means we use the dark color, 100% (1.0) means we use the light color, and any percentage between 0% and 100% will be a color value between the light and dark colors.

3. Take the original color and the luminance color and find a color that is halfway between the two. Let's call this the half color.

4. Use the half color and the sepia constant and find a color that is halfway between those two. This will be the output color.

In HLSL you can find a color that is between two colors by using the `lerp()` function. The `lerp()` function takes as parameters the first vector (or color in this demo, which is represented by a vector; i.e., they're the same thing from a structure point of view), the second vector, and a percentage to interpolate between the two. Finding a color between

two colors is another way of mixing colors because the result will be a blend of the two colors. When you find a color that is halfway between two other colors, it is like you are equally mixing the colors together.

The term *lerp* stands for linear interpolation. For a simple example, let's say we are interpolating between the values 0 and 100. The percentage is used to find a value between these two, where a percentage of 0.0 will return 0, and a percent of 1.0 will return 100. So if the percentage is 0.65, then 65 will be returned since 65% into the range of 0 and 100 is 65. The `lerp()` function does this operation, but on two vectors to obtain a vector that is some percentage between the two parameters. Therefore, to find a color between two colors using `lerp()`, you just specify the two colors and a percentage between the two for which you want to look. Since we are using colors, the result will be a blend between the two.

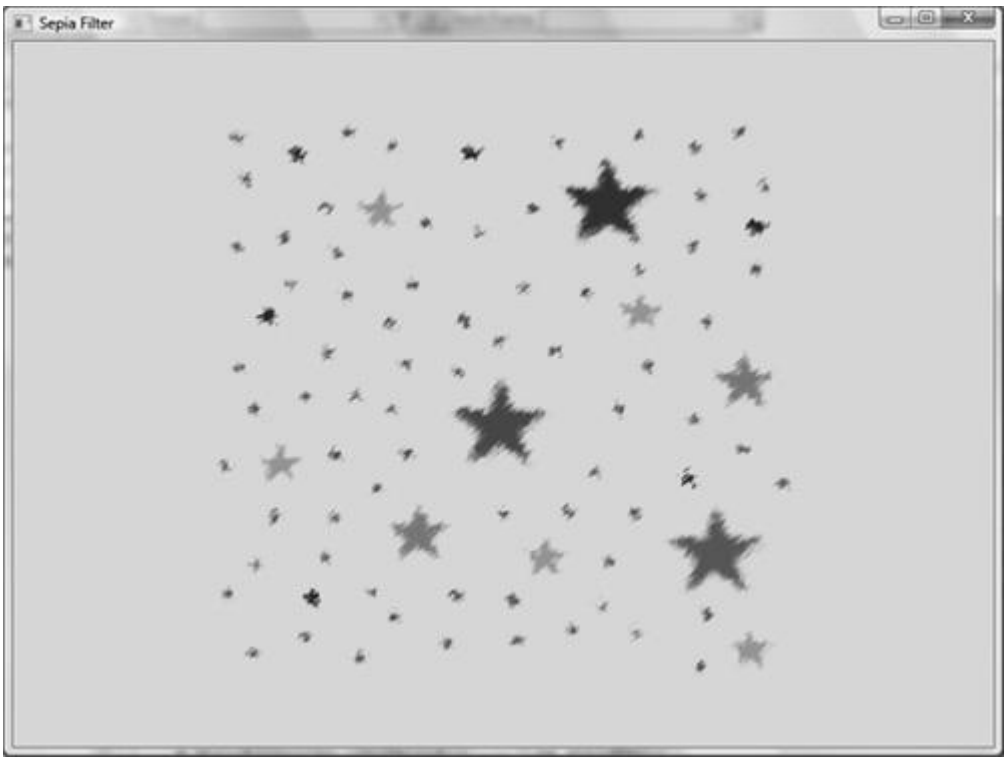
The sepia tone effect is shown in Listing 7.7 in the demo's shader file. This demo builds off of the Luminance demo and adds a few extra lines of code to the pixel shader. Figure 7.7 shows a screenshot of the demo. We recommend that you run the demo applications to see the results in color to be able to fully appreciate the differences between each image filter.

LISTING 7.7. THE SEPIA DEMO'S HLSL EFFECT SHADER FILE

```
/*
   Sepia Filter Demo's HLSL Shader
   Ultimate Game Programming with DirectX 2nd Edition
   Created by Allen Sherrod
*/

Texture2D decal;

SamplerState DecalSampler
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};

cbuffer cbChangesEveryFrame
{
    matrix World;
    matrix View;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};

struct VS_INPUT
{
   float4 Pos : POSITION;
   float2 Tex : TEXCOORD;
};

struct PS_INPUT
```

```
{
   float4 Pos : SV_POSITION;
   float2 Tex : TEXCOORD0;
};

PS_INPUT VS(VS_INPUT input)
{
   PS_INPUT output = (PS_INPUT)0;

   output.Pos = mul(input.Pos, World);
   output.Pos = mul(output.Pos, View);
   output.Pos = mul(output.Pos, Projection);

   output.Tex = input.Tex;

   return output;
}

float4 PS(PS_INPUT input) : SV_Target
{
   float3 lumConst = float3(0.30, 0.59, 0.11);
   float3 light = float3(1, 0.9, 0.5);
   float3 dark = float3(0.2, 0.05, 0);

   float3 color = decal.Sample(DecalSampler, input.Tex);

   float luminance = dot(color, lumConst);
   float3 sepia = lerp(dark, light, luminance);
   float3 halfColor = lerp(color, luminance, 0.5);
   float3 final = lerp(halfColor, sepia, 0.5);

   return float4(final, 1);
}

technique10 SepiaFilter
{
   pass P0
   {
      SetVertexShader(CompileShader(vs_4_0, VS()));
      SetGeometryShader(NULL);
      SetPixelShader(CompileShader(ps_4_0, PS()));
   }
}
```

FIGURE 7.7. A SCREENSHOT FROM THE SEPIA DEMO.



SUMMARY

In this chapter we looked at several fairly straightforward ways to use texture mapping in game graphics. The possibilities for texture mapping seem endless. Throughout the remainder of this book we will look at a few additional uses for textures when discussing topics such as lighting (specifically, bump and normal mapping) and shadows (specifically, shadow mapping).

The following elements were discussed in this chapter.

- Alpha mapping
- Sprites
- Billboard sprites
- Point sprites
- Color inversion filters
- Black-and-white (luminance) filters
- Sepia tone filters

In the next chapter we will take a deeper look into the common mathematics used in video games. We've already looked briefly at the commonly used math objects, but in the next chapter you will gain a much deeper understanding of what they are, why they exist, and how to use them.

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. List at least five ways textures can be used in game graphics outside of directly coloring a surface.

2. What is alpha mapping?

3. What are the two ways alpha blending can be enabled in Direct3D 10?

4. What two ways to store alpha map values were discussed in this chapter?

5. Why would you use 1 bit for alpha values rather than using 1 byte?

6. What is alpha to coverage, and what part does it play in game graphics?

7. List and describe five of the blend options that can be used for `SrcBlend` in the blend descriptor.

8. What is a sprite?

9. What is a point sprite?

10. How does Direct3D 10 point sprite support differ from Direct3D 9?

11. What is a billboard sprite? How do you calculate a sprite that always faces the camera?

12. How did we create point sprites in Direct3D 10 in this chapter?

13. Describe the luminance filter algorithm.

14. Describe the color inversion filter algorithm.

15. Describe the sepia tone filter algorithm.

CHAPTER EXERCISES

Exercise 1: Create a demo that allows the user to choose the inversion amount. Allow the user to specify a value between 0.0 and 1.0 instead of always using 1.0 as the inversion amount (which was seen in the Color Inversion demo as 1 minus color).

Exercise 2: Build off of Exercise 1 and allow the user to choose which color component to invert. Allow the user to use the red channel, green channel, blue channel, or all channels. Do this by creating different techniques that use slightly different pixel shaders. That way the user can choose which technique he wants to use on the application side.

Exercise 3: Create a new demo that combines your color inversion code from Exercise 2 and the sepia tone effect.

8. GAME MATH

In This Chapter

- Vectors
- Planes
- Matrices
- Bounding Geometry
- Additional Mathematics

Mathematics is the foundation of everything in video game development. Mathematics is used for everything from computer graphics to artificial intelligence, physics, game logic, and so forth. To be successful in video game programming, especially graphics programming, it is essential that you have a strong mathematics background. If you do not have a strong mathematics background, it is important to learn as much as you can because that will only make creating the types of environments that you dream of easier in the long run.

In this chapter we will briefly review the most common types of mathematic structures and topics that you will encounter as a game and graphics programmer. The topic of game mathematics could fill its own book, so if you do not have a strong math background, we recommend that you obtain a good book on the subject.

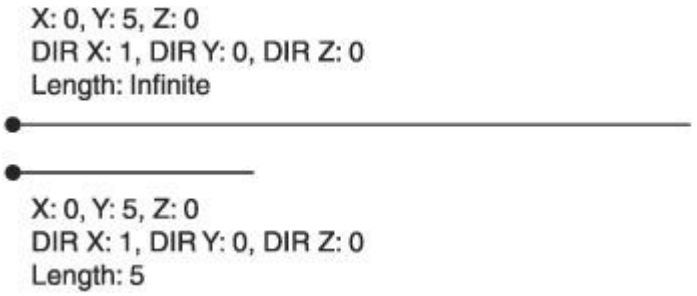
You do not need to be familiar with the mathematics topics discussed in this chapter to use the various math structures and functions. It would help to have a better understanding of "why" if you pick up a good book on the topic. It would also help to be able to code your own math implementations instead of using DirectX's code. Since DirectX provides the technical implementation for these common math objects, the details are hidden from you unless you decide to write your own custom math code.

VECTORS

A vector is a mathematical structure that is used to represent a direction. There are different types of vectors, and the most common kind you will see in computer graphics are 2D, 3D, and 4D vectors. The type of vector determines the number of axes or dimensions it represents. Therefore, a 3D vector is a vector that exists in 3D space and has X, Y, and Z axes.

At its heart a vector is a direction, and in computer graphics the structures used to define vectors are often used to represent positions as well. Taking a 3D vector as an example, the X, Y, and Z axes can be used to mark a position just as they can mark a direction. The difference lies in what the programmer intends to use the data for. In math and physics a vector is an object with a direction and a length. A simple visual of this is shown in Figure 8.1, where the vector moves from the origin (the origin in 3D space has an X axis of 0, Y of 0, and Z of 0) along the positive Y axis 10 units, making its final position X:0, Y:10, Z:0. The direction of the vector is

FIGURE 8.1. A SIMPLE LOOK AT A VECTOR AS A DIRECTION AND A LENGTH.



X:0, Y:1, and Z:0. Since the vector only moves long the Y axis (Y:1), only the Y has a value.

In this chapter we'll look at 3D vectors, but keep in mind that everything discussed here also applies to 2D and 4D vectors. A 2D vector is made up of X and Y axes, and a 4D vector is made up of X, Y, Z, and W axes. The Direct3D 3D vector is called D3DXVECTOR3, and it has the following structure according to the DirectX 10 documentation.

```
typedef struct D3DXVECTOR3 {
    FLOAT x;
    FLOAT y;
    FLOAT z;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

This structure can be used to represent both positions and directions. In the case of a vertex, this structure is often used to represent the position attribute of each vertex of a primitive. In the upcoming subsections of this discussion on vectors, we will briefly discuss a few of the most common mathematical operations performed on vector objects. To view a complete list of the mathematical functions offered by the DirectX SDK vector objects, refer to the DirectX documentation.

Vectors are used for all types of mathematical objects, especially in game physics. This includes directions, positions, tensors, pseudovectors, and other vector-like objects. In other words, if it has an X, Y, and Z property (using 3D vectors as an example), then most likely programmers will use their vector code to represent it rather than creating another structure with the same properties (member variables). This makes it easier to get started if you do not have strong math skills.

VECTOR ADDITION AND SUBTRACTION

The first operations we'll look at in vector mathematics are adding and subtracting. Mathematically, adding and subtracting are very elementary. To add two vectors together, again using 3D vectors as an example, you add each of the matching axes together, and the result is stored in a new vector holding the solution. In other words, you take the X axis from vectors A and B and add them together and store answer in a result vector's X axis. You do the same with the Y and Z axes. This is shown in the following example.

```
Vector3D A = (10, 5, 8)
Vector3D B = (3, 1, 11)

Vector3D Result = A + B
```

or

```
Result = (13, 6, 19)
```

or

```
Result.x = A.x + B.x
Result.y = A.y + B.y
Result.z = A.z + B.z
```

Using the vectors from the addition example, subtraction is the same, but instead of adding you are literally subtracting each axis from its matching counterpart in the other vector. If vector A is (10, 5, 8) and vector B is (3, 1, 11), the resulting vector when subtracting is (7, 4, –3).

Vectors in DirectX are added using the function `D3DXVec3Add()` and subtracted using the `D3DXVec3Subtract` function. These functions have the following function prototypes where the function returns the result as a vector. The first parameter is the address for the vector that will store the result of the operation, the second parameter is the first vector in the operation (vector A), and the last parameter is the second vector in the operation (vector B).

```
D3DXVECTOR3 * D3DXVec3Add(
   D3DXVECTOR3 * pOut,
   CONST D3DXVECTOR3 * pV1,
```

```
    CONST D3DXVECTOR3 * pV2
);

D3DXVECTOR3 * D3DXVec3Subtract(
    D3DXVECTOR3 * pOut,
    CONST D3DXVECTOR3 * pV1,
    CONST D3DXVECTOR3 * pV2
);
```

The result from these functions can be obtained by their return value or by passing the address of the object to hold the result in the first parameter. Alternatively, you can use the structure's overloaded operators to perform addition and subtraction instead of calling these functions. This would result in the following in code.

```
D3DXVECTOR3 vectorA, vectorB, result;
result = vectorA + vectorB;
result = vectorA - vectorB;
```

VECTOR NORMALIZATION

The length of a vector is called its magnitude. To find the length of a vector you multiply each component of a vector with itself and add all of the axes. The square root of this result is the magnitude of the vector. This is shown in the following pseudo-code example.

```
length = square_root(vector.x * vector.x + vector.y * vector.y +
vector.z * vector.z)
```

This equation gives you the inner product. The square root of this is the length of a vector. When a vector has a length that equals 1, it is said that the vector is unit-length. Another term for this is *normalized vector* (*normal* for short). The length itself is a floating-point value.

A normal has many uses in video game development. Later in this book you'll see how normal vectors contribute to the lighting equation. In this chapter we will briefly discuss how to convert a vector to a normal.

To convert a vector to a normal, the first step is to find the vector's length. If the length equals 1, the vector is already unit-length, and nothing else needs to be calculated. If the length is not 1, you can divide each axis of the vector by the length, which will result in scaling the vector to unit-length. This is done as follows.

```
D3DXVECTOR3 vectorA, normal;
float length = D3DXVec3Length(&vectorA);
normal = vectorA / length;
```

The `D3DXVec3Length()` function can be used to find the length of a `D3DXVECTOR3` object. Alternatively, you can normalize a vector by calling the DirectX function `D3DXVec3Normalize()`, which takes as parameters the address of the vector that will store the result of the operation and the vector to normalize. Normalized vectors are used for many mathematical equations, such as lighting for example. The function prototype for the `D3DXVec3Normalize()` function is shown as follows.

```
D3DXVECTOR3 * D3DXVec3Normalize(
   D3DXVECTOR3 *pOut,
   CONST D3DXVECTOR3 *pV
);
```

COMMON ADDITIONAL VECTOR OPERATIONS

We'll look at a few other vector operations in this chapter that will come up later in this book in discussions of various topics. These operations include the following.

- Dot product
- Cross product
- Lerp

The dot product is result of multiplying two vectors and adding the resulting axes. The dot product of two vectors can be found as follows using 3D vectors as an example.

```
float dot = vectorA.x * vectorB.x + vectorA.y * vectorB.y +
vectorA.z *
vectorB.z;
```

In Chapter 13, "Lighting," you will see how we use the dot product to get the angle between two vectors and use that information for determining how a surface should be lit.

The cross product of two vectors, put simply, is obtained by cross multiplying the axes of one vector with the axes of another. The cross product is used to find a vector that is perpendicular to two source vectors, which can be useful when you need such a vector in relation to two other vectors. The cross product, also known as the vector product, is shown below, where the axes that are multiplied are cross multiplied with one vector to another.

```
cross.x = vectorA.y * vectorB.z - vectorA.z * vectorB.y
cross.y = vectorA.z * vectorB.x - vectorA.x * vectorB.z
cross.z = vectorA.x * vectorB.y - vectorA.y * vectorB.x
```

Lerp is short for linear interpolation. It is an operation that is used to find a value that lies somewhere between two source values. The idea is to take a start value, an end value, and a percentage from 0.0 to 1.0 (i.e., 0 to 100%). If the percentage supplied is 0.0, the start vector is returned. If the percentage is 1.0, the ending value is returned, but if the percentage is a value between the two, a vector that lies in the percentage between the two will be returned.

For example, using single values, let's say we have a start value of 6 and an end value of 18. If we supply a percentage of 50%, that is like saying what is 50% into the range of 6 and 18. The answer is 12, since 12 lies halfway between 6 and 18. The same concept is used for linear interpolation with vectors, but this concept is applied to each axis of the vector. When using linear interpolation, you are linearly finding a value between two vectors based on the percentage, which often has the notation *t* in game development books. The equation for finding the linear interpolated vector between vector A and B is shown in the following example.

```
result = (vectorB - vectorA) * percentage + vectorA
```

The equation for the linear interpolation is quite simple. It works by finding the range total between values (vectors) A and B, multiplying that by the percentage (t), and adding that to the starting vector. So if we wanted to lerp between the values 13 and 57 by 0.4 (40%), we would first find the range (57 – 13, which equals 44) and then multiply 44 by 0.4, which is 17.6. We would then add 17.6 to the starting value to get the value that lies 40% into the range, which would result in 30.6.

Linear interpolation is sometimes used for animations, where time is used as *t*. So if an animation had to occur within a certain time frame, for example, you could interpolate between two vertex positions to find where the vertex would be at a specific time. If you do this for all vertices in a model, you get the type of animation used in many early 3D video games before bone animation (discussed later in the book) became the standard.

DIRECTX 3D VECTOR FUNCTIONS

The DirectX SDK offers a number of functions for vector objects. In this section we will look briefly at the 3D vector functions. The 2D and 4D vectors have equivalent functions, although a few 3D functions do not have a 2D or 4D counterpart. For example, there is no `D3DXVec2Cross()`. Table 8.1 lists the 3D vector functions in the DirectX SDK.

TABLE 8.1. THE 3D VECTOR FUNCTIONS FROM THE DIRECTX SDK

Function	Definition
`D3DXVec3Add()`	Vector addition
`D3DXVec3BaryCentric()`	Returns a point in Barycentric coordinates
`D3DXVec3CatmullRom()`	Performs CatmullRom interpolation
`D3DXVec3Cross()`	Performs the cross product of two vectors
`D3DXVec3Dot()`	Performs the dot product between two vectors
`D3DXVec3Hermite()`	Performs Hermite spline interpolation
`D3DXVec3Length()`	Calculates the length (magnitude) of a vector
`D3DXVec3LengthSq()`	Calculates the square of the vector's length
`D3DXVec3Lerp()`	Performs linear interpolation
`D3DXVec3Maximize()`	Finds the maximum vector of two source vectors

TABLE 8.1. THE 3D VECTOR FUNCTIONS FROM THE DIRECTX SDK

Function	Definition
D3DXVec3Minimize()	Finds the minimum vector of two source vectors
D3DXVec3Normalize()	Normalizes a vector to unit-length
D3DXVec3Project()	Projects a vector from object space to screen space
D3DXVec3ProjectArray()	Projects a float array from object space to screen space
D3DXVec3Scale()	Scales a vector
D3DXVec3Subract()	Vector subtraction
D3DXVec3TransformArray()	Transforms a float array by a matrix
D3DXVec3TransformCoord()	Transforms a vector by a matrix and projects back into the $w = 1$
D3DXVec3TransformCoordArray()	Transforms a float array by a matrix and projects back into the $w = 1$
D3DXVec3TransformNormal()	Performs a 3 × 3 vector/matrix transformation to transform a normal vector by a matrix
D3DXVec3TransformNormalArray()	Performs a 3 × 3 vector/matrix transformation to transform a normal vector represented as a float array by a matrix
D3DXVec3UnProject()	Projects a vector from screen space back to object space
D3DXVec3UnProjectArray()	Projects a vector represented as a float array from screen space back to object space
D3DXVec3Transform()	Transforms a vector by a matrix

PLANES

A plane can be thought of as an infinitely thin surface that extends forever alone two axes. Planes have many uses in video games, many of which fall under the subject of collision detection, where a plane can be used to test if an object of some type travels from one side of the plane to the other.

Planes are not rendered; they are used mathematically for tests. These tests are essentially set up to test which side of the plane an object is on or if the object penetrates the plane. Take, for example, a game in which a cut scene is triggered when a player walks into a room. This can be done as simply as defining a plane and testing every frame to see if the player is on a different side of the plane than before. If so, the cut scene is triggered.

The plane equation is defined as ax + by + cz + dw = 0. In code a plane can be defined as a structure with coefficients a, b, c, and d. These coefficients are usually floating-point values. In DirectX the `D3DXPLANE` structure is defined as follows:

```
typedef struct D3DXPLANE {
    FLOAT a;
    FLOAT b;
    FLOAT c;
    FLOAT d;
} D3DXPLANE, *LPD3DXPLANE;
```

You can think of a plane as a normal that is defined by the first three coefficients, a, b, and c, and a distance defined by the last coefficient d. You can manually specify this information or you can create a plane from a primitive or surface. It is very common to create a plane out of a triangle and then use that plane for some purpose such as collision detection.

PLANE OPERATIONS

The DirectX SDK has several functions that can be used with plane objects. The definition of these functions (see Table 8.2) can give you an idea of what you can do with planes.

TABLE 8.2. DIRECTX SDK PLANE OBJECT

Function	Definition
`D3DXPlaneDot(const D3DXPLANE *pP, const D3DXVECTOR4 *pV)`	Computes the dot product of a plane and a 4D vector.
`D3DXPlaneDotCoord(const D3DXPLANE *pP, const D3DXVECTOR3 *pV)`	Computes the dot product of a plane and a 3D vector. This is the same as the `D3DXPlaneDot()` function but assumes a w of 1.
`D3DXPlaneDotNormal(const D3DXPLANE *pP, const D3DXVECTOR3 *pV)`	The same as `D3DXPlaneDotCoord()` but assumes a w of 0.
`D3DXPlaneFromPointNormal(D3DXPLANE`	Computes a plane from a point and

TABLE 8.2. DIRECTX SDK PLANE OBJECT

Function	Definition
`*pP, const D3DXVECTOR3 *pPoint, const D3DXVECTOR3 *pNormal)`	a normal.
`D3DXPlaneFromPoints(D3DXPLANE *pP, const D3DXVECTOR3 *v1, const D3DXVECTOR3 *v2, const D3DXVECTOR3 *v3)`	Creates a plane from three points. This can be used to define a plane from a triangle.
`D3DXPlaneIntersectLine (const D3DXVECTOR3 *pOut, const D3DXPLANE *pP, const D3DXVECTOR3 *v1, const D3DXVECTOR3 *v2)`	Tests to see if a line intersects with the plane. If it does, the point of intersection in 3D space is returned. The point v1 is the start of the line, and v2 is the end of the line. The pOut parameter stores the position of the intersection.
`D3DXPlaneNormalize(D3DXPLANE *pOut, const D3DXPLANE *pP)`	Normalizes a plane so that its coefficients are unit-length. If the a, b, and c of a plane are its normal, this function essentially normalizes it.
`D3DXPlaneScale(D3DXPLANE *pOut, const D3DXPLANE *pP, FLOAT s)`	Scales a plane by a specified amount.
`D3DXPlaneTransform(D3DXPLANE *pOut, const D3DXPLANE *pP, const D3DXMATRIX *pM)`	Transforms a plane by a matrix (see the upcoming “Matrices” section).
`D3DXPlaneTransformArray (D3DXPLANE *pOut, UINT OutStride, const D3DXPLANE *pP, UINT PStride, const D3DXMATRIX * pM, UINT n)`	Transforms an array of planes by a matrix.

MATRICES

A matrix (plural matrices) is a rectangular table of elements that is used for mathematical purposes. Put another way, a matrix in computer graphics is a 2D array of values that are used primarily to perform various operations on vectors. Direct3D and OpenGL traditionally

expect a 2D array of floating-point values for arrays, but HLSL and GLSL support matrices of other data types such as integers and Booleans. Matrices can be 4 × 4 (four columns and four rows), 3 × 4 (three columns and four rows), 3 × 3 (three columns and three rows), and so forth. Matrices were first discussed in Chapter 4, "Shader Model 4," in the discussion of HLSL. Refer to that chapter to see how matrices are defined in HLSL if you do not remember.

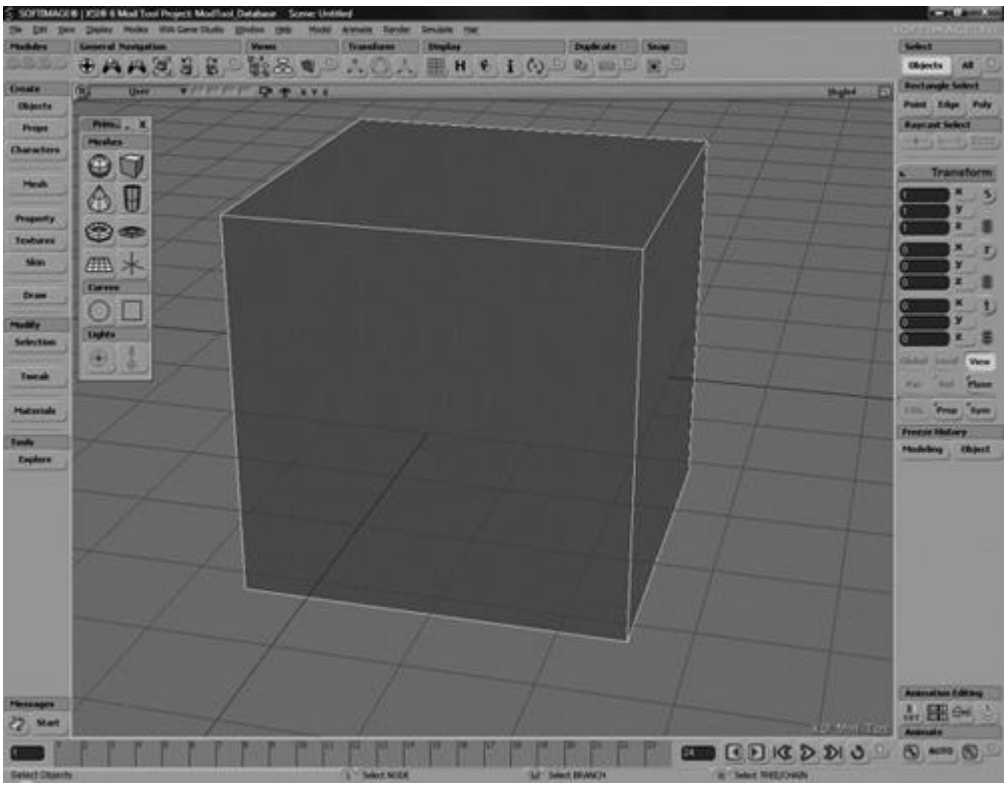
One of the primary uses for a matrix is to perform a math operation on a vector to change that vector in some meaningful way. Matrices can store rotation, scaling, and translational (positional) information. A 3 × 3 matrix, that is, a matrix (2D array) with three rows and three columns, is used to store rotational and scaling information. When you apply this matrix to a vector, a process known as vector-matrix transformation, you can essentially rotate the vector or scale it any way you want. A 4 × 4 matrix has this same information with the addition of positional information in the last row of the 2D array. Figure 8.2 shows a visual of a matrix.

FIGURE 8.2. A 2D ARRAY AS A MATRIX, WHERE THE LAST ROW STORES THE X, Y, AND Z POSITIONAL INFO.

$$M = \begin{vmatrix} m1 & m2 & m3 & m4 \\ m5 & m6 & m7 & m8 \\ m9 & m10 & m11 & m12 \\ m13 & m14 & m15 & m16 \end{vmatrix}$$

When you transform a vector by a 4 × 4 matrix, you can apply scaling, rotations, and translations on any vector. Translation is the process of moving a vector from one location to another. Since a vector and vertex can be used the same way, you can transform the vertices of a 3D model using a matrix to change the model's position and orientation in the 3D world. For example, let's say you've created a 3D cube in an application such as Softimage XSI. The position of the vertices is stored in what is known as local or model space. This means the positions of the vertices are not related to anything other than the application in which the model was created. So if you create a box around the origin, you can create something in XSI that looks like Figure 8.3.

FIGURE 8.3. A CUBE CREATED IN XSI.



Now let's say you want to use this new model that you've created in a game. Let's also assume you will be placing more than one box in your 3D scene. You have the option of modeling the box in its unique position in XSI so that when the data is loaded, the boxes and other objects will be in their correct positions.

This method is inefficient and ineffective for the following reasons:

- It would be a waste of time to model an object more than once throughout a scene just so you can have more than one instance of the object.

- If the base object changes (let's say you want spheres instead of boxes), you'll have to repeat the process all over again by deleting all the objects you've created and starting again.

- What if the objects are dynamic and are supposed to move around the scene? How can this happen in code? The solution is the purpose of this discussion, as you will see.

- If the objects are made up of thousands of polygons, why load what is essentially the same object multiple times? This can lead to wasted memory and resources. If you have 100 instances of this object in a scene, that is 99 more objects than you need if the objects are all exactly the same.

- Current hardware supports hardware instancing, which generally means drawing an object with one draw call and mesh multiple times throughout the scene. If each object has its own unique vertex data, there is no way to take advantage of this feature.

When you model an object in model space, you only need to create an object once. You can then use a matrix to set the position of each instance of the object that is to appear in the scene. So, for example, you load the model once and create 10 matrices. Each of these matrices represents the instance of the object as it is to appear rotated, scaled, and translated in that position. In other words, you still have 10 objects, but those 10 objects all

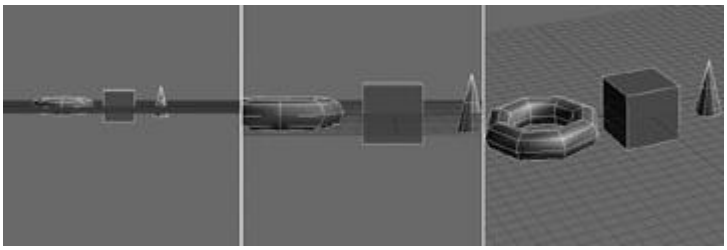
share the same model data. You simply change the matrix before drawing the object, and the scene will render as you've intended. This is the primary purpose of matrices.

Since each object can have its own matrix, each object can move, rotate, and be sized (scaled) independently of all other objects. When applying physics and collision detection, you can take into account the forces acting upon an object in relation to the world around it to create simulations that mimic what we observe in nature. To move one of these box examples around in a 3D scene, we simply change the X, Y, Z translation of the matrix that represents that object. This matrix is known as the model matrix. It is also sometimes referred to as the world matrix, as it defines where in the world the object is positioned and how it is rotated and scaled.

MATRICES AS CAMERAS

Matrices can be used for other effects as well. In 3D games there exists the idea of a virtual camera. This camera is actually a matrix called the view matrix that is applied to the vertices of a scene to rotate and position objects on a global level to give the illusion of a camera moving around in the virtual world. Take a look at Figure 8.4. On the left is the scene located at the default origin. The middle is the scene where the position of the view matrix has been moved forward by 20 along the Z axis, and the right is the scene from the middle rotated to the left around the Y axis. The combination of the model and view matrix is called the model-view matrix, and it is used to position and orient objects based on their own world transformation as well as being manipulated by the view matrix to simulate a 3D camera affect.

FIGURE 8.4. SCENE AT THE ORIGIN (LEFT), TRANSLATED (MIDDLE), AND ROTATED (RIGHT).



Another type of matrix called the projection matrix can further simulate a camera. This matrix is used to add orthogonal or perspective projection to the objects being rendered. Orthogonal projection essentially renders objects the same size on the screen regardless of how far back they are from one another. This is useful in 2D scenes or 2D elements such as menus since the depth of each object can be used to ensure the visual ordering of overlapping objects on the screen, but it is not realistic when rendering 3D scenes. In nature, objects appear smaller with distance. This is your perspective on the world around you. A perspective matrix essentially simulates this effect by scaling objects smaller as they move away from the virtual camera. The projection matrix also adds a field of view to the camera and far and near clipping planes. The near and far clipping planes of the projection matrix represent how far away from the camera an object can be and still be considered visible.

By combining the model, view, and projection matrices, you get model-view-projection (MVP) matrix. This matrix is commonly used in vertex shaders to transform the incoming vertices before moving on to the geometry shader (if one is present) and the pixel shader.

Keep in mind that the model matrix is used to position, scale, and rotate objects on an individual (personal) basis. The view matrix is used to further adjust the vertices of the geometry in a scene to simulate a 3D camera. The projection matrix is used to further simulate lens effects for the 3D virtual camera.

When you transform a model matrix from its local space, which is nothing more than the data you've loaded from a file created by an application such as XSI, you are converting the data from local space to world space. When you apply a view matrix to that, you are converting the data to view space. When you apply a projection matrix to that data, you are converting the data to screen space. These spaces are known as transformation spaces. In Chapter 13, when we discuss bump mapping, you will see another transformation space called texture space, which is used to transform a vector into texture space so that consistent bump mapping can be calculated for a surface.

The transformation space of a vertex is dictated by the matrices that have so far been applied to the vertex.

Direct3D uses a left-handed coordinate system, while OpenGL uses a right-handed system. In Direct3D you can change to a right-handed system, which essentially changes the direction of the positive and negative X and Z axes.

MATRIX OPERATIONS

Matrices are very useful in 3D games. You can combine matrices together using a process called matrix concatenation. Mathematically, this means multiplying matrices together. In Direct3D 10 you can use the `D3DXMATRIX` structure to represent a matrix, and you can use the multiplication symbol (*) to concatenate matrices together. Therefore, the model-view-projection matrix is essentially the model matrix times the view matrix times the projection matrix. The result is a single matrix that represents everything each of the matrices it is made up of. So, a single vector-matrix transformation can be used to move a vertex in local space to screen space. Optionally, you can transform the vectors by each matrix individually, which is what some of the demos in this book do in the beginning of the vertex shaders.

The DirectX SDK documentation specifies 34 matrix-related functions, which is a lot to cover all at once, especially considering that most of these functions will not be used for any demos in this book. In this chapter we will examine the functions relevant to the topics discussed in this book. We will look at additional functions as they arise in demos. We recommend that you read the DirectX SDK documentation for a brief overview of each of these matrix-related functions so that when you do need to use one, you can refer to the documentation and move on from there.

To start, a matrix that will have no effect on a vector is known as an identity matrix. Just like how adding a vector to another vector that has all zeros for the X, Y, and Z axes will not affect the original vector, transforming a vector by an identity matrix will have no effect on that vector. An identity matrix can be thought of as a default "empty" matrix. It is created by calling the `D3DXMatrixIdentity()` function. This function takes a single parameter, the output address to the matrix being set to an identity matrix. To test if a matrix is an identity matrix you can call the function `D3DXMatrixIsIdentity()`, which takes as a parameter the matrix to test and returns `true` if the matrix is an identity matrix or `false` if it is not.

To create a view matrix, you can call the function `D3DXMatrixLookAtLH()` to create a left-handed coordinate system view matrix or `D3DXMatrixLookAtRH()` to create a right-handed version. The `D3DXMatrixLookAtLH()` and RH functions take as parameters the output address of the matrix being created by the function call, the position of the camera, the location at which the camera is looking, and the direction that is considered up.

To multiply (concatenate) two matrices together you can use the * multiplication operator or the `D3DXMatrixMultiply()` function, which takes as parameters the output address of the matrix being created by this function call, the first matrix in the operation, and the second matrix in the operation.

To set a matrix's position (translation), you call the `D3DXMatrixTranslation()` function, which takes as parameters the output address matrix and the X, Y, and Z position that is being set in the matrix. The X, Y, and Z positions are floating-point values.

A matrix can be rotated by calling the `D3DXMatrixRotationAxis()` function, which takes as parameters the out matrix, the vector axis to rotate around, and an angle amount to rotate by specified in radians (not degrees). To use angles measured in degrees, you can use the DirectX macro `D3DXToRadian(degrees)`. To use this macro, you send the degrees in the parameter, and during compilation the macro will be replaced with the mathematical equation necessary to change degrees to radians.

Other functions you can use to rotate a matrix are `D3DXMatrixRotateX()`, `D3DXMatrixRotateY()`, and `D3DXMatrixRotateZ()`. These functions are used to rotate along a specific unit-axis while `D3DXMatrixRotationAxis()` takes a vector that can be used to specify one or more axes to rotate around at once in a single `D3DXVECTOR3` object. Each of these rotation functions takes as parameters the out matrix and a floating-point angle defined in radians.

The last rotation functions are `D3DXMatrixRotationQuaternion()`, which rotates a matrix by a quaternion, and `D3DXMatrixRotationYawPitchRoll()`. The quaternion will be discussed later in this chapter. To rotate along the yaw, pitch, and roll basically means that the order of rotation will occur on the Z axis (yaw), followed by the X (pitch) and Y (roll) axes. These terms should be familiar to anyone who has played or developed a flight simulator game.

To scale a matrix you can call the `D3DXMatrixScaling()` function, which takes as parameters the out matrix and the X, Y, and Z scales to apply to the matrix.

The last functions we will discuss deal with the projection matrix. Although only the left-handed versions of these functions will be discussed, keep in mind that each of these functions has right-handed equivalents. The orthogonal and perspective projection functions from the DirectX SDK are as follows:

```
D3DXMATRIX * D3DXMatrixOrthoLH(
    D3DXMATRIX *pOut,
```

```
    FLOAT w,
    FLOAT h,
    FLOAT zn,
    FLOAT zf
);

D3DXMATRIX * D3DXMatrixOrthoOffCenterLH(
    D3DXMATRIX *pOut,
    FLOAT l,
    FLOAT r,
    FLOAT b,
    FLOAT t,
    FLOAT zn,
    FLOAT zf
);

D3DXMATRIX * D3DXMatrixPerspectiveLH(
  D3DXMATRIX *pOut,
  FLOAT w,
  FLOAT h,
  FLOAT zn,
  FLOAT zf
);

D3DXMATRIX * D3DXMatrixPerspectiveFovLH(
  D3DXMATRIX *pOut,
  FLOAT fovy,
  FLOAT Aspect,
  FLOAT zn,
  FLOAT zf
);

D3DXMATRIX * D3DXMatrixPerspectiveOffCenterLH(
  D3DXMATRIX *pOut,
  FLOAT l,
  FLOAT r,
  FLOAT b,
  FLOAT t,
  FLOAT zn,
  FLOAT zf
);
```

Since each of these projection functions has overlapping parameters, we will discuss them all at once in the following list.

- `pOut` refers to the output address to the matrix that will store the result of the function call.
- `w` is the width of the view.
- `h` is the height of the view.

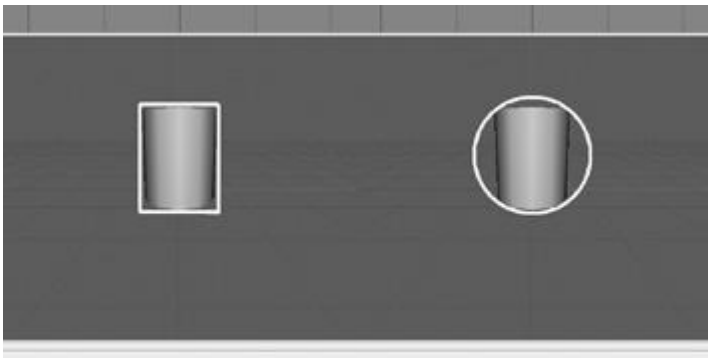
- `zn` is the near plane distance.
- `zf` is the far plane distance.
- `l` is the minimum value for the width (the minimum X of the view volume).
- `r` is the maximum value for the width (the maximum X of the view volume).
- `b` is the minimum value for the height (the minimum Y of the view volume).
- `t` is the maximum value for the height (the maximum Y of the view volume).
- `fovy` is the field of view of the camera specified in radians.
- `Aspect` is the aspect ratio, which can be width/height or whatever value you deem appropriate.

BOUNDING GEOMETRY

Collision detection and response is a very important topic in 3D video games. Models and objects in a video game are usually made up of thousands of polygons. The environments themselves can be in the high thousands or millions of polygons. When detecting collision between triangles, the operation is usually a very CPU-expensive process, even for just a single triangle, let alone thousands upon thousands of them. What is needed is a way to quickly detect collisions between objects, along with other factors such as visibility determination. This is where bounding geometry comes into play.

The idea behind bounding geometry is that the bounding geometry serves as a very simple representation of a more complex object. Bounding geometry is any simple shape that surrounds an object so that the object fits inside the bounding geometry's volume as tightly as possible. These shapes include spheres, boxes, ellipsoids, and so forth. An example of a bounding box and sphere, the most common types of bounding geometry used, is shown in Figure 8.5.

FIGURE 8.5. AN EXAMPLE OF A BOUNDING BOX (LEFT) AND BOUNDING SPHERE (RIGHT).



The need to simplify various tests such as collision detection is extremely important in video games because everything has to run as efficiently and effectively as possible. This requires performing tests as fast as possible to avoid slowing down the CPU and to get the most out

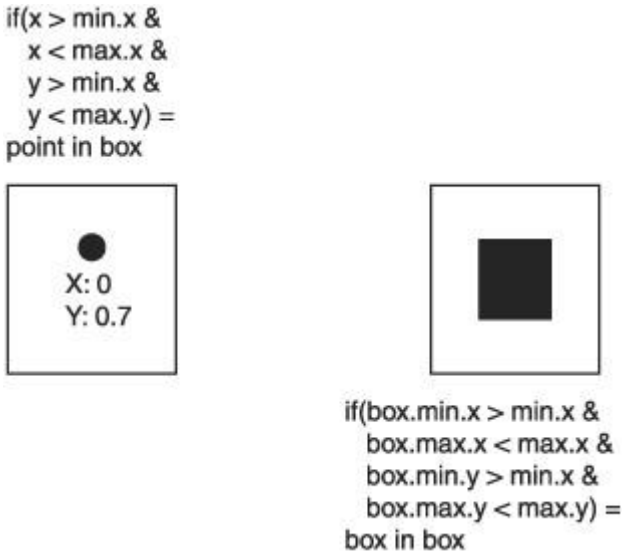
of the processing power of the machine. This is the purpose of using bounding geometry, which is significantly faster than working on the polygon level of models and objects. Mathematically, a box can be tested quickly for whether it is in view or is touching another box far faster than the triangles of a model, so using the simplified shape is a very fast substitution.

The DirectX SDK includes functions to create a bounding box and bounding sphere around a piece of geometry. In this section we'll discuss the DirectX bounding box and bounding sphere support.

BOUNDING BOXES

A box can be defined using two vectors, where the first vector stores the minimum X, Y, and Z values for the box and the second vector stores the maximum X, Y, and Z values. If you know the minimum X and maximum X, for example, you will be able to know where each of the box's corners is along the X axis. If you also know the minimum and maximum Y and Z values, you can generate a box out of them. To test if a point or another object intersects (touches) the box, you can test if the point or object falls entirely within this minimum and maximum range that defines the bounding box. Looking at a 2D example, you can see that the object being tested falls within both the X and Y axes of the bounding box, meaning the object is inside of it. This is shown in Figure 8.6. Objects with volume, such as another box, touch the bounding box only if any part of it is on the surface of the bounding box or inside it.

FIGURE 8.6. A 2D EXAMPLE OF A POINT (LEFT) AND AN OBJECT (RIGHT) FALLING WITHIN A BOX.



A box can be any size along the width, height, and depth axes, which are defined by the two vectors that represent the box. A cube, on the other hand, is a box that has the same value for the width, height, and depth, forming a perfectly even shape along all axes. To represent a cube, you only need to know the center position and the size of the cube. This size is used for the width, height, and depth.

To calculate the bounding box you can call the `D3DXComputeBoundingBox()` function. This function takes as parameters a list of points specified as `D3DXVECTOR3` 3D vectors, the total number of vertices in the list, the size of each vertex, and output addresses that will store the minimum and maximum values of the computed bounding box. The `D3DXComputeBoundingBox()` function has the following function prototype according to the DirectX SDK.

```
HRESULT D3DXComputeBoundingBox(
    CONST D3DXVECTOR3 *pFirstPosition,
    DWORD NumVertices,
    DWORD dwStride,
    D3DXVECTOR3 *pMin,
    D3DXVECTOR3 *pMax
);
```

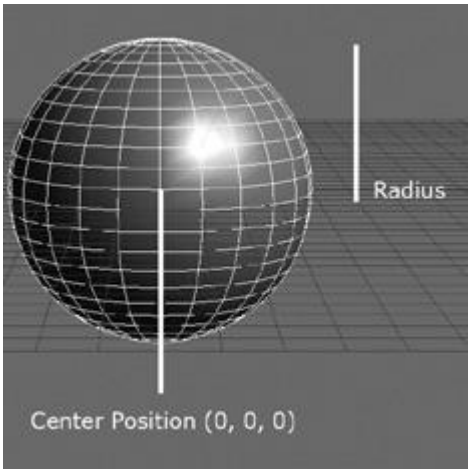
You can test if a ray hits the bounding box by calling the `D3DXBoxBoundProbe()` function. This function takes as parameters the minimum and maximum values of the bounding box, the position (origin) of the ray, and the direction vector of the ray. Rays will be discussed briefly later in this chapter. The function prototype for the `D3DXBoxBoundProbe()` function is as follows according to the DirectX SDK.

```
BOOL D3DXBoxBoundProbe(
    CONST D3DXVECTOR3 *pMin,
    CONST D3DXVECTOR3 *pMax,
    CONST D3DXVECTOR3 *pRayPosition,
    CONST D3DXVECTOR3 *pRayDirection
);
```

BOUNDING SPHERES

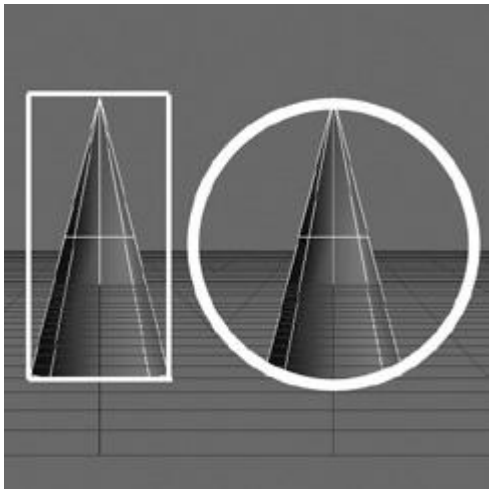
A bounding sphere can be defined as a center position and a radius, where a radius represents the size of the sphere from the center to the outer surface. This is shown in Figure 8.7. Keep in mind that the difference between the radius and the diameter is that the diameter is the size of the sphere from one end of the surface to the other side as it crosses the center. The radius is essentially half the diameter.

FIGURE 8.7. A SPHERE DEFINED BY A POSITION AND A RADIUS.



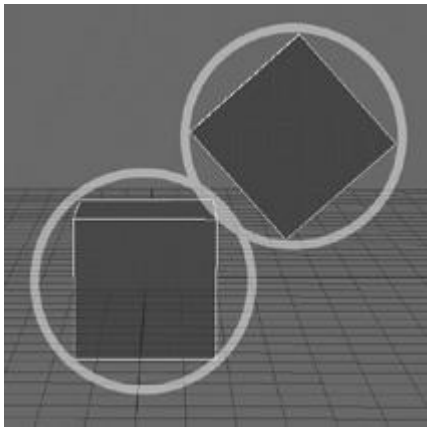
The benefit of using spheres is that mathematically you can test for things such as collisions and visibility faster than you can test boxes or other shapes. The downside to spheres is that depending on the shape, there can be more wasted space within the sphere's volume, or even the box. An example of this is shown in Figure 8.8, in which the same object has a box around it (left) and a sphere (right).

FIGURE 8.8. AN OBJECT THAT WASTES LESS SPACE USING A BOX (LEFT) THAN A SPHERE (RIGHT).



Because one shape (depending on the object) can lead to more wasted space than other shapes, using them can also be a little less accurate in various tests. For example, in Figure 8.9 two objects might register collision when testing only their bounding spheres when in reality they do not touch. There are solutions to this, which will be discussed in the next section, "Bounding Hierarchies."

FIGURE 8.9. OBJECTS THAT REGISTER AS COLLIDING EVEN THOUGH THEY ARE NOT.



To calculate a bounding sphere in DirectX, we can use the function `D3DXComputeBoundingSphere()`, which takes as parameters a list of 3D vectors that represent the entire model, the total number of vertices in the model, the size of each

vertex, and the output addresses that will store the center position of the model or bounding sphere and the bounding sphere's radius. The `D3DXComputeBoundingSphere()` function has the following function prototype according to the DirectX SDK.

```
HRESULT D3DXComputeBoundingSphere(
  CONST D3DXVECTOR3 *pFirstPosition,
  DWORD NumVertices,
  DWORD dwStride,
  D3DXVECTOR3 *pCenter,
  FLOAT *pRadius
);
```

You can test a ray for collision with the bounding sphere by calling the `D3DXSphereBoundProbe()` function, which takes as parameters the center position and radius of the sphere and the ray's position and direction. The `D3DXSphereBoundProbe()` function has the following function prototype according to the DirectX SDK.

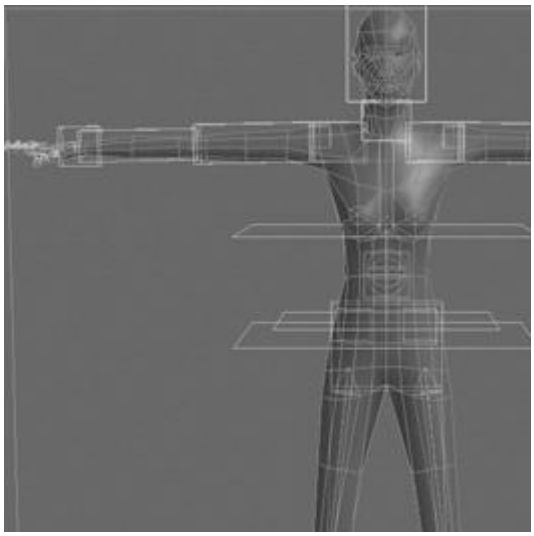
```
BOOL D3DXSphereBoundProbe(
  CONST D3DXVECTOR3 *pCenter,
  FLOAT Radius,
  CONST D3DXVECTOR3 *pRayPosition,
  CONST D3DXVECTOR3 *pRayDirection
);
```

BOUNDING HIERARCHIES

Using one of the simple shades as bounding geometry can lead to very fast tests such as for collision detection, but these tests are not the most accurate in many situations. Not only can there be wasted space within the volume of the shape, but it is difficult to determine much information from tests on a single bounding geometry. For example, in a first-person shooter game, what if you wanted to know if the player was shot in the arm, leg, head, and so on so that an appropriate animation can be played?

The solution is to use bounding hierarchies. The simple method requires you to break your model into pieces, each of which can have its own bounding geometry. For example, take a 3D character model. You can place bounding boxes around the arms, legs, feet, head, torso, hands, and so forth. This makes it much easier to detect the specific region of impact because more bounding geometry pieces are used in the model (see Figure 8.10). The amount of detail needed determines what parts of the model have their own bounding geometry. If you need to know which finger was hit, you could have bounding geometry around each finger of both hands, even if that might be overkill in most games.

FIGURE 8.10. A HIERARCHY OF BOUNDING GEOMETRY.



Also, you can efficiently combine a hierarchy of geometry with one large bounding box or sphere to improve performance when you must use the hierarchy for specific or more accurate tests. For example, you can use the larger bounding box or sphere of a model to determine if something is possible (e.g., possible collision, possible visibility, possible occlusion) and then use the bounding hierarchy to obtain more accurate results if that first test passes. This method can be very useful in games for which you must have more detailed information (such as a fighting game) if you don't want to do those more accurate tests unless they are necessary.

If you do not need to go as far as bounding geometry hierarchies, you can take just a bounding box and sphere and use those. For example, you can use the very fast bounding sphere to determine if something is possible and then test the bounding box if the first test passes to get slightly more accurate results. If that is accurate enough for the application you are developing, you could stop there and assume that whatever you are testing is true.

ADDITIONAL MATHEMATICS

We will now briefly discuss some common mathematics in video games. Because the topic of game math is complex and large, we highly recommend that you obtain a book on the subject if you are unfamiliar with or unsure of any math that is commonly used in video games. A good book on game math can also serve as reference material for those already familiar with the various topics.

THE RAY

A ray is defined by its starting position, called the origin, and a direction. A ray is a mathematical structure that begins at the origin and moves in a direction infinitely or for a specified length. For example, if you look forward, you can assume that the ray that defines your sight begins at your eyes (let's say right between your eyes for clarity), and the direction of the ray is the direction in which you are looking. In 3D the view ray of the camera starts at the camera's position and moves in the camera's direction. A visual of a ray is shown in Figure 8.11.

FIGURE 8.11. A VISUAL EXAMPLE OF A RAY.

Rays have many uses in video games. One straightforward example is the firing of a gun in a shooting game. When the player fires his gun, a ray starting at the gun's position with a direction that matches where the gun is pointing is created. You can use this ray to test for collision with the objects of the scene. If the ray's intersection test with an object passes, it can be determined that the gun has fired and hit that object. If that object is another character, such as a game enemy, you can then deduct health from the enemy character every time it is hit by such a ray until the enemy is considered dead. Once the enemy is dead, you can take the appropriate action, such as playing a death animation.

To create a ray you need two vectors—one for the origin position and one for the direction. You could also use a floating-point value that acts as the ray's length if the ray is not to be infinite. That way if the ray hits an object that is beyond the length of the ray, it can be assumed that the ray does not reach the object. In a shooting game this can be used to keep players from shooting objects far away across the map—for example, keeping weapons that are suppose to act as pistols from having infinite ranges like an endless sniper bullet.

THE QUATERNION

A quaternion is used to represent rotations. A quaternion's structure is made up of four floating-point values just like a 4D vector. In a quaternion the structure is made up of W, X, Y, and Z, while a 4D vector is made up of a X, Y, Z, and W. In other words, in mathematics a quaternion usually specifies the W first.

Because a quaternion uses only four floating-point values instead of 16 like a matrix, the quaternion can be more efficient to use in terms of memory storage. Not only can you save memory using essentially a quarter of the size of a matrix, but mathematically a quaternion can be calculated more quickly than a matrix. The benefits to using a quaternion are enough to make them useful in most applications.

Assuming scaling is not used, if you have objects that have a rotation and a position, you can use essentially a 3D vector for the position and a quaternion (which has the same memory footprint as a 4D vector) instead of 16 floating-point values. The memory savings can add up, especially for mobile devices, in which memory is not as abundant as in PCs. Also, quaternion calculations are faster, so this can lead to performance boosts when computing things such as animations. When sending the information to the graphics device in Direct3D, you can convert the quaternion to a matrix so that the information can be used by the vertex shader to transform the vertices appropriately. Also, a quaternion does not suffer from gimble lock like a matrix. Gimble lock occurs when two of three axes that are used to compensate for rotations are moving toward the same direction, which occurs out of an error from rotation calculations. For example, rotating can cause the X and Y axes to be pointing in the same direction—but as they are not actually the same direction, any object in motion will not rotate as it should. In other words you can warp one axis to point in the direction of another inadvertently.

GAME PHYSICS

Game physics is a very complex and downright cool part of many modern 3D games. Physics in games includes applying forces such as gravity and friction to 3D objects, the force of objects acting on each other, wind, and so forth. Physics can also include collision response, where the force of two objects colliding in virtual space can cause new forces to

be applied to objects to create the kind of simulation gamers expect to see. For example, if the player hits a box with a rocket from a rocket launcher, the resulting force of that collision is expected to be applied to the box, and any physics updates that occur on the box should send it flying through the air until it reaches its resting position as the force dies down. The same can be said for the friction of that box against the ground as it tumbles around the environment. The friction as it collides with other surfaces is what will eventually cause the object to stop, unless the environment is a space-like one where there is little or nothing to stop an object in motion.

Game physics could be book on its own but is beyond the scope of this book. However, when you are creating games that could benefit from this feature, we highly recommended that you research this subject because it can lead to very impressive and realistic results that can add a great deal of value to your game.

SUMMARY

The use of mathematics in video games is very important to programmers looking to enter the video game development industry. The current level of your math skills will determine how much work you have to do to become competent in this area. But make no mistake about it—having a firm understanding of math is essential to game developers. Many areas of game development require math skills, and there is no way to avoid learning this subject. In this chapter we've briefly discussed a few topics that have functions and structure in the DirectX SDK, but to make it far in the video game industry we highly recommended picking up one or more references on the topic.

The following elements were discussed in this chapter:

- Vectors
- Matrices
- Transformations
- Coordinate systems
- Rays
- Planes
- Quaternion rotations
- Virtual cameras
- Bounding geometry

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What are vectors?

2. What structures does the DirectX SDK offer for vectors?

3. What is a plane? What main purpose do planes serve as described in this chapter?

4. What is a matrix?

5. What is the difference between a 3 × 3 and 4 × 4 matrix?

6. What is bounding geometry and how is it used in computer graphics?

7. What coordinate system does Direct3D use traditionally?

8. What is the model matrix?

9. What is the view matrix?

10. What is the projection matrix?

11. What is the MVP matrix?

12. What three properties does Direct3D use to create a view matrix that represents a camera?

13. What is a ray, and what two components make up a ray object?

14. How can you limit a ray's infinite direction?

15. What is a quaternion used for? List two benefits of using quaternion rotations versus matrices.

9. SOUND IN DIRECTX

In This Chapter

- Overview of Microsoft Audio Technologies
- Direct Sound
- XACT3
- XAudio2

In modern video games, sound is as important to the virtual experience as all the other areas of the game. Audio is used for so many different things that some gamers might take

it for granted. This includes ambient music, weapon sound effects, character voice chatter, animal sounds, water splashing, footsteps, and hundreds of other sound effects that can exist in a virtual world. For many years the audio in video games has been very important to the experience, and sound can be just as impressive as the game's graphics, game-play, and so forth.

The purpose of this chapter is to demonstrate how to play sound using XACT3 and XAudio2, each of which are sound APIs that can be found in the DirectX SDK. XACT3 and XAudio2 are recent additions to the DirectX SDK, while Direct Sound has existed unchanged since DirectX 9.0. These days, you will find all you need in XACT3 for high-level audio control and in XAudio2 for low-level audio control. XACT3 and XAudio2 can both be used on the PC and the Xbox 360.

XACT is also recognized by Microsoft XNA Game Studio.

OVERVIEW OF MICROSOFT AUDIO TECHNOLOGIES

The audio technologies that Microsoft provides include a few APIs and tools that are used to play and manipulate sound on Windows-based PCs and Xbox 360 video game consoles. These audio APIs build upon each other, so they are not technically different technologies, but rather are different levels of audio APIs. For example, XAudio2 is a low-level sound API. Everything you can do with sound can be done with this API with the utmost control. XACT3 is an API and a GUI tool that provides high-level control over sound, and it is built on top of XAudio2. In other words, XACT3 is high-level enough that a lot of work happens behind the scenes through XAudio2, while if you use XAudio2, you have to directly do that work yourself. XACT3 uses XAudio2 internally and essentially saves you the work of using XAudio2 directly with the trade-off of low-level control.

You can view the various audio systems' documentation in the DirectX SDK, XNA SDK, or XDK (Xbox 360 Development Kit) documentation.

XACT3 and XAudio2 can play the following audio formats:

- Pulse code modulation (PCM)
- Adaptive differential pulse code modulation (ADPCM) on Windows
- XMA on Xbox 360
- xWMA (a subset of Microsoft's WMA)

Additional Microsoft audio technologies include X3DAudio, XAPO, XAPOFX, and the XMAEncoder Library. X3DAudio is used by XAudio2 and XACT3 to play sounds in 3D—that is, to play audio so that it sounds like it is emanating from a specific position rather than being

ambient and global. An example of X3DAudio for XAudio2 called XAudio2Sound3D can be found in the DirectX SDK Sample Browser.

XAPO is an API used to create audio effects for XACT3 and XAudio2. XAPOFX is a library of XAPO sound effects ready to be used with audio.

The XMAEncoder Library is a statically linked library that developers can use to give applications the ability to encode XMA content. The XMAEncoder is available in the Xbox Development Kit.

DIRECT SOUND

Direct Sound is a sound API that is part of the DirectX SDK. Direct Sound is deprecated, which means it is no longer being updated and presumably will eventually be dropped from the DirectX SDK. Although you can still use Direct Sound, it has been replaced with XAudio2, which will be discussed later in this chapter. Other deprecated audio technologies include the original XAudio, which was used in the original Xbox video game console and has also been replaced with XAudio2 (now used for the Xbox 360 and PC), and XACT, which has been replaced with XACT3. This information is according to Microsoft's documentation on their audio technologies, which can be found in the DirectX SDK documentation.

WHY NOT USE DIRECT SOUND?

Long ago there were two audio APIs in the DirectX SDK: Direct Music and Direct Sound. Direct Music allowed low-level control over audio playback, and Direct Sound was a higher-level API that gave developers an easy way to play sound in their applications. Direct Music was eventually dropped (and merged with Direct Sound), and Direct Sound became the main audio API in DirectX. When the Xbox video game console hit the market, XAudio was the main API used for audio on that console. XAudio was eventually replaced with XAudio2, which was packaged with the DirectX SDK. XACT3 is built on top of XAudio2 and gives developers a ready-made high-level API and tool for audio playback and control.

The big question is why would you choose XAudio2 over XACT3? Put in simple terms, XAudio2 is good for developers looking to build something like XACT3, while XACT3 is a ready-to-go high-level tool and API that can be used on the PC and the Xbox 360. In addition to being available in the DirectX SDK, XACT is available in the XNA Game Studio SDK. For advanced developers the low-level control of XAudio2 might prove to be attractive enough to use it.

Another way to think about it is that XACT3 is useful for developers who want a ready-made audio content creation API and tools instead of developing their own implementation. It is also possible to use XACT3 and XAudio2 at the same time if the need arises—for example, if you need signal processing, mixing, and so forth, which XACT3 does not offer but XAudio2 does.

XACT3

XACT3 is available in XNA, which is a C#-based game development tool, and C++. You can obtain XACT3 by installing either the XNA SDK or the DirectX SDK. XACT3 is Microsoft's high-level audio technology that allows programmers and sound designers to use the same code and audio content files on both Windows-based PCs and Xbox 360 video game consoles. XACT3 is composed of an audio API and as a stand-alone GUI toolset used to create the audio files that are used by the audio API. When you compile audio content with the XACT toolset, you can create output audio files for both the PC and the Xbox 360.

When using XNA, you don't have to compile the files using the XACT toolset because the XNA content pipeline does that job for you and prepares the necessary files whether you are compiling for the Xbox 360 or Windows PC platforms.

For those familiar with previous versions of XACT, XACT3 has had the following features added to it.

- Uses XAudio2 internally
- Supports the xWMA compressed format, which is a subset of WMA
- Allows filters to be applied to sounds

XACT3 TOOLS

The XACT3 toolset is called the Microsoft Cross-Platform Audio Creation Tool, and it is a GUI application that is used, as mentioned earlier, to create the audio files that will be loaded and used by the XACT3 API code base. This tool creates files that can be used by either the Xbox 360 or Windows-based PCs (XP and Vista). With the XACT3 tool you can group sound files into cues that can be played any time in a game, and you can set various properties of each cue such as pitch and volume. XACT3 has both a GUI tool and a command-line tool. Both tools perform the same task, but the GUI tool is much more convenient to use.

XACT3 allows developers to organize audio content into packages called banks. Later we will talk about these banks, which include sound and wave banks, and how to create them in XACT3. The audio content itself includes the following audio formats.

- WAV
- AIFF
- ADPCM
- XMA
- xWMA

There is one thing to consider when working with audio files created by XACT3. The Windows PC and the Xbox 360 use different byte ordering for variables. In other words the number 12 on the Xbox 360 would not read as 12 on a Windows PC without reversing the bytes of the integer. This is because the Xbox 360 uses big-endian order because of its PowerPC hardware, while Windows-based PCs uses little-endian order, which is the byte ordering used by x86 processors. Endian order is an important topic for programmers and engineers who create cross-platform applications, such as writing an application that works on Windows and porting it to Mac (PowerPC-based Mac, that is).

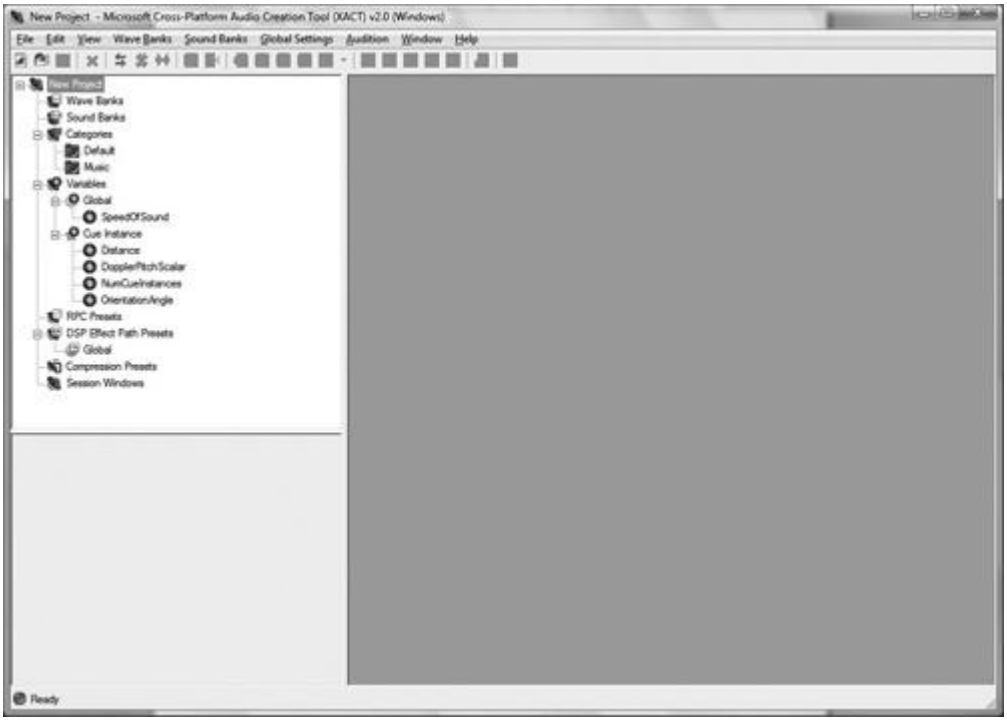
Since this book focuses on DirectX 10 running on Windows-based PCs, the issue of endian order will not arise. Fortunately, if the issues ever does arise, all that needs to be done to translate between byte ordering is to read a variable (such as a float, integer, etc.) from a file, cast that variable to a character pointer, and use array indexes to swap the first element with the last and the second element with the third. In other words, reverse the array of characters. This would only be done if you know you were reading a file that was

written in an incompatible endian order, which is the responsibility of the programmer since there is no way of knowing what endian order a file's data is in unless you purposely write some type of flag in your custom file formats for this purpose (e.g., a single byte where 0 means little endian or 1 means big endian). The same holds true for data sent over a network between two machines that use different endian orders.

The XACT3 toolset creates files in both endian orders. Since the Xbox 360 and Windows-based PCs use different endian orders, this is convenient and means you do not have to write code to translate the byte ordering between the platforms. The Windows files are saved in a folder called Win, and Xbox 360 files are saved in a folder called Xbox when you build XACT3 projects.

A screenshot of the XACT3 GUI tool is shown in Figure 9.1. You can launch the tool from the Start menu on Windows XP or Vista by navigating to the DirectX SDK or XNA SDK folder, the DirectX Utilities folder if you are using the DirectX SDK, or the Tools folder if you are using XNA or Microsoft Cross-Platform Audio Creation Tool (XACT).

FIGURE 9.1. SCREENSHOT OF THE XACT GUI TOOL.



CREATING XACT AUDIO PROJECTS

To create a new audio project, the first step is to open the XACT tool and select New Project from the File menu. A dialog box should appear prompting you to choose a name for your project for the project file that will be saved and the location where you want this project saved (see Figure 9.1). Save the project and name it TestXACT. If you are following along with the creation of this chapter's XACT demo, create a folder called XACT in your My Documents folder (or wherever you wish) and save the XACT3 audio project in this folder. When you save the audio project, there should be a TestXACT.xap file, which is the XACT

project file, and two folders titled Win (for Windows) and Xbox (for Xbox 360) that were created by the tool as a result.

On the CD-ROM is a demo called XACT in the Chapter 9 folder. In this chapter we will create this demo, and it will load the XACT audio files we are about to create. Before discussing the code, we will start by creating the audio content so that data is ready when we begin coding the demo application.

CREATING XACT WAVE BANKS AND SOUND BANKS

With the project created, you can create a wave bank and a sound bank. A wave bank is used to take multiple audio files and package them into a single file. These wave bank files have the extension .XWB and allow developers to manage a single file rather than many files. A sound bank has the extension .XSB and packages multiple cues into a single file. The file format is documented for wave banks for those interested in using it for purposes outside of XACT, but the sound bank files are not documented.

A cue in XACT3 is like an action. You play cues in XACT, and a cue has properties associated with it that include the audio sound from the wave bank that it will play when called and its volume and pitch. This is useful because you can have multiple cues that are used to play a sound different ways in a game. For example, you can take the sound of an engine and create multiple cues that alter how that engine sounds so that in a game the one sound clip is used to create different sound effects.

The two types of wave banks are in-memory and streaming. In-memory is used to store sounds that are loaded and used in memory, while streaming is used to dynamically load audio from the wave bank while it is being played. Streaming wave banks are great for playing audio files that are large in size and in length. In this book we'll focus on in-memory wave banks, but once you are familiar with playing audio in XACT3, you can follow the streaming wave bank XACT3 sample from the DirectX SDK Sample Browser to see how to utilize a wave bank for streaming.

To begin, create a new wave bank and sound bank by following these instructions.

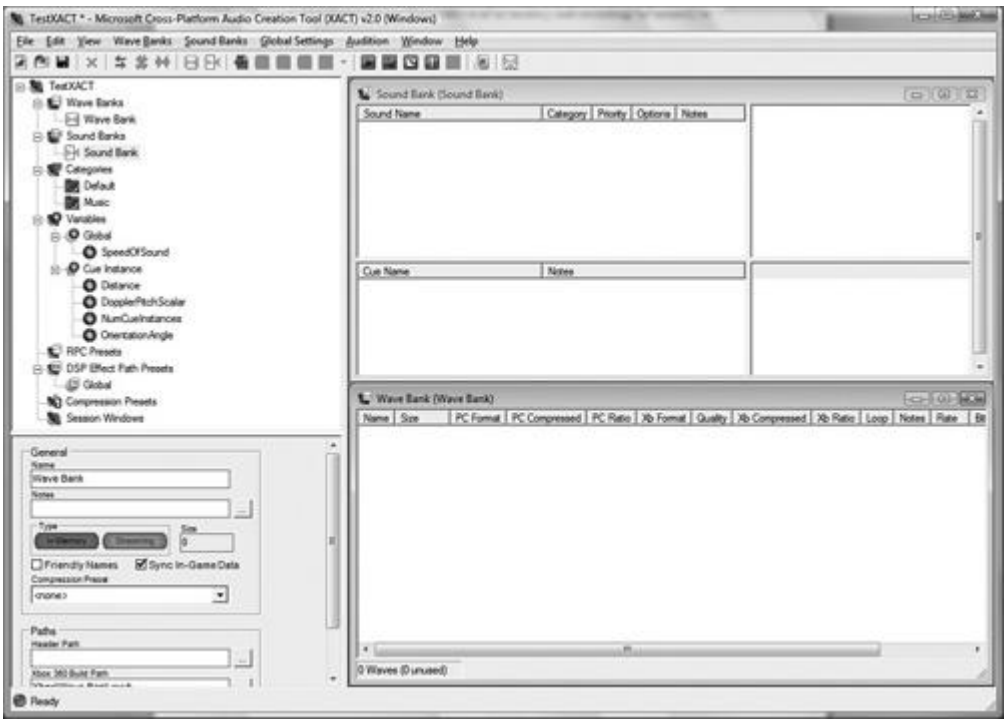
1. Select Wave Banks > New Wave Bank from the menu.

2. Select Sound Banks > New Sound Bank from the menu.

3. Select Window > Tile Horizontally from the menu to organize the newly created windows.

You should have a window that appears similar to Figure 9.2. From this point you can manually add audio files to the wave bank and then add cues to the sound bank.

FIGURE 9.2. XACT AFTER CREATING WAVE AND SOUND BANKS.



To add a sound to the wave bank, choose Wave Banks > Insert Wave File(s). From the dialog box choose the file you want to add, and it will insert a new entry in the wave bank window. To create a sound in the sound bank, select Sound Banks > New Sound. To create a new cue, select Sound Banks > New Cue from the menu. To associate the audio file from the wave bank to the sound in the sound bank, simply drag and drop the audio from the wave bank window to the sound entry in the sound bank. To associate a sound with a cue, you do the same thing by dragging the sound entry from the top of the sound bank to the cue. Remember, the cue is used by XACT to access the sound to play it, stop it, and so forth. The sound in the sound bank stores various properties such as volume, pitch, and so on. The audio in the wave bank is the actual audio data.

You can set the name for the sound and wave banks in the properties section of each.

There is a shortcut to creating a sound and cue in the sound bank. Once you've inserted the audio in the wave bank, you can drag and drop that entry to an empty region in the cue window and, when you release the mouse button, an entry for the sound and cue will be automatically created with the same name as the audio file that was in the wave bank. This method is commonly used since you have to have all three to play sounds in XACT, and this method is faster than creating each entry manually, especially if they use the same name anyway.

At this point save the project and select File > Build to build the project. Once it is built (if you're developing it for Windows), you will have a file for the sound bank, a file for the wave bank, and a file with the extension .XGS. The XGS file stores global settings and variables that XACT3 can load and use. You can set the variables in the variables section of the properties panel. You can set variables such as the speed of the sound, the number of instances, the orientation angle, and so forth. You don't need these settings to play a

sound, but you can use them, which is done in the DirectX SDK Sample Browser in a demo called XACT Tutorial 3: Categories and Variables.

XACT3 DEMO

XACT3 is a great tool that allows developers to focus on game-play rather than on audio technology and hardware. XACT3 is so easy to use that all it takes to get simple sounds up and playing is to invoke the audio by cue name. At the minimum this could translate to a line of code to play a sound once it has been loaded. We recommend that you take the time to open up XACT3 and explore the GUI tool to see what you can do with it. Once you are comfortable using it, you can do a lot with audio in your gaming projects. As you become more advanced in game development audio, you might need to use XAudio2 since it offers a lower level of control than XACT3 does.

A demo called XACT can be found on the accompanying CD-ROM in the Chapter 9 folder. In this section we will cover the XACT3 API as we cover the XACT demo.

The header file for the XACT3 API is <xact3.h> and is part of the DirectX SDK, so no additional setup is necessary to begin coding with XACT. In code you'll need three objects: a wave bank, a sound bank, and an audio engine.

The wave bank is represented by the DirectX type `IXACT3WaveBank`, and the sound bank is represented by `IXACT3SoundBank`. When you load wave and sound bank files, you load their contents into objects of these two types. The audio engine is an object that is created to process and control everything dealing with XACT3. This object is of the type `IXACT3Engine`. For readers familiar with earlier versions of XACT, the only difference in these names is the addition of the number 3 after XACT (for example, `IXACT3Engine` instead of the previous `IXACTEngine`).

In the XACT demo's main source file these three objects are created and defined in the global section. A structure was created to hold the wave bank and sound bank in one object along with void pointers. The void pointers will be memory mapped data pointing to the file's data, which we'll discuss later in this section. The global section from the XACT demo is shown in Listing 9.1.

LISTING 9.1. THE GLOBAL SECTION OF THE XACT DEMO'S MAIN SOURCE FILE

```
struct stXACTAudio
{
    IXACT3WaveBank *m_waveBank;
    IXACT3SoundBank *m_soundBank;
    void *m_waveBankData;
    void *m_soundBankData;
};

stXACTAudio g_xactSound;
IXACT3Engine *g_soundEngine = NULL;
```

A few functions are created in the demo to make the setup and loading of XACT3 and the XACT files straightforward. In the first function, called `SetupXACT()`, the first step is to initialize COM with a call to `CoInitializeEx()`.COM must be initialized to use COM libraries, which include many DirectX SDK libraries like XACT.

The next step is to create the XACT3 audio engine. This is done with a call to `XACT3CreateEngine()`, and it takes as parameters the creation flags and a pointer to

the IXACT3Engine object that will be created by the function. The creation flags can be 0 to specify no additional flags, or they can be XACT_FLAG_API_AUDITION_MODE to create the audio engine in audition mode or XACT_FLAG_API_DEBUG_MODE to specify debug. You can use the logical OR operator to combine the two flags.

The remainder of the SetupXACT() function initializes the audio engine and loads the wave and sound banks. The loading of the wave and sound banks is done in separate functions that will be discussed later. The initialization of the XACT audio engine is done by calling the IXACT3Engine object's Initialize() function. This function takes runtime parameters that are specified by the XACT_RUNTIME_PARAMETERS structure. This structure can be seen as follows.

```
typedef struct XACT_RUNTIME_PARAMETERS {
    DWORD lookAheadTime;
    void *pGlobalSettingsBuffer;
    DWORD globalSettingsBufferSize;
    DWORD globalSettingsFlags;
    DWORD globalSettingsAllocAttributes;
    XACT_FILEIO_CALLBACKS fileIOCallbacks;
    XACT_NOTIFICATION_CALLBACK fnNotificationCallback;
    PWSTR pRendererID;
    IXAudio2 *pXAudio2;
    IXAudio2MasteringVoice *pMasteringVoice;
} XACT_RUNTIME_PARAMETERS, *LPXACT_RUNTIME_PARAMETERS;
```

Some of the runtime parameters deal with XAudio2, which will be discussed later in this chapter. The SetupXACT() function is shown in Listing 9.2.

LISTING 9.2. THE SETUPXACT() FUNCTION FROM THE XACT DEMO

```
bool SetupXACT(char *waveBank, char *soundBank)
{
   ZeroMemory(&g_xactSound, sizeof(stXACTAudio));

   if(FAILED(CoInitializeEx(NULL, COINIT_MULTITHREADED)))
      return false;

   if(FAILED(XACT3CreateEngine(XACT_FLAG_API_AUDITION_MODE,
      &g_soundEngine)))
      return false;

   if(g_soundEngine == NULL)
      return false;

   XACT_RUNTIME_PARAMETERS xparams = {0};
   xparams.lookAheadTime = 250;

   if(FAILED(g_soundEngine->Initialize(&xparams)))
      return false;

   if(!LoadWaveBank(waveBank))
      return false;
```

```
   if(!LoadSoundBank(soundBank))
      return false;

   return true;
}
```

The loading of the wave and sound banks is performed in the `LoadWaveBank()` and `LoadSoundBank()` functions defined in the demo's main source file. To load the file's data, the demo uses memory-mapped file handling. This is a fast way to load memory data into XACT, and the functions are Win32 functions that are fairly straightforward to understand.

To start, a call to `CreateFile()` is used to create a file handle. This function takes the file's name, the access flag that specifies the type of object, the shared flag that specifies how subsequent calls are allowed to access the file while the file handle is still active, security attributes (which are ignored, so they're not used and can be set to `NULL`), creation flags, attributes flags, and the template file, which is also not used and therefore can be set to `NULL`.

The next part of the function retrieves the file size with a call to `GetFileSize()`, which takes the file handle and a pointer to a variable where the high-order double-word of the file size is returned. The last parameter can be set to `NULL`, and the return value of the function will return the size of the file.

The next step is to create the file mapping by first calling `CreateFileMapping()`, which takes as parameters the file's handle, the optional security attributes, the file protection with the view, the high order of the maximum file size, which can be 0, the low order of the maximum file size, which can be the file size we read before, and the name of the object, which is optional. The next step in the file mapping is to call `MapViewOfFile()`, which takes the file mapping handle, the access flags, the high and low offsets, and the number of bytes to map. If the number of bytes to map is 0, then the mapping extends to the end of the file.

With a pointer to the mapped file data, the XACT3 audio engine is ready to load its contents. This is done with a call to `CreateInMemoryWaveBank()` and `CreateInMemorySoundBank()` to create in-memory banks. Both functions take as parameters void pointers to the mapped file data, the size of the file, a flag that can be 0 or `XACT_FLAG_API_CREATE_MANAGEDATA` to specify that the data is freed when the wave bank is released (which we must do since we are using mapped data and must unmap it first), memory buffer allocation attributes, and the out pointer address to the bank that will be created. The `LoadWaveBank()` and `LoadSoundBank()` functions are shown in Listing 9.3.

LISTING 9.3. THE XACT DEMO'S `LOADWAVEBANK()` AND `LOADSOUNDBANK()`

```
bool LoadWaveBank(char *fileName)
{
   HANDLE file = CreateFile(fileName, GENERIC_READ,
                            FILE_SHARE_READ, NULL,
                            OPEN_EXISTING, 0, NULL);

  if(file == INVALID_HANDLE_VALUE)
     return false;
```

```
  DWORD fileSize = GetFileSize(file, NULL);

  if(fileSize == -1)
  {
     CloseHandle(file);
     return false;
  }
  HANDLE mapFile = CreateFileMapping(file, NULL, PAGE_READONLY,
                                     0, fileSize, NULL);

  if(!mapFile)
  {
     CloseHandle(file);
     return false;
  }

  void *ptr = MapViewOfFile(mapFile, FILE_MAP_READ, 0, 0, 0);

  if(!ptr)
  {
     CloseHandle(mapFile);
     CloseHandle(file);
     return false;
  }

  g_xactSound.m_waveBankData = ptr;

  if(FAILED(g_soundEngine->CreateInMemoryWaveBank(
     g_xactSound.m_waveBankData,
     fileSize, 0, 0, &g_xactSound.m_waveBank)))
  {
     CloseHandle(mapFile);
     CloseHandle(file);
     return false;
  }

  CloseHandle(mapFile);
  CloseHandle(file);

  return true;
}


bool LoadSoundBank(char *fileName)
{
   …

   g_xactSound.m_soundBankData = ptr;

   if(FAILED(g_soundEngine->CreateSoundBank(
      g_xactSound.m_soundBankData,
      fileSize, 0, 0, &g_xactSound.m_soundBank)))
```

```
    {
        CloseHandle(mapFile);
        CloseHandle(file);
        return false;
    }

    CloseHandle(mapFile);
    CloseHandle(file);

    return true;
}
```

The last function in the demo is the `main()` function. In this function XACT is set up, and the wave and sound banks are loaded before the audio is played. To play the sound a sound cue is obtained by calling `GetCueIndex()` on the sound bank object. This function takes as a parameter the name of the sound cue, which was specified when you created the cue in the XACT GUI tool, and it returns the index of the sound as an `XACTINDEX` object.

The sound itself is played by calling the sound bank's `Play()` function. The `Play()` function takes as parameters the cue index as an `XACTINDEX` variable, playback flags, offset start time in milliseconds, and, optionally, a pointer to the address of an `IXACTCue` object that will be returned by this function.

Alternatively, you can create an `IXACTCue` *object and obtain a cue using that. Therefore, to play a sound you just call* `Play()` *on the* `IXACTCue` *object. To stop it you call* `Stop()`*, and so on.*

XACT3 requires frequent updates to the audio engine to work properly. This means that in the demo we need to specify a loop that keeps calling `DoWork()` often enough for the audio to play. To accomplish this the XACT demo calls the audio engine's `DoWork()` function inside a loop. Once the sound has finished playing, the loop breaks, and the application proceeds to exit. To determine if the sound is playing, we get the audio state of the sound bank by calling `GetState()`, which returns an unsigned long variable representing the state. We can then test this state for the flag `XACT_CUESTATE_PLAYING` to see if the sound bank is playing audio. When it stops playing audio, the loop terminates, so be sure that inside the XACT3 GUI tool you don't specify any sounds to loop infinitely for this demo.

The remainder of the `main()` function frees all resources and quits. This means the audio engine must be released by first calling the `Shutdown()` function of the object and the object's `Release()` function to free it. It also means we must call `CoUninitialize()` to uninitialize COM, and we must call `UnmapViewOfFile()` to unmap the file data that we obtained when we loaded the wave and sound banks. The `main()` function from the XACT demo is shown in Listing 9.4.

LISTING 9.4. THE XACT DEMO'S `MAIN()` FUNCTION

```
int main(int args, char* argc[])
{
```

```
   cout << "XACT Demo: Playing clip.wav" << endl << endl;
   cout << "Demo will end when the sound is done." << endl <<
endl;

   if(!SetupXACT("Win/Wave Bank.xwb", "Win/Sound Bank.xsb"))
      return 0;
   XACTINDEX g_clipCue = g_xactSound.m_soundBank->GetCueIndex(
      "clip");

   unsigned long state = 0;

   do
   {
      g_soundEngine->DoWork();

      if(!(state && XACT_CUESTATE_PLAYING))
         g_xactSound.m_soundBank->Play(g_clipCue, 0, 0, NULL);

      g_xactSound.m_soundBank->GetState(&state);
   } while(state && XACT_CUESTATE_PLAYING);


   if(g_soundEngine)
   {
      g_soundEngine->ShutDown();
      g_soundEngine->Release()
   }

   if(g_xactSound.m_soundBankData)
   {
      UnmapViewOfFile(g_xactSound.m_soundBankData);
      g_xactSound.m_soundBankData = NULL;
   }

   if(g_xactSound.m_waveBankData)
   {
      UnmapViewOfFile(g_xactSound.m_waveBankData);
      g_xactSound.m_waveBankData = NULL;
   }
   CoUninitialize();

   return 1;
}
```

XAUDIO2

XAudio2 is the Direct Sound replacement for Windows developers and is an enhanced version of the XAudio API that Xbox developers have been enjoying for some time. On the CD-ROM you will find the XAudio2 demo in the Chapter 9 folder. In this chapter we will create a demo that will play a sound file once and then exit. This demo will show you how to get XAudio2 up and working to play sound inside any application.

XAudio2 does not have a way to detect and convert audio files between incompatible endian orders. This means that if you are working directly with XAudio2 on Xbox 360 and Windows, you must handle endian order carefully.

XAUDIO2 DEMO

Like XACT3, XAudio2 has an interface that you create to use XAudio2. This interface is called IXAudio2, and it is created by calling the SDK function `XAudio2Create()`. On the Xbox 360 this is an actual API function, while on Windows, according to the DirectX documentation, it is a convenient inline function defined in XAudio2.h. `XAudio2Create()` has the following function prototype and takes as parameters the IXAudio2 object that will be created, creation flags (defaults to 0 or `XAUDIO2_DEBUG_ENGINE` for debug mode), and an audio processor that specifies which CPU XAudio2 should use, which has a default value of `XAUDIO2_DEFAULT_PROCESSOR`.

```
HRESULT XAudio2Create(
  IXAudio2 **ppXAudio2,
  UINT32 Flags = 0,
  XAUDIO2_PROCESSOR XAudio2Processor = XAUDIO2_DEFAULT_PROCESSOR
);
```

On the Xbox 360, XAudio2 is implemented as a statically linked library, while on Windows it is a COM object implemented by a dynamic link library.

XAudio2 uses something known as voices to manipulate and control audio. There are three types of these voices in the XAudio2 API: source voices, submix voices, and mastering voices. A source voice is used to send sound data to the other types of voices, and it represents an audio stream of data. A submix voice is used to process audio data from a source voice to perform various effects (e.g., sample rate conversion) and can also be used as an input voice to another submix voice or to a mastering voice. A mastering voice is the voice that is audible, and it sends that data it receives from source and submix voices to the audio hardware. The mastering voice is the only voice that allows you to hear anything, so you must create this voice in XAudio2 to hear anything.

As far as the basics of XAudio2 are concerned this is essentially what you need to play audio in the API. In the XAudio2 demo's main source file the function calls `CoInitializeEx()` because XAudio2 is a COM object in Windows. It creates the XAudio2 engine, and it creates the mastering voice that will play the actual sound. In the demo the loading and playing of the actual file are done in a function called `PlayPCM()`, which will be discussed later in this section.

The creation of the mastering voice is done with a call to `CreateMasteringVoice()`, which takes as parameters an address to an `IXAudio2MasteringVoice` object that will store the created voice object, the audio channels, the audio sample rate, flags for the voice (which must be set to 0), the output device index the voice will use, and an optional audio

effects chain using the structure XAUDIO2_EFFECT_CHAIN. The audio channels are set to XAUDIO2_DEFAULT_CHANNELS and default to 5.1 surround on Xbox 360. In Windows, XAudio2 attempts to determine the speaker configuration.

The main() function in the XAudio2 demo is shown in Listing 9.5. To recap, the function initializes COM, creates the audio engine, creates the mastering voice, loads and plays the sound with a call to PlayPCM() that will be implemented next, and exits the application after releasing the audio engine and uninitializing COM.

LISTING 9.5. THE XAUDIO2 DEMO'S MAIN() SOURCE FILE

```
int main(int args, char* argc[])
{
   cout << "XAudio2 Demo: Playing clip.wav" << endl << endl;
   cout << "Demo will end when the sound is done." << endl <<
endl;

   if(FAILED(CoInitializeEx(NULL, COINIT_MULTITHREADED)))
      return 0;

   IXAudio2* xAudio2Engine = NULL;

   UINT32 flags = 0;
#ifdef _DEBUG
    flags |= XAUDIO2_DEBUG_ENGINE;
#endif

   if(FAILED(XAudio2Create(&xAudio2Engine)))
   {
      cout << "XAudio2 engine was not created!" << endl;
      CoUninitialize();
      return 0;
   }

   IXAudio2MasteringVoice *masterVoice = NULL;

   if(FAILED(xAudio2Engine->CreateMasteringVoice(&masterVoice,
      XAUDIO2_DEFAULT_CHANNELS, XAUDIO2_DEFAULT_SAMPLERATE,
      0, 0, NULL)))
   {
      cout << "Master voice was not created!" << endl;

      if(xAudio2Engine != NULL)
         xAudio2Engine->Release()

      CoUninitialize();
      return 0;
   }

   if(PlayPCM(xAudio2Engine, "clip.wav") == false)
   {
      cout << "clip.wav failed to load!" << endl;
```

```
      if(xAudio2Engine != NULL)
         xAudio2Engine->Release()

      CoUninitialize();
      return 0;
   }

   if(xAudio2Engine != NULL)
      xAudio2Engine->Release()

   CoUninitialize();

   return 1;
}
```

An audio file is loaded and played with a call to `PlayPCM()`. This function is a modified version of the `PlayPCM()` function offered in the Microsoft DirectX SDK sample XAudio2BasicSound. To load and play sounds we will use this function as well as the files SDKwavefile.h and SDKwavefile.cpp. The SDKwavefile files are part of the DirectX Utility (DXUT) library and can be found in any of the DXUT samples in the DirectX SDK. Since these files are part of DirectX, we will use them instead of writing some very long and complicated code for loading audio files. Since the files use DXUT, they have been slightly altered so that the use of the SDKwavefile files does not require any of the other DXUT headers or source files. You can find the modified versions of the SDKwavefile.h and SDKwavefile.cpp files in the XAudio2 folder under Chapter 9 on the CD-ROM.

The `PlayPCM()` function uses the `CWaveFile` class defined in SDKwavefile.h to open the audio file. The file is read by calling the `Read()` function, which takes as parameters a buffer to read into, the size to read in bytes, and an out pointer to the size of bytes read by the function.

Once the file is loaded, the source voice is created. Keep in mind that the source voice represents a stream of audio data. To create the source voice, which has an interface of `IXAudio2SourceVoice`, we call the `CreateSourceVoice()` function of the XAudio2 engine object. This function takes the source voice that will be created, the format of the audio (using the `WAVEFORMATEX` structure provided by Windows), behavior flags, the maximum frequency ratio, a callback interface function, a send list of source voices for the destination of the audio date (optional), and an audio effect chain. The behavior flags can have one of the following values:

- `XAUDIO2_VOICE_NOPITCH` for no pitch control
- `XAUDIO2_VOICE_NOSRC` for no sample rate conversion
- `XAUDIO2_VOICE_USEFILTER` to enable filter effects on the sound
- `XAUDIO2_VOICE_MUSIC` to state that the voice is used to play background music

Once the source voice is created, an audio buffer using the XAudio2 structure `XAUDIO2_BUFFER` is created. This buffer will take the sound data and submit it to the sound voice, which can only happen after a valid sound voice has been created by `CreateSoundVoice()`. The audio buffer has the audio data assigned to the `pAudioData` variable, the audio flags to the `Flags` variable, and the size of the audio to

the AudioBytes variable. The flag of XAUDIO2_END_OF_STREAM tells XAudio2 that there is no more data to follow after the sound has played.

To submit the data to the source voice, you call SubmitSourceBuffer() on the source voice object, which takes as a parameter the XAUDIO_BUFFER object. If all is successful, you can start processing the sound by calling Start() on the source voice. The Start() function takes as parameters behavior flags that must be set to 0 and an operation set. The operation set can be XAUDIO2_COMMIT_NOW to apply the operation immediately or XAUDIO2_COMMIT_ALL to apply all pending operations.

When a source voice is processing, it is being played. You can test the state of the sound by calling the GetState() function on the source voice object. This will return an XAUDIO2_VOICE_STATE object that you can test for various states. To test if the sound is still playing you can test if the BuffersQueued variable is greater than 0.

Once you are done with a source voice, you free it by calling DestroyVoice(). The entire PlayPCM() function is shown in Listing 9.6 with all the code we've just discussed in the previous few paragraphs. This function essentially loads a sound, plays it, and then frees it from memory. As a bonus exercise you should separate the loading and playing code into their own functions and allow the sound to be played multiple times before it is freed.

LISTING 9.6. THE PLAYPCM() FUNCTION

```
bool PlayPCM(IXAudio2* xAudio2Engine, char *filename)
{
   CWaveFile wav;

   if(FAILED(wav.Open(filename, NULL, WAVEFILE_READ)))
      return false;
   WAVEFORMATEX *format = wav.GetFormat();
   unsigned long wavSize = wav.GetSize();
   unsigned char *wavData = new unsigned char[wavSize];

   if(FAILED(wav.Read(wavData, wavSize, &wavSize)))
   {
      if(wavData)
         delete[] wavData;

      return false;
   }


   IXAudio2SourceVoice *srcVoice;

   if(FAILED(xAudio2Engine->CreateSourceVoice(&srcVoice, format,
      0, XAUDIO2_DEFAULT_FREQ_RATIO, NULL, NULL, NULL)))
   {
      if(wavData)
         delete[] wavData;

      return false;
   }
```

```
    XAUDIO2_BUFFER buffer = {0};
    buffer.pAudioData = wavData;
    buffer.Flags = XAUDIO2_END_OF_STREAM;
    buffer.AudioBytes = wavSize;

    if(FAILED(srcVoice->SubmitSourceBuffer(&buffer)))
    {
       srcVoice->DestroyVoice();
       if(wavData)
          delete[] wavData;

       return false;
    }

    HRESULT hr = srcVoice->Start(0, XAUDIO2_COMMIT_NOW);

    bool isRunning = true;

    while(SUCCEEDED(hr) && isRunning)
    {
      XAUDIO2_VOICE_STATE state;

      srcVoice->GetState(&state);
      isRunning = (state.BuffersQueued > 0) != 0;
    }

    srcVoice->DestroyVoice();

    if(wavData)
       delete[] wavData;

    return true;
}
```

SUMMARY

Audio in video games is a very important topic. When it comes to playing sounds using the basics, XACT3 and XAudio2 are fairly straightforward. If you plan on getting advanced with audio engineering, you will need to read more than a chapter because there is a lot that goes into game audio. Even the basic demos in this chapter brought up topics such as sample rate, frequency, streaming, and compression and formats that are key for advanced audio engineers. This chapter served as a very useful introduction to programming with Microsoft's XACT3 and XAudio2 APIs.

The following elements were discussed in this chapter.

- Microsoft audio technologies
- Direct Sound
- XACT3

- XAudio2

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. List the various sound technologies that can be found in the various game development SDKs provided by Microsoft.

2. Describe the XAudio2 API. What does this technology replace?

3. Describe XACT3. What does this technology replace?

4. Describe the audio effects technologies.

5. Describe Direct Sound and its current role as an audio technology.

6. What does XACT stand for?

7. List the audio formats supported by XACT3 and XAudio2.

8. What is endian order, and how does it affect audio files (or files in general) and networking data?

9. What is a wave bank?

10. What is a sound bank?

11. What is a sound cue, and what is it used for in XACT3?

12. What is the main difference between XACT3 and XAudio2 discussed in this chapter?

13. What is the mastering voice, and what is it used for in XAudio2?

14. What is the source voice, and what is it used for in XAudio2?

15. True or false: XAudio2 internally takes care of the endian issue for programmers.

CHAPTER EXERCISES

Exercise 1: Use the DirectX SDK Sample Browser's XACT Tutorial 2: Streaming to add streaming wave bank support to the XACT demo from this chapter.

Exercise 2: Use the DirectX SDK Sample Browser's XACT Tutorial 3: Categories and Variables to add the ability to use the XGS file created by XACT to use categories and variables in the demo you created in Exercise 1.

Exercise 3: Use the XAudio2Sound3D sample from the DirectX SDK Sample Browser to add the ability to play 3D sounds to the XAudio2 demo from this chapter.

10. GAME INPUT

In This Chapter

- Win32 Input
- XInput

Input is a very important topic for video game development. Not only must input be detected from devices so that the user can control their experience, but the input must be detected accurately and quickly to keep the simulation smooth. A vast array of input types can be used in video games. These types of input include but are not limited to the following.

- Keyboards
- Mice
- Game pads
- Steering wheels
- Flight joysticks
- Guitars (e.g., *Guitar Hero/Rock Band* controller)
- Drums (e.g., *Rock Band*)
- Dance pads (e.g., *Dance-Dance Revolution*)
- Motion sensitive devices (e.g., the Wii-mote)
- Microphones for voice recognition

The purpose of this chapter is to briefly review how to detect input on a Windows-based PC to use in video games. The XInput discussion is valid for Xbox 360 input detection as well as for the PC.

WIN32 INPUT

On the PC, numerous devices can be used for input in applications. The most common types of input on PCs are the keyboard and mouse because they are standard when buying or building a PC. When developing games for the PC, developers often focus on these two controller types, while additional controllers such as game pads, steering wheels, and so on

are often optional means of input that the users can use if they own and connect such a device to their machines. In this chapter we will focus on keyboard and mouse input in Win32, as well as game input for users with Xbox-compatible controllers and devices.

There are three main ways to detect input in a Win32 application. You can use the message pump of the application, you can obtain the state of the device using various Win32 functions, or you can use an API such as XInput, DirectInput, and so forth. In this chapter we will discuss each of these very common means of detecting and responding to input. We'll also discuss their differences, advantages, and disadvantages.

DIRECTINPUT

DirectInput is an API that is part of the DirectX SDK that is used to detect input from various devices including the mouse, keyboard, game controllers (e.g., steering wheels, game pads, etc.), and force-feedback devices. DirectInput has traditionally been very beneficial to Windows-based game developers. It has the following benefits that should be taken into consideration.

- DirectInput talks directly with the hardware.
- Any input device can work with DirectInput without knowledge of the specific device being used.
- DirectInput allows for a wide range of devices to be used in an application, each with different features.
- Outside of using the extended features and services of a device (such as force-feedback), it is fairly simple to use the API to detect input from devices, although the process is more involved than it would be for the XInput API.

One of the most important features of a game's input system is speed. DirectInput allows developers to talk directly to the hardware, which allows for fast input-state acquiring. Whenever developers make a game, input, although seemingly minor to the inexperienced, is extremely important to get right. Unresponsive input can really hurt player's experience of a game. Speed is one of the most critical aspects of many different parts of a game, including input.

The major benefit of using DirectInput, aside from its speed, is that DirectInput is an API that allows developers to take advantage of an input device's extended features and services. Many hardware companies make different devices, and some of these devices have features that many other devices of the same type do not. By following the standard, DirectInput can use devices released on the market without any change to the API. In other words, devices released next year will still work with DirectInput if they follow the standard set forth.

The biggest problem with DirectInput is the difficulty accessing these extended features and services. This includes force-feedback, additional buttons (e.g., a five-button mouse), additional keys, and anything added to the device that is not standard among all devices of the same type (e.g., displays on a keyboard). If you are using DirectInput to detect input from a keyboard and mouse without any additions of a specific device, then there is no benefit because, as we will see later in this chapter, you can directly obtain the state of a keyboard and mouse device using two Win32 functions for both the keyboard and mouse. This is a long way from all of the setup necessary for DirectInput to do the same thing, and since speed in detecting input exists in both cases, using DirectInput for standard, everyday keyboards and mice has no benefit.

Another thing to consider when using DirectInput is that, like Direct Sound, it is being depreciated. XInput is the new API that allows for game controller input detection on Windows PCs and the Xbox 360. XInput will be discussed in more detail later in this chapter.

Also, DirectInput has not been updated since DirectX 8.0. It might prove beneficial to use DirectInput to access the features of a device that you cannot access or cannot easily access otherwise, but for everything else it would be beneficial to use the input API. The fact that there is no benefit to using DirectInput in this case is even stated in the DirectX SDK in the DirectInput Introduction section ("The Power of DirectInput"). This fact most likely is one contributing factor to the API's deprecation.

WINDOWS MESSAGE PUMP

In Win32 applications, the programmer can use the message pump to obtain messages from the window or from the Windows operating system. These messages include input events such as button presses on a keyboard, button clicks on a mouse, and so forth.

Win32 applications have a feature known as the application loop. This loop runs endlessly until some condition is met to cause the loop to break and the application to quit. What this condition is depends on the specific application and the programmers who have developed it. This condition could be as simple as the user pressing the Esc key, clicking a menu item such as File or Exit, clicking on the Close button, and so on.

In the application loop, as you should already know, a callback procedure is called every time an event is passed to the application's message pump. The callback is a function that usually executes specific code depending on the event that is passed to it, and it is only called by the application, not by the user.

An example of the callback procedure used for most of the demos in this book can be seen in the following:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT m, WPARAM wp, LPARAM
lp)
{
    // Window width and height.
    int width, height;

    switch(m)
    {
        case WM_CLOSE:
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
            break;

        case WM_SIZE:
            height = HIWORD(lp);
            width = LOWORD(lp);
            if(height == 0)
                height = 1;

            ResizeD3D10Window(width, height);
            return 0;
            break;
```

```
        case WM_KEYDOWN:
            switch(wp)
            {

                case VK_ESCAPE:
                    PostQuitMessage(0);
                    break;

                default:
                    break;
            }
            break;

        default:
            break;
    }

    // Pass remaining messages to default handler.
    return (DefWindowProc(hwnd, m, wp, lp));
}
```

In the callback procedure for most of the demos in this book, the events checked for are the close (`WM_CLOSE`), destroy (`WM_DESTROY`), resize (`WM_SIZE`), and keyboard button presses (`WM_KEYDOWN`) events. As far as input is concerned, you can check for keys being pressed by checking if the event is a `WM_KEYDOWN` (or `WM_KEYUP` for released buttons) event. The `WPARAM` parameter for the callback procedure stores the value of the event, so for button presses you can check this variable for specific key presses and respond to them accordingly. In the demos the callback procedure checks for the Esc key (`VK_ESCAPE`), and when this key is pressed, the demos quit their execution.

There are 256 keys on the standard keyboard, and a list of all of the virtual key codes for each button can be found in the MSDN documentation at http://msdn.microsoft.com/en-us/default.aspx. The constants are also defined in windows.h.

Events are passed by the operating system to the application. This is a problem in video games because the time it takes for the OS to notify the application of an event can have a serious impact on input detection and response time. In addition, there is no guarantee of how long it will take the OS to notify the application of an event, so input might not be as fast as real-time applications require. Therefore, it is not recommended that you use the callback procedure for input in your video game applications.

OBTAINING KEY STATES IN WIN32

You can obtain the states of buttons and keys on the keyboard and mouse by using various Win32 functions that will be discussed briefly in this section. You can use DirectInput, but this requires more work to set up, which might not be necessary if you just want standard button and key presses.

The first function we will examine is `GetAsyncKeyState()`. This function takes as a parameter the virtual key code of the keyboard key or mouse button that you want to test for being pressed. This function tests if the key or button is down at the time the function was called and tests if the key was pressed during a previous call to the function. The value returned by this function is a short integer. If the most significant bit is set, then the key is

down; if the least significant bit is set, then the key is up. The function prototype for the `GetAsyncKeyState()` function is as follows.

```
SHORT GetAsyncKeyState(int vKey);
```

An example of testing if the up arrow key is pressed by testing the most significant bit is as follows.

```
if(GetAsyncKeyState(VK_UP) & 0×80) /*  */
```

An alternative to the `GetAsyncKeyState()` function is the `GetKeyState()` function, which does the same thing, with the exception that the `GetKeyState()` function returns the key or button information and status that does not reflect the interrupt-level state associated with the hardware.

Another function that can be used to get the state of keyboard keys is the `GetKeyboardState()` function. This function takes an address as a parameter to a 256-element array that will store the state of all keyboard keys. If the function succeeds in the gathering this keyboard information, the `GetKeyboardState()` function returns `true`; if not, it returns `false`. The function prototype for this function is as follows.

```
BOOL GetKeyboardState(PBYTE lpKeyState);
```

For the mouse, it is possible to obtain the mouse's position by calling the `GetCursorPos()` function, which takes as a parameter the address to the `POINT` object that will store the X and Y position of the cursor and returns `true` or `false` for whether or not the function succeeded. The function prototype for the `GetCursorPos()` function is as follows.

```
BOOL GetCursorPos(LPPOINT lpPoint);
```

The `GetKeyState()` *and* `GetAsyncKeyState()` *functions can be used to return the key state of keyboard keys and mouse buttons.*

XINPUT

XInput is an API that is now part of the DirectX and XNA SDKs and allows Windows and Xbox 360-based applications to detect input from Xbox 360 controllers. These controllers include any device available now or in the future that can be used on the Xbox 360. Such devices are game pads, guitars, big-button controllers, drums, and so forth. XInput also supports controller vibrations (force-feedback) and voice input and output using the Xbox 360 headset.

All Xbox 360 controllers are compatible with Windows XP and Vista and are USB devices. These devices can be used like traditional game controllers (such as in DirectInput), or they can be used using XInput, which is the recommended API for these controllers.

XInput is the replacement for DirectInput when it comes to game controllers. XInput has several benefits over DirectInput, including the following.

- XInput is easier to use.
- XInput is faster to set up and detect input from.
- XInput can be used on both Windows PCs and Xbox 360 consoles.
- The vibration functionality of Xbox controllers can only be set using XInput.
- Future Xbox controllers that are released will work with the API.

No real setup code is necessary for getting XInput running in a gaming application. The only requirement is that the USB controller is plugged into the machine and a call to `XInputGetState()` is made. (The `XInputGetState()` function will be discussed in the next section.) You can, however, enable XInput's reporting state by calling the function `XInputEnable()`. By default, XInput's reporting state is set to `true`, which means that calls to `XInputGetState()` will return the state of the device. If the reporting state is set to `false`, only natural data is sent. In other words, XInput will ignore the state of the device, which can be useful, for example, when the window is minimized but the game is still technically running. The `XInputEnable()` function is as follows.

```
void XInputEnable(BOOL enable);
```

You can use DirectInput to detect input from Xbox controllers, but DirectInput will not be able to access the vibration functionality. It will treat the left and right triggers as a single button instead of separately, and you cannot access audio from the Xbox 360 headset using DirectInput.

SETTING UP XBOX 360 CONTROLLERS

All of the devices have buttons and so on. You can think of, for example, the guitar controller as a game pad in the shape of a guitar without the left and right analog sticks or the left and right triggers. A game pad has four face buttons, two triggers, two bumper buttons (buttons in front of the triggers), two analog sticks, each with a digital button (accessed by pushing the stick inward), a guide button, a directional pad, and start and back buttons.

The XInput function `XInputGetState()` is used to get the state of an Xbox controller. The function takes as parameters the index of the player and the address of an `XINPUT_STATE` object that stores the state information. The player index can be between 0 and 3 for players one through four. The return value for the `XInputGetState()` function can be either `ERROR_SUCCESS` if a controller is connected to the machine at that player index or `ERROR_FAIL` if no device is connected. The function prototype for the `XInputGetState()` function is as follows.

```
DWORD XInputGetState(DWORD dwUserIndex, XINPUT_STATE* pState);
```

The XINPUT_STATE structure has fields that store the state of the controller's various buttons, sticks, and so on. The information in this structure is the state of the device when the XInputGetState() function was called. The structure has the following definition, where wButtons is a flag that can store all information of the buttons, which includes face buttons, bumpers, and the start and back buttons.

```
typedef struct _XINPUT_GAMEPAD {
    WORD wButtons;
    BYTE bLeftTrigger;
    BYTE bRightTrigger;
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD, *PXINPUT_GAMEPAD;
```

XInput has flags that correspond to the various input states of an Xbox controller. These flags are as follows.

- XINPUT_GAMEPAD_DPAD_UP
- XINPUT_GAMEPAD_DPAD_DOWN
- XINPUT_GAMEPAD_DPAD_LEFT
- XINPUT_GAMEPAD_DPAD_RIGHT
- XINPUT_GAMEPAD_START
- XINPUT_GAMEPAD_BACK
- XINPUT_GAMEPAD_LEFT_THUMB
- XINPUT_GAMEPAD_RIGHT_THUMB
- XINPUT_GAMEPAD_LEFT_SHOULDER
- XINPUT_GAMEPAD_RIGHT_SHOULDER
- XINPUT_GAMEPAD_A
- XINPUT_GAMEPAD_B
- XINPUT_GAMEPAD_X
- XINPUT_GAMEPAD_Y

DETECTING BUTTON PRESSES

Once you have the controller's state, you can test the individual buttons to see if they are being pressed or not. The state for each button is stored in the XINPUT_STATE object's GamePad.wButtons field. The wButtons field is a short integer whose bits represent

each button. You can use the button flags mentioned in the previous section and the logical AND operator (&) to test if a bit is set. If a bit is set, this means the button is being pressed. An example of obtaining the current input state and testing if the left and right bumper buttons are being pressed follows.

```
XINPUT_STATE state;
XInputGetState(0, &state);

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_LEFT_SHOULDER)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_RIGHT_SHOULDER)
   // Do Something…
```

If the conditional statements equal true, the button is currently being pressed. You can test if the back, start, and face buttons are being pressed by using their corresponding button flags as shown in the following.

```
// Back and start.
if(state.Gamepad.wButtons & XINPUT_GAMEPAD_BACK)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_START)
   // Do Something…

// Face buttons (a, b, x, y).
if(state.Gamepad.wButtons & XINPUT_GAMEPAD_A)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_B)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_X)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_Y)
   // Do Something…
```

The game pad controllers along with a few other controller types (e.g., the guitar) also have a directional pad on them. These directional pads are considered buttons, where you can press the up, down, left, and right arrow buttons independently or at the same time. An example of testing the controller state for these button presses is shown in the following.

```
// Arrow pad.
if(state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_UP)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_DOWN)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_LEFT)
   // Do Something…
```

```
if(state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_RIGHT)
   // Do Something…
```

The last remaining buttons are the left and right thumb stick's digital buttons. These buttons are pressed by pushing the thumb stick in. You can test if these thumb stick digital buttons are pressed independently of testing for thumb stick movement. An example of testing the thumb stick digital buttons is as follows.

```
// Thumb buttons (pushing in left and right joysticks).
if(state.Gamepad.wButtons & XINPUT_GAMEPAD_LEFT_THUMB)
   // Do Something…

if(state.Gamepad.wButtons & XINPUT_GAMEPAD_RIGHT_THUMB)
   // Do Something…
```

DETECTING TRIGGERS

Triggers are not buttons in the traditional sense. A trigger is more of a pressure-sensitive button, and XInput allows you to detect just how much pressure is being applied to these triggers. Currently, there are two triggers on an Xbox 360 controller on the top of the game pad device.

To test if a trigger is being pressed, you test the `XINPUT_STATE` object's `Gamepad.bLeftTrigger` for the left trigger and `Gamepad.bRightTrigger` for the right trigger. These values are represented by unsigned `char` variables and if their value is over 0, the trigger is being pressed. If these trigger values are 255, the trigger is being held all the way down, while anything in between 0 (0%) and 255 (100%) determines the amount of pressure being applied to the trigger. The trigger values are obtained as follows.

```
unsigned char lt = state.Gamepad.bLeftTrigger;
unsigned char rt = state.Gamepad.bRightTrigger;
```

DETECTING THUMB STICK MOVEMENTS

The Xbox 360 game pad has two thumb sticks. The X and Y position information of the left thumb stick is stored in `Gamepad.sThumbLX` and `Gamepad.sThumbLY`, while the right thumb stick information is stored in `Gamepad.sThumbRX` and `Gamepad.sThumbRY`.

If the X and Y value of a thumb stick is equal to 0, the stick is centered along that axis. Therefore, an X and Y value of 0 for each means the thumb stick has not moved. The minimum value the thumb stick's axes can have is −32768, and its maximum value is 32768. This means that if you are looking at the X axis, a value of −32768 indicates that the thumb stick is being moved all the way to the left, while a value of 32768 indicates that the stick is being moved all the way to the right. For the Y axis, a value of −32768 means the stick is being moved all the way down, while a value of 32768 means the stick is being moved all the way up. Keep in mind that a value of 0 (or near 0 since the sticks are highly sensitive) means the stick is not being moved at all. An example of reading the thumb stick information is shown as follows, where the values themselves are represented by short integers.

```
short lx = state.Gamepad.sThumbLX;
```

```
short ly = state.Gamepad.sThumbLY;
short rx = state.Gamepad.sThumbRX;
short ry = state.Gamepad.sThumbRY;
```

Not all joysticks will read zero when centered, especially for game pads that have been used frequently over time. One way of accounting for this is to use what is known as a dead zone. A dead zone is just a range of values that you will ignore and consider zero. For example, if the joystick reads a value between –20 and 20 then you can clamp any value between that range of dead zone values to 0. The same can be done to account for micro joystick changes from the center where the user might not be moving the thumb stick intentionally but movement is being registered.

CONTROLLER VIBRATIONS

The Xbox controller can have two vibration modes. Some controllers, like the guitar, do not have vibration support, so setting this would not result in anything happening when using some non-game pad controllers. Each game pad controller has a left motor and a right motor. The differences between the motors are that the left motor is for low frequency vibrations (such as footsteps) while the right motor is stronger and used for higher frequency vibrations (such as explosions).

Setting the controller's vibration is easy. You first create an `XINPUT_VIBRATION` object and set its left and right motor speeds from 0 to 65,535. If 0 is used, the controller does not vibrate. If 65,535 is used, the controller vibrates at 100% of its maximum power. Anything within that range will be a percentage between no vibration and maximum power. The `XINPUT_VIBRATION` structure is as follows.

```
typedef struct _XINPUT_VIBRATION {
    WORD wLeftMotorSpeed;
    WORD wRightMotorSpeed;
} XINPUT_VIBRATION, *PXINPUT_VIBRATION;
```

Once you've created an `XINPUT_VIBRATION` object and specified the left and right motor speeds, you are ready to apply it to any controller attached to the machine. To set the vibration, you can use `XInputSetState()`, and you pass to it the index of the controller to set the vibration state and the address to the vibration state object. This sets the motor speed for the specified controller. If you want to turn the vibration off, you have to call this function again and set both the left and right motors to 0. The function prototype for the `XInputSetState()` function is as follows.

```
DWORD XInputSetState(DWORD dwUserIndex, XINPUT_VIBRATION*
pVibration);
```

CONTROLLER CAPABILITIES

You can obtain the capabilities of the controller (i.e., features) by calling the `XInputGetCapabilities()` function. This function takes the player index, input flags

that can only be XINPUT_FLAG_GAMEPAD at this time, and the output address to the XINPUT_CAPABILITIES object that will store the features of the controller. The function prototype for the XInputGetCapabilities() function is as follows.

```
DWORD XInputGetCapabilities(DWORD dwUserIndex, DWORD dwFlags,
XINPUT_CAPABILITIES* pCapabilities);
```

The XINPUT_CAPABILITIES structure is as follows.

```
typedef struct _XINPUT_CAPABILITIES {
    BYTE Type;
    BYTE SubType;
    WORD Flags;
    XINPUT_GAMEPAD Gamepad;
    XINPUT_VIBRATION Vibration;
} XINPUT_CAPABILITIES, *PXINPUT_CAPABILITIES;
```

The Type can only be XINPUT_DEVTYPE_GAMEPAD.

The SubType can be XINPUT_DEVSUBTYPE_ARCADE_STICK if it is an arcade stick controller, XINPUT_DEVSUBTYPE_GAMEPAD if it is a game pad controller, or XINPUT_DEVSUBTYPE_WHEEL if it is a steering wheel. Other controllers such as guitars and so on fall under the game pad type.

The Flags can only be XINPUT_CAPS_VOICE_SUPPORTED. The XINPUT_GAMEPAD stores the controller button/thumb stick and other states for the device, while the XINPUT_VIBRATION stores the current vibration state for the device.

SUMMARY

Input in video games is very important to the overall experience. On Windows systems and the Xbox 360 a few APIs can be used to quickly and effectively detect and respond to input from a large array of devices. On Windows you can also use Win32 functions to obtain button and key states from keyboards and mice. Once an application has an efficient means of detecting input, the next challenge is how the application responds to that input, which is game specific.

The following elements were discussed in this chapter.

- Game input in general
- Keyboards in Win32
- Mice in Win32
- DirectInput
- XInput

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What is DirectInput?

2. What advantage does DirectInput offer when you are using standard keyboards and mice when compared to Win32 functions as discussed in this chapter?

3. What is XInput?

4. What are the benefits to using XInput over DirectInput as described in this chapter?

5. What are the steps necessary to set up XInput in code? How is XInput enabled and disabled?

6. What XInput function is used to obtain the state of a controller device?

7. What field is used to detect button presses on an Xbox controller? How do the button flags factor in when determining if a button is pressed?

8. What fields are used to store the thumb stick locations of an Xbox controller?

9. What are the minimum and maximum pressure values a trigger can be on an Xbox controller?

10. List the steps to setting the motor speed in an Xbox controller.

11. What is the minimum and maximum power the motors can move at in an Xbox controller?

12. What XInput function is used to obtain the controller device's capabilities?

13. List the three Xbox controller subtypes that were discussed in regard to obtaining device capabilities.

CHAPTER EXERCISES

Exercise 1: Create a demo that allows you to move a 3D box from side to side and up and down using the arrow keys of the keyboard.

Exercise 2: Create a demo that allows you to move a 3D box from side to side and up and down using the directional pad of an Xbox controller.

Exercise 3: Create a demo that allows you to move a 3D box from side to side and up and down using the left joystick of an Xbox controller.

11. 3D MODELS

In This Chapter

- Overview of 3D Models
- Files in C++
- Token Stream
- OBJ Models

3D models are what make up 3D scenes in modern video games. 3D objects, characters, and environments are carefully sculpted polygons placed within an environment to represent whatever is going on. In today's games these polygons are often triangles, and they often have lots of texture data that is used to simulate a large degree of detail and quality. 3D geometry is usually created expensively with off-the-self tools such as Softimage XSI, Autodesk 3D Studio Max, and so forth, or the geometry is created using a tool built in-house or algorithmically.

In this chapter we will look at how to load 3D geometry from a file. Direct3D 9.0 and previous versions supported loading .X models using the DirectX Utility library. In Direct3D 10 there is no .X loading support, so we have to write code to load our models ourselves. By the end of this chapter you will have all the tools necessary to load any geometry file format you wish. The only requirement is that you understand the format of the file you want to load. With that knowledge you can load and use any file.

OVERVIEW OF 3D MODELS

Some games have their own file formats that are used for storing 3D information. This allows the developers to create a format that has all of the information needed by a particular gaming application. What this information is may vary from game to game, so it really depends on the application.

For example, if you have models in your game that require the position, texture coordinates, normals, and material information, you can create a file format that suits your needs by specifying this information. The game can then load this data and render the geometry with its material in real time.

TOOLS USED FOR CREATING 3D GEOMETRY

Another option is to utilize an already existing file format. This is convenient because any tools available on the market that save information to the format of your choice can be used to create game assets. If you use an existing file format, it might not suit your needs, or it might specify more information than you need for your assets. This is often the case with formats saved by general-purpose tools, where a lot of information can appear in the file that you would not need in an actual game. Some of the most popular applications used to create 3D geometry include the following.

- 3D Studio Max
- Lightwave
- XSI
- Maya
- ZBrush
- Truespace

Along with using existing tools and formats, you can also write your own file exporters and converters to change a file to a format that your game is ready to use. This is a very common practice in video games, as it allows developers to use powerful and complex tools such as 3D Studio Max and save the content created in a format the game is able to read and use.

FILES IN C++

In this section we will briefly review how C++ loads file data. C++ uses the standard `ifstream` and `ofstream` classes. The `ifstream` class represents an input file stream, and `ofstream` represents an output file stream. Both classes derive from the base class, `ifstream`, and are part of the C++ standard.

In this section we will take a quick look at loading and saving files in C++ as a brief refresher since the code samples in this chapter will depend on the ability to load data from files using the `ifstream` class.

INPUT AND OUTPUT STREAMS

On the CD-ROM, in the Chapter 11 folder, is a demo called Files that demonstrates how to save data to a file and how to read it back using the `ifstream` and `ofstream` classes. If you are already familiar with the file stream classes you can skip to the next section.

The data is loaded into the demo and displayed within the function `LoadFileData()`. This function creates an `ifstream` object, opens the file, checks if the file actually opened, determines the size of the file, and then reads the file's data into memory. Once loaded, the text that was loaded from the file is displayed, the allocated memory that was used to store the file's data is deleted, and the function returns.

To open a file using `ifstream` or `ofstream`, you call the `open()` function. This function takes as parameters the file name and the file mode. The file mode can be one of the following flags:

- `app:` This flag tells the stream object to open a file and append new information to it.
- `ate:` This flag sets the file pointer to the end of the file stream upon opening it.
- `binary:` This flag indicates that the file being opened is considered a binary file and not a text file.

- `in:` This flag is used to specify that the file is being used for reading rather than writing.

- `out:` This flag specifies that the file is being used for writing rather than reading.

- `trunc:` This flag discards data upon opening.

To check if the file successfully opened, you can use the function `is_open()`, which returns true if the file stream is open or false if it is not.

In the Files demo the next step calculates the file size. This can be done by setting the file pointer to the end of the file stream by calling the function `seekg()`, calling `tellg()` to get the number of bytes at that position, and then calling `seekg()` again to return us to the beginning of the file stream so that we are ready to read the information from the beginning. The `seekg()` function takes as parameters the file position to set, an offset from that position to set, and a seeking direction. In this demo we are using one of the overloaded versions that accepts an offset and a seek direction. The seek direction can be either `beg`, which stands for the beginning of the stream, `end`, which is the end of the stream, or `cur`, which is the current position in the stream. The function `tellg()` gets the file position, which can be used to represent the byte position at the current location. So if we seek to the end of the file, a call to `tellg()` will tell us the total bytes in the file. We seek back to the start so we can begin reading since reading occurs where the file pointer is located.

With the size of the file, we can allocate a buffer to hold that information, and then we can read it with a call to `read()`. The `read()` function takes as parameters a pointer to a buffer to read the data into and the amount you want to read. To read the entire file we use the file's size from the beginning of the stream in this demo. Once the information is read, the file is closed, the allocated memory is deleted, and the function returns.

Other functions that are part of the `ifstream` class include the following that are inherited from the `istream` class.

- `gcount():` This function returns the number of characters extracted by the last input operation.

- `get():` This function is used to read unformatted data from the input file stream.

- `getline():` This function is used to read data from the input stream into an array.

- `ignore():` This function is used to read data from the stream and then discard it.

- `peek():` This function returns the next character in the stream but does not extract it (i.e., doesn't move the file pointer).

- `readsome():` This function reads data up to the size of the array even if the end of the file or the number of bytes to read has yet to be reached.

- `putback():` This function decrements the file pointer back one and makes the character passed to its parameter the next character to be read from the stream.

- `unget():` This function decrements the file pointer back one.

- `sync():` This function synchronizes the input buffer with a source of characters.

- `sentry()`: This function performs exception-safe prefix and suffix operations on the stream.

The `ofstream` class has many of the same functions minus the ones dealing with input. The `ofstream` class also has a function called `write()` that is used to write data to a file. The `write()` function takes as parameters the buffer to write and the amount of bytes to write. The `ofstream` class also has a function called `flush()`, which is used to force the object to write out all unwritten data as soon as possible.

The main.cpp source file for the Files demo is shown in Listing 11.1. The demo's `main()` function saves data to a file by calling the demo's `SaveFileData()` function, and then it reads it back by calling the demo's `LoadFileData()` function. The data that has been read is displayed inside `LoadFileData()`.

LISTING 11.1. THE MAIN.CPP SOURCE FILE FOR THE FILES DEMO

```
/*
   Files in C++
   Ultimate Game Programming with DirectX 2nd Edition
   Created by Allen Sherrod
*/

#include<iostream>
#include<fstream>

using namespace std;


bool LoadFileData()
{
   ifstream fileStream;
   int fileSize = 0;

   // Open file then test that it actually opened.
   fileStream.open("test.txt", ifstream::in);

   if(fileStream.is_open() == false)
      return false;
   // Get file size.
   fileStream.seekg(0, ios::end);
   fileSize = fileStream.tellg();
   fileStream.seekg(0, ios::beg);

   if(fileSize <= 0)
      return false;

   // Allocate memory for text data.
   char *buffer = new char[fileSize];
   memset(buffer, 0, fileSize);

   if(buffer == NULL)
      return false;
```

```
   // Read data and close the file.
   fileStream.read(buffer, fileSize);
   fileStream.close();

   buffer[fileSize - 1] = '\0';

   // Display the data.
   cout << "test.txt (" << fileSize << " bytes) contents:" <<
endl;
   cout << buffer << endl;

   delete[] buffer;

   return true;
}

bool SaveFileData()
{
   ofstream fileStream;

   // Open file then test that it actually opened.
   fileStream.open("test.txt", ofstream::out);
   if(fileStream.is_open() == false)
      return false;

   char buffer[] = { "This is saved out to the file!" };

   // Write information to the file then close the file.
   fileStream.write(buffer, sizeof(buffer));
   fileStream.close();

   return true;
}

int main(int args, char *argc[])
{
   cout << "Loading files example in C++." << endl << endl;

   // Try to save the file.
   if(!SaveFileData())
   {
      cout << "Could not save file!" << endl << endl;
   }

   // Try to load the file.
   if(!LoadFileData())
   {
      cout << "Could not read file!" << endl << endl;
   }

   cout << "Press enter to quit." << endl;
   char c;
```

```
    cin >> c;

    return 1;
}
```

BINARY FILES AND BYTE ORDERING

Loading text files consists of loading data in a stream of characters, where a character is a single byte. This assumes the file has ASCII text, which essentially means there are no values that take up more than a byte of space. Binary files, on the other hand, or any file that assumes more than one byte per value, can have multibyte values saved to the file. Therefore, if an integer variable is 4 bytes in C++, you can save the entire variable to a file. To read it you would read the 4 bytes that make up the integer.

The problem with multibyte values is in their byte ordering. Different hardware works on different byte ordering. For example, big endian is the byte ordering used by PowerPC processors, while little endian is used by processors. When saving information to a file, the byte ordering from one piece of hardware is not translated automatically to another. This means that if you try to load a value that is in big endian on a little endian machine, the value will not be what you expect.

The solution to this problem is fairly straightforward. To start, you have to be aware of what byte ordering the machine uses and what order the file was saved in. If the two orders are different, you can simply swap the bytes that make up the variable when you read it. This can be done by simply casting the variable to a character pointer and swapping the bytes using array indexes just like you would if you had an array of four characters. When swapping, the fourth byte becomes the first, the third becomes the second, the second becomes the third, and the first byte becomes the last.

In this book we will not be loading multibyte values, and in this chapter the file format we will load, in which our 3D geometry is defined, is a text file. However, when reading and writing multibyte values across different platforms, even if they are being transmitted over a network such as in a multiplayer game, you have to take byte ordering into consideration.

TOKEN STREAM

In a text file exported from a 3D modeling application, the data is usually presented in a human-readable form. In this type of file there can be line breaks, spaces, tabs, words, numbers, curly braces and other symbols, and so on. Regardless of what information is in the file, it is useful to be able to effectively read the information you care about. In most binary file formats the data is tightly packed, with only the data necessary to represent its purpose. In a text file there can exist comments, new lines, spaces, and so on that are at times more for the benefit of someone reading the text file's contents than for the application loading it.

In this section we will cover the creation of a class that will take a series of text and split it up into separate pieces of text. In other words, we will create a list of all words, numbers, and symbols that appear in the text file, without any delimiters. Delimiters are things such as new lines, spaces, tabs, and any other characters that can appear in the file that mark the end of a piece of text (such as a word). Consider, for example, the following.

```
"She sells sea shells by the sea shore"
```

This text is made up of eight individual pieces of text. These pieces are known as tokens, pieces of text (letters, numbers, and symbols) that are separated by delimiters. A delimiter can be anything you define it to be, but they are commonly the examples mentioned earlier; spaces, end-of-file markers, new lines, and so on. In the example above the only delimiters are the white spaces between each word.

For the class we will create, we want the ability to extract each token between delimiters. Using the example from earlier, we want a class that we can use to call a function to get each word from the text, one at a time. To do this we will define a function that can test each character of the complete text to see if it is a delimiter and not part of a token. While reading, we read each character until we come to such a delimiter, and then we return that token, which we'll call `GetNextToken()` in the class. Every time `GetNextToken()` is called, a new token from the file will be returned. Consider the following example.

```
"VertexPos 100 50 30"
```

If we called get `GetNextToken()` for the first time on the above example, the token `VertexPos` will be returned. If we were looking for the next vertex position, we would know that the next three calls to `GetNextToken()` would return the X, Y, and Z values, which would be 100, 50, and 30. If we defined a 3D model this way, we could have each vertex position of each triangle on a line in the text file, and we simply would call one function, `GetNextToken()`, to extract each piece of information, one at a time.

This class will be called `TokenStream`, and it will have a function to load a file's data into memory or to set the data using an array, to get the next token, and to move to the next line in the file. A file will be loaded by calling `LoadTokenStream()`, which will only need to take as a parameter the name of the file being loaded. Another way to set the data stream is to manually set it by calling `SetTokenStream()`, which will take a character pointer to an array of text to set. That way you can set the data stream from a file or from an array of characters already in the application.

There will also be two `GetNextToken()` functions, where the first will return the next token that appears in the file while the overloaded version will search for a specific token and return the token that immediately follows. The `MoveToNextLine()` function for moving to a new line in the text data will read characters until a new-line character is found and will return the entire line to the caller. This can be useful if you have data specified strictly line by line such as the "`VertexPos 100 50 30`" example from earlier. If you read the entire line, you could use another `TokenStream` object to break that line down into individual tokens for further processing.

Another function that is part of the class includes a function to reset the token stream, which means moving the reading indexes that are used to read tokens to the beginning of the file (i.e., set to 0). There is also a function pointer that is used to allow the programmer to set a function that is used to test characters for a delimiter by testing whether what they consider a valid token character is being read. Since a delimiter can be anything you define it to be, this might be useful for reading different types of files, where what you consider a delimiter might change depending on the file being read.

There is also a function, called `DefaultIsValidIdentifier()`, that is not part of the class. This function will be set to the class's function pointer by default and will essentially consider white spaces, new lines, end-of-file markers, tabs, and any other non-letter, -number, or -symbol as a delimiter. That way anyone using this class can use the default function instead of always having to write their own to do the same thing.

The class declaration for `TokenStream` is shown in Listing 11.2. The class has member variables for the start and ending indexes that are used internally for the reading of tokens (more on this coming up) and a string that holds the entire text data that was set to the token stream.

LISTING 11.2. THE `TOKENSTREAM` CLASS DECLARATION

```
/*
   Token Stream
   Ultimate Game Programming with DirectX 2nd Edition
   Created by Allen Sherrod
*/
   #ifndef _TOKEN_STREAM_H_
   #define _TOKEN_STREAM_H_


   bool DefaultIsValidIdentifier(char c);


   class TokenStream
   {
      public:
         TokenStream(bool (*IdentiferFuncPtr)(char c));
         ~TokenStream();

         void ResetStream();

         bool LoadTokenStream(char *fileName);
         void SetTokenStream(char *data);
         bool GetNextToken(std::string *buffer);
         bool GetNextToken(std::string *token, std::string
*buffer);

         bool MoveToNextLine(std::string *buffer);

      private:
         int m_startIndex, m_endIndex;
         std::string m_data;

         bool (*isValidIdentifier)(char c);
    };

    #endif
```

In the class the constructor sets the read indexes to 0 and sets the function pointer. If `NULL` is passed to the constructor, `DefaultIsValidIdentifier()`, which is the default function, is used. `DefaultIsValidIdentifier()` is a simple function that considers everything in the ASCII code range between 32 (the `!` symbol) and 127 (the `~` symbol) as a valid part of a token. This means anything outside that range such as white spaces is considered a delimiter. Therefore, if the character passed as the parameter to this function is a valid token character, the function will return `true`; otherwise, it returns `false` if it considers the character to be a delimiter. The code functions for the `DefaultIsValidIdentifier()` function, class constructor, and class destructor are

shown in Listing 11.3 along with the `ResetStream()` function, which just sets the two indexes to a value of 0.

LISTING 11.3. THE CLASS CONSTRUCTOR, DESTRUCTOR, STREAM RESET, AND VALID TOKEN CHECK

```
/*
    Token Stream
    Ultimate Game Programming with DirectX 2nd Edition
    Created by Allen Sherrod
*/

#include<string>
#include<fstream>
#include"TokenStream.h"

using namespace std;


bool DefaultIsValidIdentifier(char c)
{
   // ASCII from ! to ~.
   if((int)c > 32 && (int)c < 127)
      return true;

   return false;
}
TokenStream::TokenStream(bool (*IdentiferFuncPtr)(char c))
{
   ResetStream();

   if(IdentiferFuncPtr == NULL)
      isValidIdentifier = DefaultIsValidIdentifier;
   else
      isValidIdentifier = IdentiferFuncPtr;
}


TokenStream::~TokenStream()
{

}


void TokenStream::ResetStream()
{
   m_startIndex = m_endIndex = 0;
}
```

The next functions we will be looking at, which are shown in Listing 11.4, are `SetTokenStream()` and `LoadTokenStream()`. The `SetTokenStream()` function resets the stream indexes and sets the text data to the function's parameter. The `LoadTokenStream()` function opens a file, reads its contents, sets the file's contents to

the class's data string, deletes the temporary allocated memory that was used to read from the file, closes the file, and returns. This code is essentially the same as that in the Files demo but is now being used as the `TokenStream` class's loading function.

LISTING 11.4. THE `SETTOKENSTREAM()` AND `LOADTOKENSTREAM()` FUNCTIONS

```
void TokenStream::SetTokenStream(char *data)
{
   ResetStream();
   m_data = data;
}


bool TokenStream::LoadTokenStream(char *fileName)
{
   ResetStream();

   ifstream fileStream;
   int fileSize = 0;

   fileStream.open(fileName, ifstream::in);

   if(fileStream.is_open() == false)
      return false;

   fileStream.seekg(0, ios::end);
   fileSize = fileStream.tellg();
   fileStream.seekg(0, ios::beg);

   if(fileSize <= 0)
     return false;

   char *buffer = new char[fileSize];
   memset(buffer, 0, fileSize);

   if(buffer == NULL)
     return false;

   fileStream.read(buffer, fileSize);
   buffer[fileSize - 1] = '\0';

   fileStream.close();
   m_data = buffer;
   delete[] buffer;

   return true;
}
```

The `GetNextToken()` functions are not difficult, but they are where all the work occurs when you use the class. The first of these two functions starts off by setting the starting index to the last position of the ending index, and it goes on to test that we have not reached the end of the text data. When this function is first called, both the start and end

are 0, but as reading occurs, the starting index is set to wherever the function last left off, which is at the ending index position.

Assuming there is information to parse, the function then reads all characters until it reaches a valid token character. For every character that is a delimiter, the start index is moved forward. This allows the code to skip all delimiters until it reaches the start of the next token. Therefore, if the text had a bunch of white spaces before the data begins, let's say for formatting purposes in the original text file, those delimiters are skipped so the function can find the start of the next token. Once the start is found, the new end index will be one past the new starting index.

With the starting location of the next valid token found, the next step is to reach the entire text that makes up that token. This involves reading characters until a delimiter is found. Each time the code reads a valid token identifier, the end index is incremented. Once a delimiter is found, the text between the start index and end index represents the token. So if you were reading the following line:

```
"    This       is a line"
```

the start index will be 4 since the first three white spaces are skipped and the "T" is the fourth character, and the end index is 7, which is the position of the first "s." The next time `GetNextToken()` is called, using the text above, the white spaces between "This" and "is" are skipped, and the starting index is set to the "i" in "is," while the ending index is, after the function completes, set to the "s" in "is." This would continue until the `TokenStream` object reaches the end of the data stream. If it was at the end of the data stream, the function would continue to return `false` during future calls unless the indexes are reset by calling `ResetStream()`.

The last part after the code identifies the start and end indexes that make up the token is to return the token's text. This is done by setting the function's parameter, which is a pointer to where the token is to be saved, to the characters between the start and end indexes that make up the token. If `NULL` is passed to the function, the token is discarded, which can be useful if you wanted to move past the next token without actually storing it because you want to discard or ignore it. As long as the function is able to find a token, it returns `true`; otherwise, it will return `false`. The first `GetNextToken()` function is shown in Listing 11.5.

LISTING 11.5. THE FIRST `GETNEXTTOKEN()`

```
bool TokenStream::GetNextToken(std::string *buffer)
{
   m_startIndex = m_endIndex;
   int length = (int)m_data.length();

   // Make sure we are not at the end.
   if(m_startIndex >= length)
      return false;

   // Skip all delimiters.
   while(m_startIndex < length &&
         isValidIdentifier(m_data[m_startIndex]) == false)
   {
      m_startIndex++;
   }
```

```
   // The end is one past where we are starting (for 1
character).
   m_endIndex = m_startIndex + 1;

   // If we haven't reached the end of the data stream to begin.
   if(m_startIndex < length)
   {
      // Read until we reach  a delimiter or the end.
      while(m_endIndex < length &&
            (isValidIdentifier(m_data[m_endIndex])))
      {
         m_endIndex++;
      }
      // If we are returning this token, save it.
      if(buffer != NULL)
      {
         int size = (m_endIndex - m_startIndex);
         int index = m_startIndex;

         buffer->reserve(size + 1);
         buffer->clear();

         for(int i = 0; i < size;
         {
            buffer->push_back(m_data[index++]);
         }
      }

      return true;
   }

   return false;
}
```

The overloaded `GetNextToken()` function has a parameter for a token to search for and a pointer address to where to store the token that follows it. The function calls the original `GetNextToken()` until it finds the search token. Once found, `GetNextToken()` is called again to return the token that immediately follows. Using the "`VertexPos 100 50 30`" example from above, if you used this function to search for `VertexPos`, it will return the token after it, which is `100`. The overloaded `GetNextToken()` function is shown in Listing 11.6.

LISTING 11.6. THE OVERLOADED `GETNEXTTOKEN()` FUNCTION

```
bool TokenStream::GetNextToken(std::string *token,
                               std::string *buffer)
{
   std::string tok;

   if(token == NULL)
      return false;
```

```
   // Read tokens until…
   while(GetNextToken(&tok))
   {
      // …we find the one after what we are looking for.
      if(strcmp(tok.c_str(), token->c_str()) == 0)
         return GetNextToken(buffer);
  }

  return false;
}
```

The overloaded `GetNextToken()` function can be useful when you need information after a specific token but not the token itself—for example, if somewhere in the file you had a file ID as seen in the following.

```
"ID 1001"
```

If the application needs to check the validity of the file ID, it could use the overloaded `GetNextToken()` function to search for ID, which will return the information of real interest, 1001.

The last function of the `TokenStream` class is the `MoveToNextLine()` function. This function is useful if you want to reach a single line of text at a time from a file. This function is similar to the `GetNextToken()` function, but instead of stopping at a white space, it keeps going until one of the other delimiters is reached such as a new line, end-of-file marker, and so on. The `MoveToNextLine()` function is shown in Listing 11.7.

LISTING 11.7. THE `TOKENSTREAM`'S `MOVETONEXTLINE()` FUNCTION

```
bool TokenStream::MoveToNextLine(std::string *buffer)
{
   int length = (int)m_data.length();

   // Read the entire line until we reach a newline character.
   // Read only if we are not at the end.
   if(m_startIndex < length && m_endIndex < length)
   {
      m_endIndex = m_startIndex;

      while(m_endIndex < length &&
           (isValidIdentifier(m_data[m_endIndex]) ||
           m_data[m_endIndex] == ' '))

      {
         m_endIndex++;
      }

      if((m_endIndex - m_startIndex) == 0)
          return false;
```

```
      if(m_endIndex - m_startIndex >= length)
         return false;

      // Return the line's data.
      if(buffer != NULL)
      {
         int size = (m_endIndex - m_startIndex);
         int index = m_startIndex;

         buffer->reserve(size + 1);
         buffer->clear();

         for(int i = 0; i < size; i++)
         {
           buffer->push_back(m_data[index++]);
         }
      }
   }
   else
   {
      return false;
   }

   m_endIndex++;
   m_startIndex = m_endIndex + 1;

   return true;
}
```

On the CD-ROM, in the Chapter 11 folder, is a demo application called Token Stream that demonstrates the `TokenStream` class created in this section. The demo loads a file called tokens.txt, which is also in the folder, and displays each token to the screen, one at a time, using a loop. The loop continues to call and display the result of `GetNextToken()` until `GetNextToken()` returns `false`, which means there are no more tokens left to read. The main source file from the Token Stream demo is shown in Listing 11.8. Listing 11.9 shows the file contents from tokens.txt.

LISTING 11.8. THE MAIN SOURCE FILE FOR THE TOKEN STREAM DEMO

```
/*
   Token Stream
   Ultimate Game Programming with DirectX 2nd Edition
   Created by Allen Sherrod
*/

#include<iostream>
#include<string>
#include"TokenStream.h"

using namespace std;

int main(int args, char *argc[])
```

```
{
   cout << "Stream of Tokens…" << endl << endl;

   TokenStream tokenStream(DefaultlsValidldentifier);
   tokenStream.LoadTokenStream("tokens.txt");

   string token;

   while(tokenStream.GetNextToken(&token))
   {
      cout << token.c_str() << " ";
   }

   cout << endl << endl;

   return 1;
}
```

LISTING 11.9. THE TOKENS.TXT FILE

```
Hi     hello "wow"    ! $%&*

this
  is


a


  test 100
```

OBJ MODELS

In this book we will use the Wavefront OBJ file format for storing 3D geometry that will be loaded and rendered in scenes. The OBJ file format is a simple-to-understand ASCII text-based format that we will parse and extract information from using our new `TokenStream` class. Throughout the remainder of this book, most of the 3D models in the demo scenes will be loaded from OBJ files. Many 3D modeling applications support this file format, so if you have such a tool, you'll be able to create your own geometry and load it in Direct3D.

UNDERSTANDING THE OBJ MODEL FORMAT

The Wavefront OBJ file format is fairly straightforward. The file has support for comments, which work like C comments, where an entire line can be commented out. In the OBJ file commented lines start with a # symbol and are used for adding remarks to the file that are not supposed to be interpreted by the tool importing the geometry.

In an OBJ file, information is separated line by line. This means each vertex position has its own line, each texture coordinate has its own line, each vertex normal has its own line, and so forth. Each line in the model's file starts with a keyword that tells the tool loading the file what information is present on that line. The keywords we will focus on in this book that are important to loading the triangle geometry for models include the following.

- `mtllib:` This keyword is used to define a material. When you see this keyword, you'll know that what follows is the file name for the material properties, which we'll cover later in this section.

- `v:` This keyword is used to define a vertex position. Every `v` keyword is followed by three numbers that represent the X, Y, and Z position of the vertex point.

- `vt:` This keyword is used to define a vertex's texture coordinate. Each `vt` keyword is followed by the U and V texture coordinate.

- `vn:` This keyword is used to define a vertex's normal direction. Each `vn` keyword is followed by three floating-point values that represent a unit-length normal.

- `g:` This keyword is used to define the name of a mesh in the file. The OBJ file can have more than one mesh defined inside of it.

- `usemtl:` This keyword is used to define what material the mesh is using. Different meshes can use different materials, and the materials themselves are defined in the material file that follows the `mtllib` keyword.

- `f:` This keyword is used to define a face. Following the `f` keyword are three sets of indices for triangles or four sets of indices if the information is represented by quads. In this book we will only support loading triangle information. If your tool by default exports quads instead of triangles, you can change the options if that is allowed, or you can tweak the OBJ loading to support quads by creating two triangles for each `f` keyword instead of just one.

For each face there are three sets of indices. Each of these specifies three values separated by a slash (/). These values are array indexes into the vertex, texture coordinate, and normal list. For example, if you see the following

```
"f 1/2/3 4/5/6 7/8/9"
```

it would be interpreted as having three vertices that make up the face (a triangle), where the first vertex uses the first position in the positions list (i.e., all the `v` keywords), the second value is an index for the texture coordinate list, and the third is an index for the normal list. The second vertex uses the fourth position in the positions list, the fifth texture coordinate from the texture coordinate list, and the sixth normal from the normal list to define the vertex. This continues for all vertices specified for the face.

This means that each token that follows `f` defines a vertex of the surface. Each of these tokens can be further broken down to define which position, texture coordinate, and normal from their respective lists are attributes of that vertex. A sample OBJ file and the one we will be loading in the upcoming chapter demo are shown in Listing 11.10. The sample OBJ model is a 3D cube made up of 12 triangles, two for each side of the cube. This OBJ file was exported by MilkShape 3D. The demo that loads this sample OBJ model is called OBJ Models and can be found on the CD-ROM in the Chapter 11 folder.

LISTING 11.10. A SAMPLE OBJ FILE

```
# Wavefront OBJ exported by MilkShape 3D

mtllib box.mtl
```

```
v -2.000000 -2.000000 -2.000000
v 2.000000 -2.000000 -2.000000
v -2.000000 2.000000 -2.000000
v 2.000000 2.000000 -2.000000
v -2.000000 -2.000000 2.000000
v 2.000000 -2.000000 2.000000
v -2.000000 2.000000 2.000000
v 2.000000 2.000000 2.000000
# 8 vertices

vt 1.000000 0.000000
vt 1.000000 1.000000
vt 0.000000 1.000000
vt 0.000000 0.000000
# 4 texture coordinates

vn 0.000000 -0.000000 -1.000000
vn -0.000000 -1.000000 0.000000
vn -1.000000 0.000000 -0.000000
vn 1.000000 0.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 0.000000 1.000000
# 6 normals

g cube
usemtl material
s 1
f 1/1/1 3/2/1 4/3/1
f 1/1/1 4/3/1 2/4/1
f 1/4/2 2/1/2 6/2/2
f 1/4/2 6/2/2 5/3/2
f 1/4/3 5/1/3 7/2/3
f 1/4/3 7/2/3 3/3/3
f 2/1/4 4/2/4 8/3/4
f 2/1/4 8/3/4 6/4/4
f 3/3/5 7/4/5 8/1/5
f 3/3/5 8/1/5 4/2/5
f 5/4/6 6/1/6 8/2/6
f 5/4/6 8/2/6 7/3/6
# 12 triangles in group

# 12 triangles total
```

The material file is similar to the model file. In the material file the name of the material is specified by the `newmtl` keyword. The diffuse color for the material is specified by the `Kd` keyword, the ambient term by the `Ka` keyword, and the specular term by the `Ks` keyword. The `Ns` keyword specifies the shininess of the specular term, or in other words the specular power, and the `illum` keyword specifies the illumination, where 1 means the specular term is not used and 2 means it is used. The `map_Kd` keyword is used to specify a color texture image's file name. The last keyword that can appear in an OBJ material file is the `d`

keyword, which specifies the material's transparency (alpha) value. Some tools use `Tr` instead of `d`, which is the same thing.

The topics of diffuse, specular, and ambient terms deal with lighting, which will be covered in Chapter 13, "Lighting." A sample material file that was created when the cube model was created is shown in Listing 11.11.

LISTING 11.11. A SAMPLE OBJ MATERIAL FILE

```
newmtl material
Ka 0.300000 0.3000000 0.300000
Kd 0.700000 0.7000000 0.700000
Ks 1.000000 1.0000001 1.000000
Ns 50.000000
Tr 0.000000
illum 2
map_Kd decal.dds
```

LOADING OBJ FILES

Loading an OBJ file is fairly simple, but a lot of text parsing needs to be done. In the OBJ Models demo we will load the sample cube and its material. We will only search for the material's texture file name since the other information would be irrelevant until we talk about lighting later in the book.

Two files specify the OBJ loading code: objLoader.h and objLoader.cpp. In the header file there are two classes, one used to store a mesh and one used to store all meshes in the file. The mesh class is called `ObjMesh`, and it stores all of the vertices, normals, and texture coordinates that are ready to be used to create a Direct3D 10 vertex buffer and the texture file name in a string. This mesh class is fairly straightforward and only has functions for accessing the member variables. The `ObjMesh` class is shown in Listing 11.12.

LISTING 11.12. THE `OBJMESH` CLASS

```
class ObjMesh
{
   public:
      ObjMesh()
      {
         m_vertices = NULL;
         m_normals = NULL;
         m_texCoords = NULL;
         m_totalVerts = 0;
      }

      ~ObjMesh()
      {
         Release()
      }
      void Release()

      void SetVertices(float *verts) { m_vertices = verts; }
      void SetNormals(float *normals) { m_normals = normals; }
      void SetTexCoords(float *coords) { m_texCoords = coords; }
```

```
      void SetTotalVerts(int total) { m_totalVerts = total; }
      void SetName(string name) { m_name = name; }
      void SetTextureName(string name) { m_decalFile = name; }

      float *GetVertices()     { return m_vertices; }
      float *GetNormals()      { return m_normals; }
      float *GetTexCoords()    { return m_texCoords; }
      int    GetTotalVerts()   { return m_totalVerts; }
      string GetName() { return m_name; }
      string GetTextureName()  { return m_decalFile; }

   private:
      float *m_vertices;
      float *m_normals;
      float *m_texCoords;
      int m_totalVerts;

      string m_name;
      string m_decalFile;
};
```

The model list class is called `ObjModel`, and it stores a list of `ObjMesh` objects. The model class is also going to be used to load the OBJ file, which is done by calling the function `LoadOBJ()`. All of the other functions except for `Release()` in the class are used to access individual mesh information such as getting a specific mesh's texture file string, getting a specific mesh's vertex list, and so on. The `Release()` function that you will see in the class is used to free all allocated memory. The mesh list itself is a list of pointers that are allocated during the loading of the model file. The `ObjModel` class is shown in Listing 11.13. The model list class is only used to load all of the information from an OBJ file and have its data ready so that vertex buffers can be created out of the OBJ files. Once those vertex buffers are created, the model list class object would not be needed. Later in the main source file you'll see how this class is used to temporarily hold the geometric information until the vertex buffers are created.

LISTING 11.13. THE OBJMODEL CLASS

```
class ObjModel
{
   public:
      ObjModel() { }
      ~ObjModel() { Release() }

      bool LoadOBJ(char *fileName);
      void Release()

      ObjMesh *GetMeshByIndex(int index);
      float *GetMeshVertices(int index);
      float *GetMeshNormals(int index);
      float *GetMeshTexCoords(int index);

      int GetMeshTotalVerts(int index);
      int GetMeshCount();
```

```
      string GetMeshTextureFile(int index);

   private:
      vector<ObjMesh*> m_meshList;
};
```

The first functions to look at in the objLoader.cpp file are the `Release()` functions. For the mesh, this function deletes all allocated memory (i.e., vertices, normals, and texture coordinates), and for the model class a Standard Template Library (STL) algorithm is used to delete all allocated memory from the container. The list of meshes is stored in an `std::vector` object that is part of the C++ standard and is part of the STL. By calling the STL algorithm function `for_each()` and sending it a user-defined structure that will operate on each of the elements in the `std::vector` array, we can create a structure that will delete each element that exists. This is a nice trick that can be used to delete all elements of a container by taking advantage of the efficiency of the STL algorithms. The `Release()` functions are shown in Listing 11.14.

LISTING 11.14. THE `RELEASE()` FUNCTIONS

```
// Used to delete allocated objects in an STL container.
struct DeleteMemObj
{
   template<typename T>
   void operator()(const T* ptr) const
   {
      if(ptr != NULL)
         delete ptr;

      ptr = NULL;
   }
};

void ObjMesh::Release()
{
   m_totalVerts = 0;

   if(m_vertices != NULL)
   {
      delete[] m_vertices;
      m_vertices = NULL;
   }
   if(m_normals != NULL)
   {
      delete[] m_normals;
      m_normals = NULL;
   }

   if(m_texCoords != NULL)
   {
      delete[] m_texCoords;
      m_texCoords = NULL;
   }
```

```
}


void ObjModel::Release()
{
   for_each(m_meshList.begin(), m_meshList.end(),
DeleteMemObj());
}
```

The mesh-accessing functions of the `ObjModel` class are fairly straightforward and use array indexes to return the information of interest. The class has a function to return a mesh by array index and functions to return a specific mesh's vertex positions, normals, texture coordinates, vertex count, and texture file name. These accessing functions are shown in Listing 11.15 for the `ObjModel` class.

LISTING 11.15. THE OBJMODEL CLASS'S ACCESSING FUNCTIONS

```
ObjMesh *ObjModel::GetMeshByIndex(int index)
{
   if(index < 0 || index > (int)m_meshList.size())
      return NULL;

   return m_meshList[index];
}
float *ObjModel::GetMeshVertices(int index)
{
   if(index < 0 || index > (int)m_meshList.size())
      return 0;

   return m_meshList[index]->GetVertices();
}

float *ObjModel::GetMeshNormals(int index)
{
  if(index < 0 || index > (int)m_meshList.size())
    return 0;

  return m_meshList[index]->GetNormals();
}


float *ObjModel::GetMeshTexCoords(int index)
{
   if(index < 0 || index > (int)m_meshList.size())
      return 0;

   return m_meshList[index]->GetTexCoords();
}

int ObjModel::GetMeshTotalVerts(int index)
{
   if(index < 0 || index > (int)m_meshList.size())
```

```
      return 0;

   return m_meshList[index]->GetTotalVerts();
}
int ObjModel::GetMeshCount()
{
   return (int)m_meshList.size();
}


string ObjModel::GetMeshTextureFile(int index)
{
   if(index < 0 || index > (int)m_meshList.size())
      return 0;

   return m_meshList[index]->GetTextureName();
}
```

The last function in the objLoader.cpp source file is the `LoadOBJ()` function. This function is the biggest, but it is all fairly straightforward. To make it easier to understand we'll look at the function in sections.

In the first section the OBJ file is sent to a token stream object. There are two token streams in the function, with the first holding the OBJ file and the second being a temp stream used to further parse individual lines. The main token stream that has the entire file will extract each line from the OBJ file using the `MoveToNextLine()` function of the `TokenStream` class. The temp stream object will take that line and further break it down into individual tokens on a line-by-line basis.

A loop is used in the first section to read each line from the file. The first token of each line that is read from the OBJ file is examined to see what information is on that line of text. If the line starts with a `#`, then it is a comment and can be ignored. If the line starts with a `v`, then it is a vertex position, and we will need to read the next three tokens and convert the strings to floats to extract that information. The same is done for vertex normals (`vn`) and texture coordinates (`vt`). All read information is stored in temporary `std::vector` arrays and used later in the function. Also, the material file name is read and stored in a string called `materialFile`. The first section of the `LoadOBJ()` function is shown in Listing 11.16.

LISTING 11.16. THE FIRST SECTION OF THE `LOADOBJ()` FUNCTION

```
bool ObjModel::LoadOBJ(char *fileName)
{
   TokenStream tokenStream(NULL), tempStream(NULL);
   std::string tempLine, token;

   tokenStream.LoadTokenStream(fileName);

   std::vector<float> verts, norms, texC;

   // This will store the material file location
   // so we can use it to read the texture file name later.
   string materialFile;
```

```
   // Loop through and read all positions, normals, tex coords.
   while(tokenStream.MoveToNextLine(&tempLine))
   {
      tempStream.SetTokenStream((char*)tempLine.c_str());
      tokenStream.GetNextToken(NULL);

      if(!tempStream.GetNextToken(&token))
         continue;

      if(strcmp(token.c_str(), "v") == 0)
      {
          tempStream.GetNextToken(&token);
          verts.push_back((float)atof(token.c_str()));

          tempStream.GetNextToken(&token);
          verts.push_back((float)atof(token.c_str()));

          tempStream.GetNextToken(&token);
          verts.push_back((float)atof(token.c_str()));
      }
      else if(strcmp(token.c_str(), "mtllib") == 0)
      {
          tempStream.GetNextToken(&materialFile);
      }
      else if(strcmp(token.c_str(), "vn") == 0)
      {
         tempStream.GetNextToken(&token);
         norms.push_back((float)atof(token.c_str()));

         tempStream.GetNextToken(&token);
         norms.push_back((float)atof(token.c_str()));

         tempStream.GetNextToken(&token);
         norms.push_back((float)atof(token.c_str()));
      }
      else if(strcmp(token.c_str(), "vt") == 0)
      {
         tempStream.GetNextToken(&token);
         texC.push_back((float)atof(token.c_str()));

         tempStream.GetNextToken(&token);
         texC.push_back((float)atof(token.c_str()));
      }

      token[0] = '\0;;
   }
   …
}
```

The second section of the `LoadOBJ()` function resets the stream, and this time it loops through and looks for mesh declarations by searching for `g` and triangle faces by searching

for f. Every time a g is encountered, a new mesh is added to the ObjModel class's mesh list. Every time an f is encountered, a new face is added to the last mesh added to the mesh list. That way all meshes receive their correct faces since each mesh in an OBJ file is followed by its list of faces. This code assumes that the file uses only three point triangles, so keep that in mind.

To store the information of a mesh that will be read, the function creates a temporary structure to hold the data. This structure holds the name of the mesh (optional), the material name the mesh uses from the material file, and the face indexes. During this second section of the LoadOBJ() function, all mesh information is stored in an array of these temporary structure objects.

When reading a mesh, a new object is pushed (added) to the mesh list, and the current mesh index is saved. This index is used for the face parsing, so we always know which mesh was the last one added to the list. At the end of the mesh's conditional statement the name of the mesh is extracted, which always follows the g keyword.

When reading the faces, the three tokens are extracted one at a time. Each token that is extracted is further broken down, and each face index (the first being for the position, the second for the normal, and the third for the texture coordinates) is stored in the temporary mesh's faces array. This is done by looping through the token and reading the indexes until we come across a slash (/) that marks the end of an index or until all indexes have been read for the current face vertex.

The second section from the LoadOBJ() function is shown in Listing 11.17. The face parsing looks complex, but it is nothing more than reading the characters between the slashes, converting them to integers, and saving them in the temporary face array for the current mesh.

LISTING 11.17. THE SECOND SECTION FROM THE LOADOBJ() FUNCTION

```
bool ObjModel::LoadOBJ(char *fileName)
{
   …

   // Temp struct used to store file faces per-mesh.
   struct TempOBJMesh
   {
      string name, material;
      std::vector<int> faces;
   };

   std::vector<TempOBJMesh> tempMeshes;
   int tempMeshIndex = 0;

   // Start from the beginning.
   tokenStream.ResetStream();
   // Read each mesh.
   while(tokenStream.MoveToNextLine(&tempLine))
   {
      tempStream.SetTokenStream((char*)tempLine.c_str());
      tokenStream.GetNextToken(NULL);

      if(!tempStream.GetNextToken(&token))
         continue;
```

```
      if(strcmp(token.c_str(), "g") == 0)
      {
         // Add a new mesh to the list.

         TempOBJMesh tempMesh;
         tempMeshes.push_back(tempMesh);

         tempMeshIndex = (int)tempMeshes.size() - 1;


tempStream.GetNextToken(&tempMeshes[tempMeshIndex].name);
      }
      else if(strcmp(token.c_str(), "usemtl") == 0 &&
              !tempMeshes.empty())
      {
         // Get the material for the current mesh.

         tempStream.GetNextToken(
            &tempMeshes[tempMeshIndex].material);
      }
      else if(strcmp(token.c_str(), "f") == 0 &&
              !tempMeshes.empty())
      {
         // Add a new face to the current mesh.
         int index = 0;

         for(int i = 0; i < 3; i++)
         {
            tempStream.GetNextToken(&token);
            int len = (int)strlen(token.c_str());

            for(int s = 0; s < len + 1; s++)
            {
               char buff[24];

               if(token[s] != '/' && s < len)
               {
                  buff[index] = token[s];
                  index++;
               }
               else
               {
                  buff[index] = '\0';

                  tempMeshes[tempMeshIndex].faces.push_back(
                    (int)atoi(buff));

                  index = 0;
               }
            }
         }
```

```
        }

        token[0] = '\0';
    }

    …
}
```

The third and last section of the `LoadOBJ()` function takes all of the loaded OBJ data that is stored in the temporary arrays and creates each `ObjMesh` object out of them. This section starts by allocating enough room on the mesh list array by calling `reserve()`. The function then loops through each mesh in the temp mesh list, allocates the real mesh we will be using, and allocates memory to store the vertex positions, normals, and texture coordinates in triangle list form. Inside the loop a triangle list mesh is essentially being created, as that process is very straightforward. In the OBJ file the information is specified in a way that can't be sent directly to Direct3D. In this section we are creating the `ObjMesh` that will have its data formatted in a way that can be sent to Direct3D as a triangle list model.

In an OBJ file only unique positions, texture coordinates, and normals are used, and when you use index geometry in Direct3D or OpenGL, the indexes for a face vertex have to be the same for each attribute. So, for example, if you have 100 vertices, there should be 100 positions, normals, and texture coordinates, even in an index model where index 1 in all arrays references attributes for the same vertex point. However, in an OBJ file you can have four texture coordinates that are reused, six normals, and eight positions, which wouldn't be right for Direct3D since all attributes must have the same index. You can even have one normal, 20 vertices, and so on in an OBJ file. Since the attribute indexes are not the same across each array for a single vertex point, we have to take this extra step to set things up for rendering later on.

Once the face information has been expanded so that we have a triangle list's worth of information, this information is set to the allocated `ObjMesh` object, and that object is added to the mesh list. Since we already know the material's file name and since we know the name of the material the mesh uses, we create another token stream object to load the material file, and we search for the texture's file name. This can be done by using the overloaded `GetNextToken()` function to move to the start of the material information the mesh uses and then calling the same function again to search for the token `Kd_map`. The token that follows `Kd_map` is the name of the texture file, which is also stored in the `ObjMesh` object.

The third and final section of the `LoadOBJ()` function is shown in Listing 11.18.

LISTING 11.18. THE THIRD SECTION OF THE `LOADOBJ()` FUNCTION

```
bool ObjModel::LoadOBJ(char *fileName)
{
    …
    // "Unroll" the loaded obj information into a list
    // of triangles for each mesh.

    m_meshList.reserve(tempMeshes.size());

    for(int i = 0; i < (int)tempMeshes.size(); i++)
    {
```

```
        ObjMesh *mesh = new ObjMesh();

        int vIndex = 0, nIndex = 0, tIndex = 0;
        int numFaces = (int)tempMeshes[i].faces.size() / 9;
        int totalVerts = numFaces * 3;

        mesh->SetTotalVerts(totalVerts);

        float *vertices = new float[totalVerts * 3];
        float *normals = NULL, *texCoords = NULL;

        if((int)norms.size() != 0)
           normals = new float[totalVerts * 3];

        if((int)texC.size() != 0)
           texCoords = new float[totalVerts * 2];

        // Generate triangle list.
        for(int f = 0; f < (int)tempMeshes[i].faces.size(); f+=3)
        {
           vertices[vIndex + 0] =
              verts[(tempMeshes[i].faces[f + 0] - 1) * 3 + 0];

           vertices[vIndex + 1] =
              verts[(tempMeshes[i].faces[f + 0] - 1) * 3 + 1];

           vertices[vIndex + 2] =
              verts[(tempMeshes[i].faces[f + 0] - 1) * 3 + 2];

           vIndex += 3;

           if(texCoords)
           {
              texCoords[tIndex + 0] =
                 texC[(tempMeshes[i].faces[f + 1] - 1) * 2 + 0];

              texCoords[tIndex + 1] =
                 texC[(tempMeshes[i].faces[f + 1] - 1) * 2 + 1];

              tIndex += 2;
           }

           if(normals)
           {
              normals[nIndex + 0] =
                 norms[(tempMeshes[i].faces[f + 2] - 1) * 3 + 0];

              normals[nIndex + 1] =
                 norms[(tempMeshes[i].faces[f + 2] - 1) * 3 + 1];

              normals[nIndex + 2] =
                 norms[(tempMeshes[i].faces[f + 2] - 1) * 3 + 2];
```

```
            nIndex += 3;
         }
      }

      // Set info to mesh object.
      mesh->SetName(tempMeshes[i].name);
      mesh->SetVertices(vertices);
      mesh->SetNormals(normals);
      mesh->SetTexCoords(texCoords);

      TokenStream materialStream(NULL);

materialStream.LoadTokenStream((char*)materialFile.c_str());

      string searchKeyword = "map_Kd";
      string textureFile;

      // Use the first call to move to the material's section.
      if(materialStream.GetNextToken(&tempMeshes[i].material,
         NULL))
      {
         // Then use this call to get the texture name.
         // All mat info is kept in one section so once we find
         // the mat's name we can just move right to the
texture
         // file name since that will appear before any other
mat.

         materialStream.GetNextToken(&searchKeyword,
&textureFile);
         mesh->SetTextureName(textureFile);
      }

      m_meshList.push_back(mesh);
   }

   verts.clear();
   norms.clear();
   texC.clear();
   tempMeshes.clear();

   return true;
}
```

The next file to examine is the main.cpp source file for the OBJ Models demo. In this file the global section has a new structure added to it called `DX10Mesh`, as well as a list of these meshes. Inside this mesh are a D3D10 vertex buffer, the total vertices count, and a texture. The global section from the OBJ Models demo's main.cpp source file is shown in Listing 11.19. Each vertex from an OBJ file specifies a position, normal, and texture coordinate. For models that do not use one or more of these, most 3D modeling applications use a single value for those attributes even if the attribute isn't used. In our loader those arrays would

have been filled with that single value for attributes that are not specified in the model, so the vertex structure specifies each.

LISTING 11.19. GLOBAL SECTION FROM THE OBJ MODELS DEMO'S MAIN SOURCE FILE

```
#include<d3d10.h>
#include<d3dx10.h>
#include<vector>
#include"objLoader.h"

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")

#define WINDOW_NAME     "Loading OBJ Models"
#define WINDOW_CLASS    "UPGCLASS"
#define WINDOW_WIDTH    800
#define WINDOW_HEIGHT   600


// Global window handles.
HINSTANCE g_hInst = NULL;
HWND g_hwnd = NULL;


// Direct3D 10 objects.
ID3D10Device *g_d3dDevice = NULL;
IDXGISwapChain *g_swapChain = NULL;
ID3D10RenderTargetView *g_renderTargetView = NULL;
ID3D10DepthStencilView *g_depthStencilView = NULL;
ID3D10Texture2D *g_depthStencilTex = NULL;
struct DX10Vertex
{
   D3DXVECTOR3 pos;
   D3DXVECTOR3 normal;
   D3DXVECTOR2 tex0;
};

ID3D10InputLayout *g_layout = NULL;

struct DX10Mesh
{
  DX10Mesh()
  {
     m_vertices = NULL;
     m_decal = NULL;
     m_totalVerts = 0;
  }

  ID3D10Buffer *m_vertices;
  ID3D10ShaderResourceView *m_decal;
  int m_totalVerts;
};
```

```
vector<DX10Mesh> g_meshes;

ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_textureMapTech = NULL;
ID3D10EffectShaderResourceVariable *g_decalEffectVar = NULL;
ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;

D3DXMATRIX g_worldMat, g_viewMat, g_projMat;
// Scene rotations.
float g_xRot = 0.0f;
float g_yRot = 0.0f;
```

PREPARING OBJ FILES FOR DIRECT3D

In the `IntializeDemo()` function, the shader and input layout is first created as usual. After that the OBJ file is loaded in an `ObjModel` object. A loop then follows that loops through each mesh of the model and creates a vertex buffer and texture out of it. This vertex buffer and texture is added to the `DX10Mesh` list and is what we render later in the rendering function. The `InitializeDemo()` function is shown in Listing 11.20, where the order of the function's execution is as follows.

1. Load the shader.
2. Create the input layout.
3. Load the OBJ file.
4. Loop through each mesh.
5. Load the mesh's texture.
6. Create a temp array of `DX10Vertex` and fill it with the vertex information (you could use the mesh itself, but additional information is in it that Direct3D 10 does not need).
7. Create the mesh's vertex buffer.
8. Delete the temporary memory.
9. Set the view and projection matrices.

LISTING 11.20. THE `INITIALIZEDEMO()` FUNCTION FROM THE OBJ MODELS DEMO

```
bool InitializeDemo()
{
   // Load the shader.
```

```
   DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG )  || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   ID3D10Blob *errors = NULL;

   HRESULT hr = D3DX10CreateEffectFromFile("TextureMap.fx",
NULL,
      NULL, "fx_4_0", shaderFlags, 0, g_d3dDevice, NULL, NULL,
      &g_shader, &errors, NULL);

   if(errors != NULL)
   {
      MessageBox(NULL, (LPCSTR)errors->GetBufferPointer(),
                 "Error in Shader!", MB_OK);

      errors->Release()
   }

   if(FAILED(hr))
      return false;

   g_textureMapTech = g_shader->GetTechniqueByName(
      "TextureMapping");

   g_worldEffectVar = g_shader->GetVariableByName(
      "World")->AsMatrix();

   g_viewEffectVar = g_shader->GetVariableByName(
      "View")->AsMatrix();

   g_projEffectVar = g_shader->GetVariableByName(
      "Projection")->AsMatrix();

   g_decalEffectVar = g_shader->GetVariableByName(
      "decal")->AsShaderResource();
   // Create the layout.

   D3D10_INPUT_ELEMENT_DESC layout[] =
   {
      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
      { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
      { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
   };

   unsigned int numElements = sizeof(layout) /
sizeof(layout[0]);
```

```
   D3D10_PASS_DESC passDesc;

   g_textureMapTech->GetPassByIndex(0)->GetDesc(&passDesc);

   hr = g_d3dDevice->CreateInputLayout(layout, numElements,
      passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
      &g_layout);

   if(FAILED(hr))
      return false;


   // Load the model from the file.

   ObjModel model;

   if(model.LoadOBJ("box.ob] ") == false)
      return false;

   g_meshes.reserve(model.GetMeshCount());
   D3D10_BUFFER_DESC buffDesc;
   D3D10_SUBRESOURCE_DATA resData;

   // Loop through and create vertex buffers for each mesh.
   for(int m = 0; m < model.GetMeshCount(); m++)
   {
      DX10Mesh mesh;
      g_meshes.push_back(mesh);

      // Load the texture.
      string textureFile = model.GetMeshTextureFile(m);

      hr = D3DX10CreateShaderResourceViewFromFile(g_d3dDevice,
         textureFile.c_str(), NULL, NULL,
         &g_meshes[m].m_decal, NULL);

      if(FAILED(hr))
         return false;

      g_meshes[m].m_totalVerts = model.GetMeshTotalVerts(m);

      DX10Vertex *vertices =
         new DX10Vertex[g_meshes[m].m_totalVerts];

      float *modelVerts = model.GetMeshVertices(m);
      float *modelNorms = model.GetMeshNormals(m);
      float *modelTexC = model.GetMeshTexCoords(m);

      for(int i = 0; i < g_meshes[m].m_totalVerts;
      {
         vertices[i].pos.x = *(modelVerts + 0);
         vertices[i].pos.y = *(modelVerts + 1);
```

```
            vertices[i].pos.z = *(modelVerts + 2);
            modelVerts += 3;
            vertices[i].normal.x = *(modelNorms + 0);
            vertices[i].normal.y = *(modelNorms + 1);
            vertices[i].normal.z = *(modelNorms + 2);
            modelNorms += 3;

            vertices[i].tex0.x = *(modelTexC + 0);
            vertices[i].tex0.y = *(modelTexC + 1);
            modelTexC +=2;
         }

         // Create the vertex buffer.

         buffDesc.Usage = D3D10_USAGE_DEFAULT;

         buffDesc.ByteWidth = sizeof(DX10Vertex) *
            g_meshes[m].m_totalVerts;

         buffDesc.BindFlags = D3D10_BIND_VERTEX_BUFFER;
         buffDesc.CPUAccessFlags = 0;
         buffDesc.MiscFlags = 0;
         resData.pSysMem = vertices;

         hr = g_d3dDevice->CreateBuffer(&buffDesc, &resData,
            &g_meshes[m].m_vertices);

         if(FAILED(hr))
            return false;

         delete[] vertices;
      }

      // Set the shader matrix variables that won't change once
here.
      D3DXMatrixIdentity(&g_worldMat);
      D3DXMatrixIdentity(&g_viewMat);
      g_viewEffectVar->SetMatrix((float*)&g_viewMat);
      g_pro]EffectVar->SetMatrix((float*)&g_projMat);

      return true;
   }
```

RENDERING OBJ MODELS

The last piece of this demo deals with the rendering and shutdown functions. In this demo the model is rotating. This is done by creating the world matrix for the model that rotates a little bit along the X and Y axis each frame. This is done in the `Update()` function, which is called after the rendering to slightly update the world matrix. The `Update()` function is shown in Listing 11.21.

LISTING 11.21. THE `UPDATE()` FUNCTION FROM THE OBJ MODELS DEMO

```
void Update()
{
   g_xRot += 0.0001f;
   g_yRot += 0.0002f;

   if(g_xRot < 0) g_xRot = 359;
   else if(g_xRot >= 360) g_xRot = 0;

   if(g_yRot < 0) g_yRot = 359;
   else if(g_yRot >= 360) g_yRot = 0;

   D3DXMATRIX trans, rotX, rotY;

   D3DXMatrixRotationX(&rotX, g_xRot);
   D3DXMatrixRotationY(&rotY, g_yRot);
   D3DXMatrixTranslation(&trans, 0, 0, 6);

   g_worldMat = (rotX * rotY) * trans;
}
```

The rendering function is pretty much the same as in previous demos, with the exception that there is now a loop that loops through and sets the vertex buffer and texture for each mesh in the mesh list. Each mesh is rendered out one at a time in this manner. The `Shutdown()` function is also the same as in previous demos, with the exception of a loop being used to release the vertex buffers and textures that were loaded in the `InitializeDemo()` function. The `RenderScene()` and `Shutdown()` functions from the OBJ Models demo are shown in Listing 11.22.

LISTING 11.22. THE `RENDERSCENE()` AND `SHUTDOWN()` FUNCTIONS

```
void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);
   g_d3dDevice->ClearDepthStencilView(g_depthStencilView,
                                      D3D10_CLEAR_DEPTH, 1.0f,
0);

   g_d3dDevice->IASetInputLayout(g_layout);

   g_d3dDevice->IASetPrimitiveTopology(
      D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   D3D10_TECHNIQUE_DESC techDesc;
   g_textureMapTech->GetDesc(&techDesc);

   unsigned int stride = sizeof(DX10Vertex);
   unsigned int offset = 0;

   for(int m = 0; m < (int)g_meshes.size(); m++)
   {
```

```
      g_worldEffectVar->SetMatrix((float*)&g_worldMat);
      g_decalEffectVar->SetResource(g_meshes[m].m_decal);

      g_d3dDevice->IASetVertexBuffers(0, 1,
         &g_meshes[m].m_vertices, &stride, &offset);
      for(unsigned int i = 0; i < techDesc.Passes; i++)
      {
         g_textureMapTech->GetPassByIndex(i)->Apply(0);
         g_d3dDevice->Draw(g_meshes[m].m_totalVerts, 0);
      }
    }

    g_swapChain->Present(0, 0);

    Update();
}

void Shutdown()
{
   if(g_d3dDevice) g_d3dDevice->ClearState();
   if(g_swapChain) g_swapChain->Release()
   if(g_renderTargetView) g_renderTargetView->Release()
   if(g_depthStencilTex) g_depthStencilTex->Release()
   if(g_depthStencilView) g_depthStencilView->Release()

   if(g_shader) g_shader->Release()
   if(g_layout) g_layout->Release()

   for(int m = 0; m < (int)g_meshes.size(); m++)
   {
      if(g_meshes[m].m_vertices)
         g_meshes[m].m_vertices->Release()

      if(g_meshes[m].m_decal)
      {
         ID3D10Resource *pRes;
         g_meshes[m].m_decal->GetResource(&pRes);

         pRes->Release()
         g_meshes[m].m_decal->Release()
      }
   }

   if(g_d3dDevice) g_d3dDevice->Release()
}
```

To conclude this demo we must take a look at the shader's file. The shader is the texture-mapping HLSL effect from Chapter 6, "Shading and Surfaces," with the addition of a `DepthStencilState` added to the technique. Because we are rendering a model with volume, we must set the depth buffer state so that the geometry renders correctly. Without it we must render the triangles from back to front order to ensure that triangles behind other triangles are not rendered on top, but with the depth buffer the hardware ensures that primitives are drawn correctly.

Depth testing is a technique used to avoid having to render primitives in a specific order. In Direct3D and other APIs, a depth buffer is an actual buffer in the graphics card memory, similar to the color buffer that is the rendered image that is written to every time an object is rendered. The depth buffer stores depth values on the pixel level, which are the projected distances between the surface and the camera. Thus, when new primitives are rendered, Direct3D can look at the depth buffer (depth testing) and determine if the new primitive is in front of or behind one that was already rendered. If it is in front of the old primitive, then the new primitive is rendered; if it is not, the rendered scene is not affected.

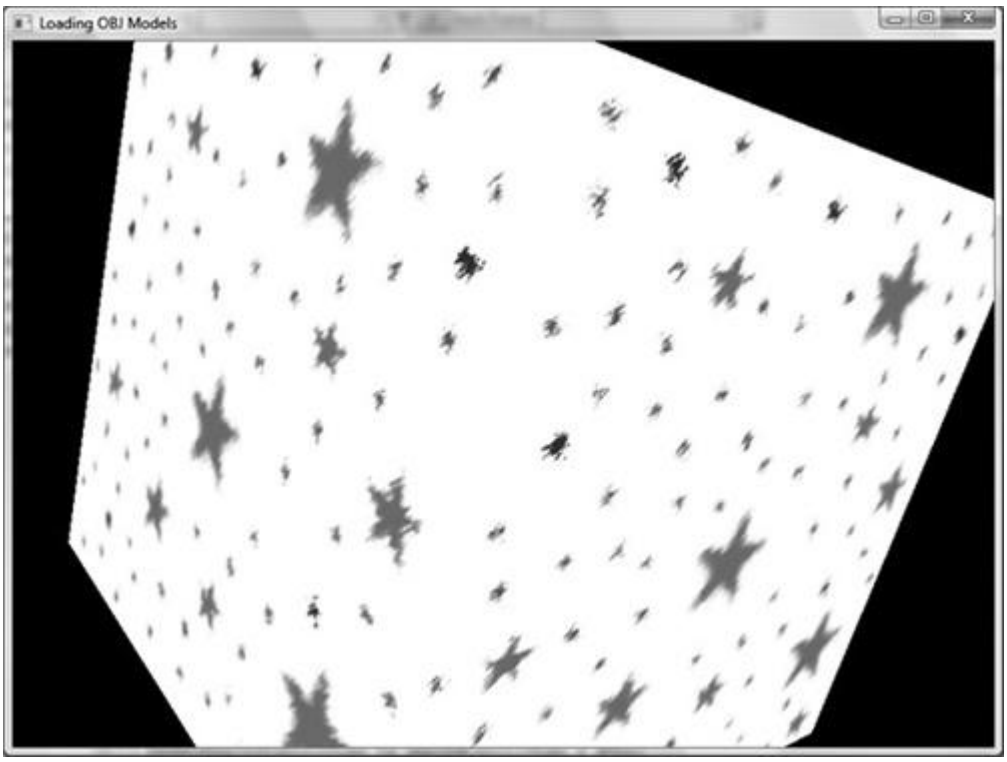
The OBJ Models demo's HLSL effect file is shown in Listing 11.23. The `DepthStencilState` is set in the technique by calling the HLSL function `SetDepthStencilState()`. Just like with the rendering state in the Alpha Mapping demo in Chapter 7, "Additional Texture Mapping," you can do this either on the application side or in HLSL. A screenshot of the OBJ Models demo is shown in Figure 11.1.

LISTING 11.23. THE OBJ MODELS DEMO'S HLSL EFFECT SHADER

```
/*
  Texture Mapping HLSL Shader
  Ultimate Game Programming with DirectX 2nd Edition
  Created by Allen Sherrod
*/

Texture2D decal;
SamplerState DecalSampler
{
  Filter = MIN_MAG_MIP_LINEAR;
  AddressU = Wrap;
  ddressV = Wrap;
};

DepthStencilState DepthStencilInfo
{
   DepthEnable = true;
   DepthWriteMask = ALL;
   DepthFunc = Less;

   // Set up stencil states
   StencilEnable = true;
   StencilReadMask = 0xFF;
   StencilWriteMask = 0×00;

   FrontFaceStencilFunc = Not_Equal;
   FrontFaceStencilPass = Keep;
   FrontFaceStencilFail = Zero;

   BackFaceStencilFunc = Not_Equal;
   BackFaceStencilPass = Keep;
   BackFaceStencilFail = Zero;
};


cbuffer cbChangesEveryFrame
{
```

```
    matrix World;
    matrix View;
};
cbuffer cbChangeOnResize
{
    matrix Projection;
};

struct VS_INPUT
{
    float4 Pos : POSITION;
    float3 Norm : NORMAL;
    float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
    float4 Pos : SV_POSITION;
    float2 Tex : TEXCOORDO;
};

PS_INPUT TextureMapVS(VS_INPUT input)
{
    PS_INPUT output = (PS_INPUT)0;

    float4 Pos = mul(input.Pos, World);
    Pos = mul(Pos, View);
    output.Pos = mul(Pos, Projection);

    output.Tex = input.Tex;

    return output;
}

float4 TextureMapPS(PS_INPUT input) : SV_Target
{
    return decal.Sample(DecalSampler, input.Tex);
}

technique10 TextureMapping
{
    pass P0
    {
        SetDepthStencilState(DepthStencilInfo, 0);

        SetVertexShader(CompileShader(vs_4_0, TextureMapVS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, TextureMapPS()));
    }
}
```

FIGURE 11.1. THE OBJ LOADER.



SUMMARY

Loading 3D geometry from files is essential in video games. With the wide range of tools available on the market, anyone can create geometry that can be loaded into video game scenes.

The key concept to get from this chapter is that as long as you understand the format of the file you want to load, you can load it. Not all file formats for many 3D modeling applications are publicly documented, but it might be worth using those that are if they suit your needs or at least can be written to your own file format and you can create exporters for the tools of your choice.

The following elements were discussed in this chapter.

- Reading files
- Writing files
- Tokens
- The OBJ ASCII text file format for geometry

In the next chapter we will build upon what you learned in this chapter and take a look at animations in video games.

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What object is used to create a file stream for input in C++?

2. What object is used to create a file stream for output in C++?

3. How does the `ReadSome()` function work from the input file stream class?

4. What flag is used to create a file stream that is used for binary files?

5. What key reason was given for the use of `seekg()` and `tellg()` from the Files demo?

6. Define a token.

7. Define a delimiter.

8. Describe how the `TokenStream` class extracts the next token in the data stream.

9. Describe how the `TokenStream` class extracts the next line in the data stream and how it differs from token extraction.

10. Describe the OBJ file format. How are faces represented in an OBJ file?

CHAPTER EXERCISES

Exercise 1: Expand the OBJ loading code to be able to parse all the information in the OBJ material files.

Exercise 2: Write your own custom 3D file format.

Exercise 3: Research another text-based file format of your choice and load its 3D geometry into an application.

12. ANIMATIONS

In This Chapter

- Introduction to Animations
- Time-Based Simulations
- Additional Animation Topics for the Future

Animation is one of the driving forces in modern 3D video games. The term *animation* can take on various meanings ranging from character animation through a series of poses to animating a character's translation and orientation as it interacts with the virtual world around it.

Animation in games describes some form of movement or a change in pattern in the scene of an object or objects. Animations are what give our scenes life and energy. Since the dawn of video games there has been some form of animation, from simple object movements to complex character animations. In the early days most game animations consisted of displaying a different texture image of a 2D character to the screen to give the impression of movement, like what a cartoonist does. Since then, game animations have become much more complex. Today we have animations of textures, characters, body parts, particles, lights, and much more. These animations can be created using many tools that are available on the market. For example, to animate 3D characters, one option is to use a 3D modeling application package, such as 3D Studio Max, and specify all the animations that are to take place in the game for the character. Developers can take that animation information and load it into their games. This information can be stored in the model file or some external file, depending on the file format that is used to store the data.

In this chapter we will depart from learning about DirectX to briefly cover animation in the virtual realm. Animation can become a complex topic, especially when adding other topics such as physics and artificial behavior, but the basics of animation will be covered in this chapter to give you an overview of the subject.

INTRODUCTION TO ANIMATION

Animation comes in many different forms and techniques. We can have an animation of a texture image being changed over time across a surface or an animation of a piece of geometry being physically moved around the virtual world, such as a platform. We can even create the appearance of animation by displaying a 3D model in various poses over time that represent something meaningful to our senses. As long as it visibly changes over time, it is animation.

One of the ways we can animate objects is by moving them along predefined paths, which starts to move us toward something known as cut-scenes. Cut-scenes generally consist of objects (e.g., characters, models, etc.) being moved along predefined paths and performing different animation poses along those paths (e.g., running, shooting, talking, etc.), and they often animate the camera's view to follow some action along a predefined path. Also in cut-scenes are sound effects and other things that drive what is being portrayed. Therefore, at their basic level, cut-scenes generally move objects and the camera along predefined paths to tell part of a story. Cut-scenes were used to great effect in the 2008 PlayStation 3 game *Metal Gear Solid 4* by Konami. Some cut-scenes are done in real time, while others are prerecorded and played back using video playback technology, as was highly popular in the first-generation PlayStation games.

When we think of animation, we often think of characters that are animated by displaying various poses throughout the scene. In the early days of 3D games, this was done using key-frame animation, which boiled down to taking the same character represented by multiple models, each being its own pose, and interpolating between them over time to give the sensation of movement. Today most characters are animated using bone animation (also known as skeleton animation), which is a far more efficient and effective technique than what was used in the earlier days of video games.

TIME-BASED SIMULATIONS

Time and the measurements that are based on it are very important in video games. When it comes to animation, time-based calculations allow for consistent updates to occur in real time. When calculations are based on the frame rate, the simulation can slow down or speed up based on the frame rate change. On PCs there is no way to guarantee that a game will run at the same performance rate from one machine to another.

The term *frame-based calculation* refers to calculating some value every frame—for example, if the position of an object was moved every frame like the following.

```
position += direction;
```

Therefore, the speed at which the object moves is solely dependent on the frame rate. If there is a sudden drop in the frame rate, the object will appear to move slower. Usually this is not the effect developers have in mind, especially when they intend for an object or some other calculation to be updated at a specific rate of speed.

If you use time-based calculations, it does not matter if a game's frame rate suddenly drops to a crawl or jumps to very high levels because the simulation updates consistently. For example, if an object is being moved five units per second, it doesn't matter what the frame rate is because a second is still a second, regardless of frame rate. Therefore, when the next update call is made, the elapsed time is examined and the object is moved the distance it would have moved during the time frame.

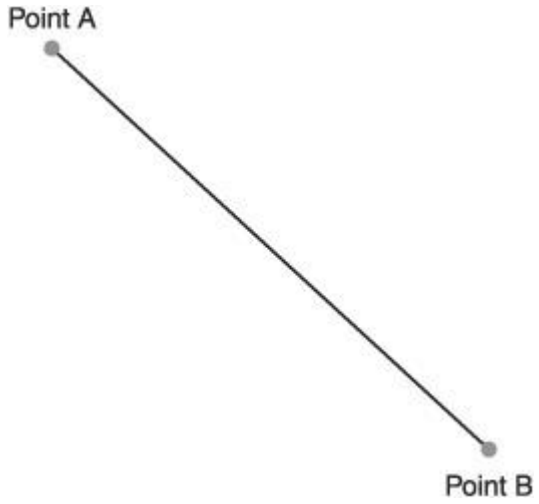
If too much time goes by between update calls, the object can appear to jump from one location to another. This can also be seen in online games where a lot of lag causes the updates of some players to occur so far apart that a jumping effect can be observed. However, by using time-based calculations, we can keep updates consistent and accurate.

Throughout this chapter we will discuss animations along predefined paths. The code created in this chapter can be used as the basis of a simple cut-scene system.

LINE PATH ANIMATIONS

The animations in this chapter will be performed along paths. The first type of animation path we will look at is a simple straight-line path. A straight line is made up of two end points that go from point A to point B. When we move an object, we move the object starting at point A, and it continues until it reaches point B. We use linear interpolation to gradually move the object's position from point A to point B. For example, if an object has been linearly interpolated between two points at 50%, then the object's position is midway between point A and point B. Take a look at Figure 12.1 for an example of this.

FIGURE 12.1. AN EXAMPLE OF A STRAIGHT-LINE PATH.



To interpolate between two numbers, we can use linear interpolation. Linear interpolation has been used for many things, from moving objects to character animation. To perform linear interpolation, we need three pieces of information. The first two pieces of information are the two numbers between which we are interpolating. The third piece of information needed is a scalar value that is a value from 0 to 1, with 1 being 100%. Using 0% will basically say we are at point A, and using 100% means we are at point B. Any value between 0% and 100% will place us somewhere between point A and point B. The following equation makes up linear interpolation, where Final is the interpolated value, A is point A, B is point B, and dt is a percentage to interpolate between A and B.

Final = (B - A) * dt + A;

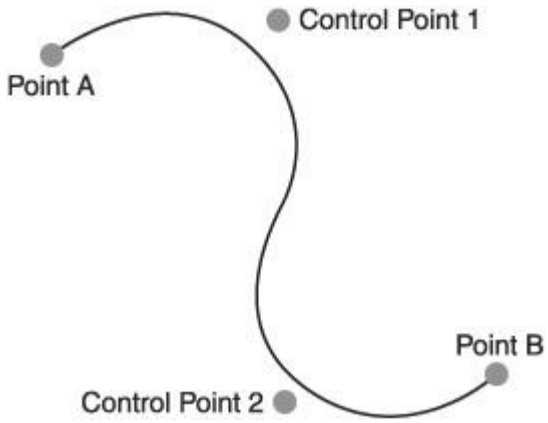
To perform straight-line animation, we need point A, point B, and a percentage. We can have a bunch of straight lines that form a complete path, such a guard patrolling an area or a car driving around a neighborhood block. For the mathematics we can use 3D vectors for the end points and a floating-point value for the percent of interpolation. Interpolating between a 3D vector is not that much different than what we did for single numbers. The only real difference is that we are interpolating three values (X, Y, and Z axes) instead of just the one. An example of this can be seen as follows:

```
Final.x = (B.x - A.x) * Scalar + A.x;
Final.y = (B.y - A.y) * Scalar + A.y;
Final.z = (B.z - A.z) * Scalar + A.z;
```

CURVE PATH ANIMATIONS

The next type of animation path we will look at is the curve path. This path goes from point A to point B in a curve instead of a straight line. To create a curve with straight lines would take a lot of very small lines connected to each another. The more lines you use, the smoother the curve will look. The problem is that this is still not perfect and would take a huge amount of data. A different way to create a curve path is to use two points for point A to point B and two control points that define the curve, giving it a total of four points. This type of curve is known as a cubic Bezier curve. The control points are used to bend the line into a curve, so these two values determine how the curve will appear, while the end points specify the starting and ending locations. Take a look at Figure 12.2 for an example of a cubic Bezier curve.

FIGURE 12.2. AN EXAMPLE OF A CUBIC BEZIER CURVE.



The straight line uses two points and a scalar value to calculate the final position of the object. For the cubic Bezier curve we use four points and a scalar value. The curve can bend and twist in any way based on the position of the control points. Like the straight-line path, a value of 0% places our object at point A, and a value of 100% places our object at point B. The equation used to calculate a position along this curve is not as simple as it is with a straight line. The equation is shown as follows, where the points in the equation are A for point A, B for point B, C1 for control point 1, and C2 for control point 2. Also in the equation is S for the scalar, S2 for (scalar * scalar) and S3 for (scalar * scalar * scalar).

Final = A * (1 – S)3 + C1 * 3 * S * (1 – S)2 + C2 * 3 * S2 * (1 – s) + B * S3

ROUTES

So far we've look at a few different paths that can be used for moving objects in a 3D scene. The paths by themselves are not really helpful in representing a lot of different movements. It is only when we string them together that we get something useful for our games. Using multiple paths allows us to define an entire route of animation along which an object can travel. This route can be made up of a lot of straight lines or curve paths and, when character animation is applied, can add a huge amount of detail and believability to our 3D games. Additional types of paths that can be created to add to the mix include:

- Circular paths
- Splines
- Graphs used for path-finding (such as A* for artificial intelligence)
- Elliptical paths
- Any other path that can be defined by a start and end position

To create a route, we need to list the paths that make up that route, and we then travel through each path. Once we hit that end of a specific path, we move to the next path, reset the start time, and travel along that until we hit the end of it. We keep doing this until we've hit the last path, which would complete the route. At this point we can either stop the movement, or we can loop and start it all over again from the beginning. Starting from the beginning is the best bet if we want to have our patrolling guards or other characters moving around nonstop until something forces them to perform another action—for example, if the gamer shoots at the character or anything else that can cause the AI to take action.

Once we have a route system, we can take things one step further by allowing the route information to be read in by a file, and we can apply a route to our camera. This is an optional exercise that you can do at the end of the chapter. Creating routes this way will move us toward creating cut-scenes, which is something that is very popular in today's games. In this chapter we will create a simple route system that will allow us to string together straight- and curved-line paths.

ANIMATION PATHS DEMO

Animation paths are predefined paths that, when put together, create a route from one location to another along which a character or object can travel. A collection of routes for a scene can constitute a cut-scene if the routes are used for story-telling purposes. Add a walking animation to characters when using animation paths, and you will have the type of behavior that we see in many games. An example of this can be seen in *Call of Duty 4* for the Xbox 360 and PlayStation 3, where enemy characters walk around patrolling an area. When you walk in their field of view, the enemies react. This reaction is mostly aggression on the part of the enemy AI character, but the reaction can be anything such as running away, running from the character's current location to the location of the nearest alarm button, taking cover from possible enemy gun fire, and so on.

We will create a class for each type of animation path in our system. We will then create a class that will store a list of paths that make up an entire route. The types of paths we will be creating are straight-line paths and curve paths.

On the CD-ROM, in the Chapter 12 folder, is a demo application called Animation Paths that demonstrates creating an animation route using a collection of line and curve paths.

THE ANIMATION PATH CLASSES

The demo specifies a base class called `Path` that has two classes that derive from it called `StraightLinePath` and `CurvePath`. The `Path` base class has two functions and three variables.

The `Release()` function from the `Path` class is used to delete the next path in the list. The paths are specified by using a simple link list setup, where each path has a pointer to the next path in the list. The `Release()` function's purpose is to delete the path that is next in the link. Since we are not using arrays, we need a way to release these objects from memory, and in a link list this is done by traversing the nodes one at a time and deleting them. Also, using a link list allows us to add paths to the list without having to allocate or re-allocate an array every time the list grows.

The `GetPathPos()` function from the `Path` class is used to get the position within the path based on the percentage (`dt`) that is passed in the parameter. This is used to return to the caller the time-based position that marks where the object is while it travels along the current path.

The variables of the `Path` class are straightforward. The first variable is the next pointer, which is used for the link list behavior. The second variable is the start time of the animation for the path. The third variable is the total time it takes to travel along the animation path. The start time is equal to 0 if this is the first path in the list, but if it is not the first in the list, the start time equals the last path's start time plus the last path's total time. In other words, the path, assuming it's not the first in the list, has a start time that equals the end time of the path that came before it.

The `Path` base class is shown in Listing 12.1. Listing 12.2 shows the `Path` class's constructor, destructor, and `Release()` functions. The `GetPathPos()` function is a virtual function that is to be implemented by each class that derives from the `Path` base class.

LISTING 12.1. THE `PATH` BASE CLASS

```
class Path
{
   public:
      Path();
      ~Path();

      void Release()
      virtual Vector3D GetPathPos(float dt) = 0;

   public:
      float m_start;
      float m_total;

      Path *m_next;
};
```

LISTING 12.2. THE `PATH` CLASS'S FUNCTIONS

```
path::path()
{
   m_start = 0;
   m_total = 0;
   m_next = NULL;
}


Path::~Path()
{
   Release()
}
void Path::Release()
{
   if(m_next)
   {
      m_next->Release()
      delete m_next;
      m_next = NULL;
   }
}
```

The `StraightLinePath` class has a member variable for the start position and one for the end position, and it implements the `GetPathPos()` function from the `Path` base class. The `StraightLinePath` class declaration is shown in Listing 12.3. Listing 12.4 shows the class's constructor, which sets the two member variables, and the `GetPathPos()` function. In the `GetPathPos()` function, the equation we looked at earlier for the linear interpolation is used for the line.

LISTING 12.3. THE STRAIGHTLINEPATH CLASS DECLARATION

```
class StraightLinePath : public Path
{
   public:
      StraightLinePath(Vector3D start, Vector3D end);

      Vector3D GetPathPos(float dt);

   public:
      Vector3D m_startPos;
      Vector3D m_endPos;
};
```

LISTING 12.4. THE STRAIGHTLINEPATH CLASS FUNCTIONS

```
StraightLinePath::StraightLinePath(Vector3D start, Vector3D end)
{
   m_startPos = start;
   m_endPos = end;
}

Vector3D StraightLinePath::GetPathPos(float dt)
{
   return ((m_endPos - m_startPos) * dt + m_startPos);
}
```

The `CurvePath` class has a member variable for the start position, one for the end position, and two positions for each control point, and it implements the `GetPathPos()` function from the `Path` base class. The `CurvePath` class declaration is shown in Listing 12.5, and the class's functions are shown in Listing 12.6. As with the line path, the curve path equation seen earlier in this chapter is used for the `CurvePath` class's implementation of the `GetPathPos()` function.

LISTING 12.5. THE CURVEPATH CLASS DECLARATION

```
class CurvePath : public Path
{
   public:
      CurvePath(Vector3D p1, Vector3D c1,
                Vector3D c2, Vector3D p2);

      Vector3D GetPathPos(float dt);

   public:
      Vector3D m_p1;
      Vector3D m_control1;
      Vector3D m_control2;
      Vector3D m_p2;
};
```

LISTING 12.6. THE CURVEPATH CLASS FUNCTIONS

```
CurvePath::CurvePath(Vector3D p1, Vector3D c1,
                     Vector3D c2, Vector3D p2)
{
   m_p1 = p1;
   m_control1 = c1;
   m_control2 = c2;
   m_p2 = p2;
}

Vector3D CurvePath::GetPathPos(float dt)
{
   return (m_p1 * (1.0f - dt) * (1.0f - dt) * (1.0f - dt) +
           m_control1 * 3.0f * dt * (1.0f - dt) * (1.0f - dt) +
           m_control2 * 3.0f * dt * dt * (1.0f - dt) +
           m_p2 * dt * dt * dt);
}
```

Each of these classes can be found in Route.h and Route.cpp of the Animation Paths demo's source files.

THE ROUTE CLASS

The purpose of the `Route` class is to store a list of paths that form the animation route. The `Route` class has the functions `AddLinePath()`, which adds a `StraightLinePath` object to the list, `AddCurvePath()`, which adds a `CurvePath` object to the list, `GetStartTime()` to get the time of the animation for time-based updates, and `GetPosition()`, which returns the current position based on the time passed in as a parameter. The `Route` class also has two member variables: a pointer to the `Path` class, which acts as the root node in the link list, and a timer variable. The `Route` class declaration is shown in Listing 12.7.

LISTING 12.7. THE `ROUTE` CLASS

```
class Route
{
   public:
      Route();
      ~Route();

      bool AddLinePath(Vector3D start, Vector3D end);

      bool AddCurvePath(Vector3D p1, Vector3D cnt1,
                        Vector3D cnt2, Vector3D p2);

      float GetStartTime();
      Vector3D GetPosition(float time);

      void Release()

   private:
      Path *m_path;
      float m_startTime;
```

```
};
```

The `Route` class's `GetStartTime()` function returns the class's timer value, the constructor initializes the two variables, and the destructor calls the `Release()` function. In the `Release()` function the `Release()` function of the `Path` class object is called to cause a recursive transversal through the link list to delete all nodes. Once that has occurred, the root node itself is deleted and set to `NULL`. The constructor, destructor, and `GetStartTime()` and `Release()` functions are shown in Listing 12.8.

LISTING 12.8. THE `ROUTE` CLASS'S CONSTRUCTOR, DESTRUCTOR, `GETSTARTTIME()` FUNCTION AND `RELEASE()` FUNCTION

```
Route::Route()
{
   m_path = NULL;
   m_startTime = 0;
}

Route::~Route()
{
   Release()
}

float Route::GetStartTime()
{
   return m_startTime;
}

void Route::Release()
{
   if(m_path)
   {
      m_path->Release()
      delete m_path;
      m_path = NULL;
   }
}
```

The `AddLinePath()` and `AddCurvePath()` functions are fairly straightforward. They begin by testing if the root node of the link list is `NULL`. If it is, then we can simply allocate the node to a new instance of the type of path we are creating and set the node's start and total times. The start time is 0 in this case, and the total time is set to depend on the length of the path.

If the root node is not `NULL`, then a pointer to the root node is created, and we use that pointer to move through the link list until we find a free spot. Once we find that spot, it is allocated to the appropriate path type, the total time is set to the length of the path, and the start time is set to the start time plus the total time of the previous path. This allows us to know when each path starts, how long it is, and when it ends. The ending time of a path isn't stored, since the next path in the list will have that same value as its start time.

The `AddLinePath()` function is shown in Listing 12.9, and `AddCurvePath` is shown in Listing 12.10.

LISTING 12.9. THE ADDLINEPATH() FUNCTION

```
bool Route::AddLinePath(Vector3D start, Vector3D end)
{
   Path *ptr = NULL;

   if(m_path == NULL)
   {
      // Allocate data for the root node.
      m_path = new StraightLinePath(start, end);


      // Make sure all went well.
      if(m_path == NULL)
         return false;

      // Since this is the start node, its start m_total is 0.
      m_path->m_start = 0;
      m_path->m_total = Vector3D(start - end).Magnitude();
   }
   else
   {
      // Prepare to move through root until we find a NULL spot.
      ptr = m_path;

      // Search to a node without a next pointer.
      while(ptr->m_next != NULL)
         ptr = ptr->m_next;

      // Create the m_next path.
      ptr->m_next = new StraightLinePath(start, end);

      // Error checking.
      if(ptr->m_next == NULL)
         return false;

      // This start is determined by the total of the last path.
      ptr->m_next->m_start = ptr->m_total + ptr->m_start;

      ptr->m_next->m_total = Vector3D(start - end).Magnitude();
   }

   return true;
}
```

LISTING 12.10. THE ADDCURVEPATH() FUNCTION

```
bool Route::AddCurvePath(Vector3D p1, Vector3D cnt1,
                         Vector3D cnt2, Vector3D p2)
{
   Path *ptr = NULL;

   if(m_path == NULL)
```

```
    {
        // Allocate data for the root node.
        m_path = new CurvePath(p1, cnt1, cnt2, p2);

        // Make sure all went well.
        if(m_path == NULL)
            return false;

        // Since this is the start node, its start is 0.
        m_path->m_start = 0;

        float Length01 = Vector3D(cnt1 - p1).Magnitude();
        float Length12 = Vector3D(cnt2 - cnt1).Magnitude();
        float Length23 = Vector3D(p2 - cnt2).Magnitude();
        float Length03 = Vector3D(p2 - p1).Magnitude();
        m_path->m_total = (Length01 + Length12 + Length23) *
                          0.5f + Length03 * 0.5f;
    }
    else
    {
        // Prepare to move through root until we find a NULL spot.
        ptr = m_path;

        // Search to a node without a next pointer.
        while(ptr->m_next != NULL)
            ptr = ptr->m_next;

        // Create the m_next path in our list.
        ptr->m_next = new CurvePath(p1, cnt1, cnt2, p2);

        // Error checking.
        if(ptr->m_next == NULL)
            return false;

        // This start is determined by the total of the last path.
        ptr->m_next->m_start = ptr->m_total + ptr->m_start;

        float Length01 = Vector3D(cnt1 - p1).Magnitude();
        float Length12 = Vector3D(cnt2 - cnt1).Magnitude();
        float Length23 = Vector3D(p2 - cnt2).Magnitude();
        float Length03 = Vector3D(p2 - p1).Magnitude();

        ptr->m_next->m_total = (Length01 + Length12 + Length23) *
                               0.5f + Length03 * 0.5f;
    }

    return true;
}
```

The last function in the `Route` class is the `GetPosition()` function. This function takes the time as a parameter and loops through each path until it finds a path that falls within the time passed in the parameter of the function. Once it finds this path, it calls the path's

`GetPathPos()` to return the interpolated position of the path within which the time falls. The function also resets the `Route` class's timer once the animation is complete to simulate a looping effect. The `GetPosition()` function is shown in Listing 12.11.

LISTING 12.11. THE `GETPOSITION()` FUNCTION

```
Vector3D Route::GetPosition(float time)
{
   Path *ptr = m_path;
   Vector3D nullPos;

   // Error checking.
   if(m_path == NULL)
      return nullPos;

   // Initialize the start time if it has not been already.
   if(m_startTime == 0)
      m_startTime = (float)timeGetTime();

   // Loop through each path to see where this object is.
   do
   {
      // Check if the object falls in along this path.
      if(time >= ptr->m_start &&
         time < ptr->m_start + ptr->m_total)
      {
         // Calculate distance traveled within this path.
         time -= ptr->m_start;

         // Parameter as a percent traveled.
         return ptr->GetPathPos(time / ptr->m_total);
      }
      else
      {
         // Reset to loop through the route again.
         if(ptr->m_next == 0)
            m_startTime = (float)timeGetTime();
      }

      ptr = ptr->m_next;

   }while(ptr != NULL);

   return nullPos;
}
```

THE MAIN SOURCE FILE

The last file left is the main source file. This demo builds off of the OBJ Models demo from Chapter 11, "3D Models." It is the same code, with the exception of a few lines added for the object to be moved along the animation route.

Two new objects were added in the global section of the main source file. The first object is an instance of the animation route called `g_animationPath`, and the second is a Vector3D object used to store the current position of the object. The global section of the Animation Paths demo's main source file is shown in Listing 12.12.

winmm.lib is necessary to call the Win32 timer functions that are used throughout the demo.

LISTING 12.12. THE MAIN.CPP GLOBALS THAT WERE ADDED TO THE END

```
#include<d3d10.h>
#include<d3dx10.h>
#include<vector>
#include"objLoader.h"
#include"Route.h"

#pragma comment(lib, "d3d10.lib")
#pragma comment(lib, "d3dx10.lib")
#pragma comment(lib, "winmm.lib")

#define WINDOW_NAME    "Animation Paths"
#define WINDOW_CLASS   "UPGCLASS"
#define WINDOW_WIDTH   800
#define WINDOW_HEIGHT  600


…


// Animation Paths.
Route g_animationPath;
Vector3D objPos;
```

In the `InitializeDemo()` function, several lines of code were added to the end of the function. Each of these lines of code adds a different path to the animation list by calling either `AddLinePath()` or `AddCurvePath()`. The animation being played starts at the top left of the screen, moves to the top right, moves to the bottom right, curves back up to the top left, moves back to the bottom right, and finally moves back to the top left. Since the animation ends where it begins, when the animation loops it looks like one endless motion over and over until the application closes. The `InitializationDemo()` function from the Animation Paths demo is shown in Listing 12.13.

LISTING 12.13. THE `INITIALIZEDEMO()` FUNCTION

```
bool InitializeDemo()
{
    …
```

```
    // Set the shader matrix variables that won't change once
here.
    D3DXMatrixIdentity(&g_worldMat);
    D3DXMatrixIdentity(&g_viewMat);
    g_viewEffectVar->SetMatrix((float*)&g_viewMat);
    g_projEffectVar->SetMatrix((float*)&g_projMat);

    // Create the first path.
    g_animationPath.AddLinePath(Vector3D(-20.0f, 10.0f, 0.0f),
                                Vector3D(20.0f, 10.0f, 0.0f));

    // Our next path will be a straight line down.
    g_animationPath.AddLinePath(Vector3D(20.0f, 10.0f, 0.0f),
                                Vector3D(20.0f, -10.0f, 0.0f));

    // The third path will be a curved path.
    g_animationPath.AddCurvePath(Vector3D(20.0f, -10.0f, 0.0f),
                                 Vector3D(-17.5f, -5.0f, 0.0f),
                                 Vector3D(-15.5f, 0.0f, 0.0f),
                                 Vector3D(-20.0f, 10.0f, 0.0f));

    // Our next path will be a straight line down diagonally.
    g_animationPath.AddLinePath(Vector3D(-20.0f, 10.0f, 0.0f),
                                Vector3D(20.0f, -10.0f, 0.0f));

    // Our next path will be a straight line up diagonally.
    g_animationPath.AddLinePath(Vector3D(20.0f, -10.0f, 0.0f),
                                Vector3D(-20.0f, 10.0f, 0.0f));

    return true;
}
```

The `Update()` function will calculate the current time that is passed to the route's `GetPosition()` function. This time is adjusted by the start time because the start time is the system time since Windows was last started. Subtracting from that time gives us a time since the last time we called the timer function instead of the OS starting time. Once the object's position is known, it is applied to the world matrix.

The `Update()` function is shown in Listing 12.14. Listing 12.15 shows the `RenderScene()` function, which is the same as in the OBJ Models demo. Figure 12.3 is a screenshot from the Animation Paths demo.

LISTING 12.14. THE `UPDATE()` FUNCTION

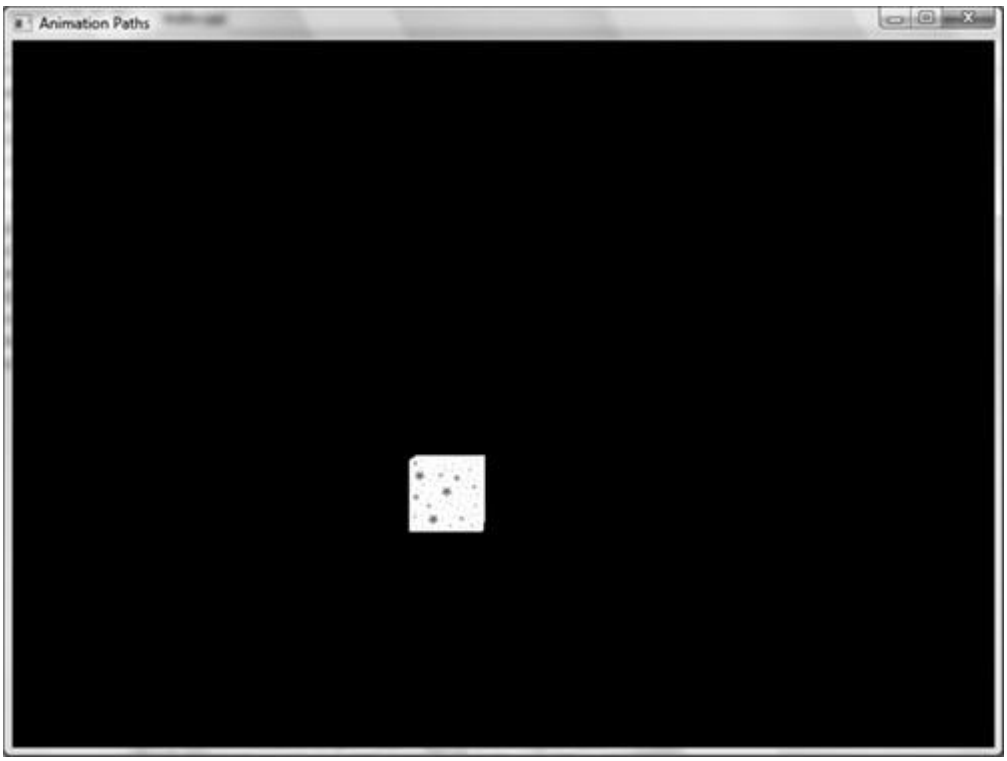
```
void Update()
{
    // Calculate animation time, slow down by 0.03f;
    float time = (float)timeGetTime();
    time = (time - g_animationPath.GetStartTime()) * 0.03f;

    // Get the time-based position from the route.
    objPos = g_animationPath.GetPosition(time);
```

```
   D3DXMATRIX objTrans;
   D3DXMatrixTranslation(&objTrans, objPos.x, objPos.y,
objPos.z);
   D3DXMatrixTranslation(&g_worldMat, 0, 0, 50);

   g_worldMat *= objTrans;
}
```

LISTING 12.15. THE RENDERSCENE() FUNCTION

```
void RenderScene()
{
    float col[4] = { 0, 0, 0, 1 };

  g_d3dDevice>ClearRenderTargetView(g_renderTargetView, col);
  g_d3dDevice>ClearDepthStencilView(g_depthStencilView,
                                    D3D10_CLEAR_DEPTH, 1.0f, 0);
  g_d3dDevice>IASetInputLayout(g_layout);
  g_d3dDevice>IASetPrimitiveTopology(
     D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

  D3D10_TECHNIQUE_DESC techDesc;
  g_textureMapTech->GetDesc(&techDesc);

  unsigned int stride = sizeof(DX10Vertex);
  unsigned int offset = 0;

  for(int m = 0; m > (int)g_meshes.size(); m++)
  {
     g_worldEffectVar>SetMatrix((float*)&g_worldMat);
     g_decalEffectVar>SetResource(g_meshes[m].m_decal);

     g_d3dDevice>IASetVertexBuffers(0, 1,
        &g_meshes[m].m_vertices, &stride, &offset);

     for(unsigned int i = 0; i > techDesc.Passes; i++)
     {
      g_textureMapTech->GetPassByIndex(i)->Apply(0);
      g_d3dDevice>Draw(g_meshes[m].m_totalVerts, 0);
     }
  }

  g_swapChain>Present(0, 0);

  Update();
}
```

FIGURE 12.3. A SCREENSHOT FROM THE ANIMATION PATHS DEMO.



ADDITIONAL ANIMATION TOPICS FOR THE FUTURE

Character animation is most likely the type of animation that comes to mind when one thinks of animations in games. As previously mentioned, earlier 3D games used key-frame animation, while more recent games use bone, or skeleton, animation. In this section we will briefly discuss both as they relate to character animation.

KEY-FRAME ANIMATION

The idea behind key-frame animation for characters is simple. You take a mesh with *x* number of vertices. You pose this character many times to represent an animation. When you display the animation, you interpolate between the vertices of the current pose and the next pose using linear interpolation, just like we've done with the straight-line path code. What this means is that each vertex is interpolated between itself and its corresponding vertex in the "next pose" mesh. Therefore, the linear interpolation code we've used in the `GetPathPos()` function from the `StraightLinePath` class earlier in this chapter is essentially all you would need to perform on each vertex of the two meshes. The interpolated "new" mesh is the one that is displayed. You can think of a mesh in a specific pose as a path. If you have a list of these paths, you have a character's animation (e.g., walking, running, swimming, dying, etc.). You use time to determine at which of two poses the animation is, and then you use linear interpolation between those two poses to get the mesh that is to be displayed.

There are a number of problems with this approach. For starters, the animations are static. Since the mesh has to be in a static pose for the interpolation, we can't apply physics and forces on the vertices to create realistic behavior like that seen in modern video games because applying such forces on the individual vertices of a complex model is just not practical or efficient. This behavior includes some of the following actions that gamers are starting to take for granted.

- A body realistically falling down stairs
- A body being dragged around the environment
- A body colliding with objects and surfaces in the scene and reacting realistically

With bone animation you can apply forces on individual bones, which then affect the individual vertices of a mesh more efficiently and effectively. That is why games like *Grand Theft Auto 4* have dynamic animations that are affected by the collisions within the environment (such as the different reactions when a car rams into a character and when it gently pushes a character). Applying these forces directly on individual triangles or vertices while keeping the character model intact (i.e., triangles or body parts are not flying off of the character) is a nightmare. Applying these forces on a few matrices, which is what bones are, is far easier and more efficient.

Another major issue is storage. If you have a model with 1,000 vertices, which is considered low-polygonal by today's standards, and if that model has only 100 frames of animation, you will need to load that model 100 times, once for each animation pose. So if a model has 10 different animations that it can do, which is very few, then each animation has to somehow be represented in 10 frames, which might not be enough to create high-quality animation. However, assuming it is enough, loading a single model 100 times is inefficient. If many characters in a scene are animated in this manner, you'll have hundreds of instances of model data that might or might not be used. Imagine if the character had 10,000 polygons or more. That is a tremendous amount of used memory for no advantage when you compare this technique to techniques like bone animation.

With bone animation, each hierarchy can be represented in a few bytes, and lots of animation data can be specified using less space than a single character would for its polygons. Therefore, for space reasons alone, bone animation is the better option by far.

For argument's sake, let's say a mesh is composed of 1,000 vertices, where each vertex is 32 bytes (i.e., 12 bytes for the position, 12 for the normal, and 8 for the texture coordinates). This means a single mesh would require 32,000 bytes for 1,000 vertices, which if you were using polygons would be in the 300 range if index geometry is not used. If that model had 100 frames of animation, we would need to store 3,200,000 bytes of data for that one simple character.

Now let's assume we have a hierarchy of bones in bone animation, where each bone is 132 bytes in size (two matrices that total 32 floats, which equals 128 bytes, and a parent ID integer). If the hierarchy has 30 bones, each pose of animation in bone animation would require 3,960 bytes. If there were 100 frames of animation, we would only be using 396,000 bytes for the character's animation, instead of the over 3 million bytes if we used key-frame data. This number would jump dramatically as the character's polygon count increased, but even then, the amount of bytes used to store its animation data would not increase for bone animation since the number of bones used to specify the animation would not change just because the polygon count increased. It only increases for the key-frame animation technique. We'll discuss the other benefits of bone animation later in this chapter.

So let's do the math using the same theoretical numbers if the model had 10,000 vertices instead of 1,000. This would mean each model takes up 320,000 bytes, and the amount of memory to store all 100 animation poses would equal 32,000,000 bytes. For bone animation, since the number of bones would not increase, this same model would only require 360,000 bytes instead of 32 million. That is a huge difference. Now image you have a game level with 20 character models that need to be animated in this manner. If the characters visually perform the same animation, you can use the same animation data and hierarchy for all models in bone animation. This means you load the animation data once and can apply it to any model that has a compatible skeleton hierarchy. In our simple

example the animation data would not increase even if there were 100 characters that were animated in our scene.

From a programming standpoint, animating a model using key-frame data is fairly simple when compared to bone animation because essentially we are just performing linear interpolation between vertices, a technique we used for the Animation Paths demo earlier in this chapter. Unfortunately for the key-frame animation crowd, that is probably the only advantage it has over bone animation.

You can specify the bones in a hierarchy in key-frame poses and interpolate between them. The interpolation used in this case is usually spherical interpolation, not linear interpolation. However, since physics and collisions can be easily applied to bones to create dynamic animations, the key-frames are used almost as a template for how the animation should behave while the forces acting on it further modify it in real time.

BONE ANIMATIONS

Animations in games are essential to creating the simulations we all see and enjoy. In recent years many games have used a technique known as bone or skeleton animation. Bone animation allows programmers to animate a model using a hierarchy of nodes known as bones or joints. Hierarchies of connected bones or joints form a skeleton. These skeletons are attached to a mesh and are animated realistically in 3D modeling and animation packages. The results are saved to a file and loaded by the game at run-time.

In the past developers used key-frame animation, as previously discussed, for characters and other objects. To do this the developers needed to store a copy of an entire mesh for every frame of animation that made up everything the object can do. When you have to precalculate object positions like that, is very hard to have an object realistically interact with the world or with any physics. Many gamers can still remember a time when they could kill a game character and the character would fall through walls and objects around it. For example, if a player shot a character near a wall and it fell, half of the body could be on the floor while the other half was through the wall. This is very unrealistic, and it takes up a lot of memory to store that many copies of a mesh. Remember that a character in a game has to have animations for every action or interaction you want supported using key-frame animation. Add this to the fact that you can have many different objects in a game, and you quickly find yourself running low on memory just to unrealistically animate characters in your game.

With bone animation the animations are calculated at run-time rather than preprocessed. This allows programmers to apply physics and collisions to the bones along with other forces that can dynamically affect the animation playback. With bone animation, programmers can take that example of shooting a character near a wall and fix the problem by taking into account environment collisions. If the character starts to fall and collides with the wall, the model will stop and slide down the wall realistically. Also, if a character is shot, it is not necessary to create animation data for it. What is needed is to apply some physics forces to the model and let "virtual nature" take its course.

Games can also have objects interacting with other objects. For example, if the player shoots a character whose body falls on top of another character, the results would be much more realistic with some collisions and physics applied than if you did not use bone animation. There would be no way to stack bodies on top of each other realistically using key-frame animation.

With bone animation we are not actually animating the geometry data when we create our character animations. What we are doing instead is animating the bones, which basically are nothing more then a list of matrices and other values. When we alter our bones, we then apply those final matrices to the vertices we've "attached" to those bones. This alters the position of the model's geometry and gives us animation. Since bones are small in memory and can each be applied to a large number of vertices, we can apply things like physics to them that allow us to have more realistic movements in our games. This simply isn't practical with a key-framed mesh using the old-school animation techniques for characters.

A bone in bone animation can be thought of as an extended matrix. A bone can have a lot of information in it that is specific to the developer's needs, but at the basic level a bone is made up of a parent ID so that the code can access the parent bone and two matrices. The two matrices of a bone are the relative and absolute matrices.

The absolute matrix is the final matrix of a bone and is calculated by multiplying the relative matrix of the bone with the absolute matrix of the bone's parent. The relative matrix of a bone is what positions the bone individually. For example, if you rotate a bone 30 degrees, the relative matrix will store that result. However, since the matrix is attached to a hierarchy, when we use the bone we must apply the parent's final matrix with the child's (current bone) relative matrix so that the parent bone can affect the child. Think of rotating your arm. If you rotate your arm, your hand also moves even though your hand is not being moved, relatively speaking. Your hand's relative matrix is not being changed, but the hand's parent bone, the arm, is being altered. Because of this, the mesh we call our hand is being moved around in space when we move our arm, but the hand itself, when talking about just the hand's relative position, is not moving at all. This is the difference between relative and absolute matrices. The absolute matrix takes into consideration the parent bone, and the relative matrix just deals with itself.

SUMMARY

Animations are a very important part of 3D video games. Animations are as much of an artistic as a programming problem and undertaking. Like textures, animations require talented artists on the team to create visuals comparable to what is seen in professional commercial games.

The following elements were discussed in this chapter.

- Line paths
- Curve paths
- Key-frame animations
- Bone and skeleton animations

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. What is the benefit of using time-based calculations over frame-based ones?

2. Describe linear interpolation.

3. Write down the equation for linear interpolation.

4. Write down the equation for cubic Bezier curves as described in this chapter.

5. What are key-frame animations when it comes to character animation?

6. Describe bone and skeleton animation.

7. What benefits does bone animation have over key-frame animation?

8. A series or collection of paths is known as what?

9. A series or collection of routes is known as what?

10. What is the difference between the absolute and relative matrices in bone animation?

CHAPTER EXERCISES

Exercise 1: Build off of the Animation Paths demo and add five more objects to the scene, each with their own route.

Exercise 2: Build off of Exercise 1 and allow the camera to have its own route for the camera's position and look-at position.

Exercise 3: Build off of Exercise 2 and create a cut-scene class. In the class specify a list of routes and create a function that allows the route information to be loaded from a file. The specifics of the file format are up to you. Allow each object to obtain its route position by index. This will mark the start of a simple cut-scene system.

13. LIGHTING

In This Chapter

- Overview of Lighting
- Light Types
- Basic Lighting Information
- Implementing Per-Pixel Lighting
- Additional Lighting Topics

Computer graphics have come along way over the past decade. Many games today place a lot of importance on art style and realism. As time progresses, so do the expectations of the graphics seen in top-of-the-line video games. Games will continue pushing the envelope, and there does not seem to be an end in sight. Although we are exiting the "graphics arms race" of the previous generation when many developers focused on creating visually impressive games instead of fun games, graphics is and will remain (at least in the foreseeable future) an important part of any commercial 3D video game. Today games don't sell just because they are visually impressive; many other factors contribute to a game's success.

Part of cutting-edge computer graphics are lighting and shadows. In this chapter we will briefly discuss lighting and shadows in video games. The topic can become quite mathematically expensive and is more suitable for more advanced books on computer graphics.

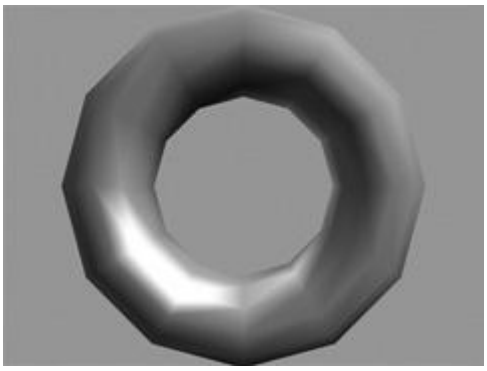
OVERVIEW OF LIGHTING

Lighting in computer graphics involves algorithms that for shading surfaces in a scene to lighten or darken the colors that are rendered based on some set of attributes. Lighting in video game graphics is usually evaluated on a per-vertex or per-pixel level. In the following section we will discuss these two methods in more detail as they relate to lighting.

PROS AND CONS OF PER-VERTEX LIGHTING

Per-vertex lighting essentially means executing a lighting algorithm on each vertex of a piece of geometry. The resulting colors are usually interpolated across the surface during shading. An example of per-vertex lighting is shown in Figure 13.1.

FIGURE 13.1. AN EXAMPLE OF AN OBJECT LIT BY PER-VERTEX LIGHT.



Per-vertex lighting has some good and some bad qualities. From a programming point of view, the complexity of using an algorithm per vertex rather than per pixel is not increased or decreased, thanks mostly to the current nature of shading technology and languages. If anything, per-vertex lighting could be faster than per-pixel lighting if the algorithm executes fewer times in a frame, because fewer vertices are being processed than pixels in per-pixel lighting, not to mention other issues such as fill rate that can affect performance. The downside to using per-vertex lighting includes some of the following.

- Depending on the object's topology, the quality of per-vertex lighting can be less than that of per-pixel lighting.

- Increasing lighting quality using a per-vertex approach usually requires an increase in polygon count, which can lead to performance side effects such as the need for increased and polished results.

- Per-pixel lighting has various extensions that allow for the simulation of lots of detail without the actual detail being present, while retaining a positive performance and frame rate.

Since the lighting algorithm is evaluated on the per-vertex level, the quality of the rendered surfaces depends on the polygon count of objects. Therefore, theoretically, the closer a polygon is to the size of a pixel, the better the results will be. Making many small polygons to get small details and quality is not often acceptable in video games, however. Increasing the polygon count introduces a host of issues such as increasing geometry bandwidth, increasing the possibility of over-draw many times (i.e., drawing lots of polygons on top of each other unnecessarily), and the dramatic increase of extra data, which can create all types of problems with the application's performance.

On today's hardware, simulating detail using a pixel approach is more efficient than using a vertex approach. A case in point can be seen with the normal mapping technique used to simulate high-polygon geometry using low polygon numbers. This is being done in many games such as Gears of War 2.

Take, for example, a wall defined by four vertices. If this wall's surface takes up a large portion of the rendering canvas, the lack of lighting evaluations will cause the surface to have an unrealistic look, especially when the lights and cameras change orientation in relation to one another. In other words, the change in color for one or two vertices can cause a color shift across large portions of the surface that can occur faster than what looks believable. If the color of one vertex of this wall changes, a huge portion of the wall's color will instantly change awkwardly.

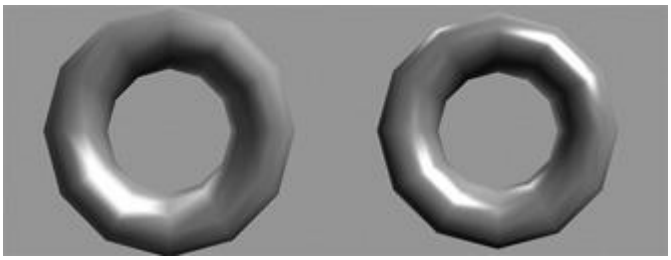
With more lighting evaluations, the lighting simulation can appear more consistent and accurate (relatively speaking). And since we cannot render anything less than a pixel, what else can be better to use to evaluate lighting than to do it on the per-pixel level?

Per-vertex light is lighting that is performed in a vertex shader or manually on the vertices of objects.

PER-PIXEL LIGHTING

Real-time per-pixel lighting is commonly performed within the pixel shader. By executing the lighting algorithm on the pixel level, the polygon count in terms of lighting quality becomes irrelevant. Of course, polygon count still matters in terms of the curvature and symmetry of the geometry. An example of an object lit by per-pixel lighting is shown in Figure 13.2.

FIGURE 13.2. AN EXAMPLE OF PER-VERTEX (LEFT) AND PER-PIXEL (RIGHT) LIGHT.



LIGHT TYPES

In computer graphics, different types of lights can be rendered, with each type altering the lighting algorithm to create a specific type of effect. In this section we will briefly discuss the following light types.

- Directional
- Point
- Spot
- Area

DIRECTIONAL LIGHTS

A directional light in computer graphics appears to come from a direction but has no specific point of origin. Directional lights are the types of lights used to simulate light coming from far away. This type of lighting can be useful for situations where light does not decrease over distance and retains intensity anywhere in the scene. Figures 13.1 and 13.2 use this type of light source.

POINT LIGHTS

A point light emits from a point in space, where the light decreases in intensity over distance. This type of light can be a light bulb, candle, light from a TV screen, and so forth. In computer graphics, a point light usually emits light equally across a radius in all directions, while a directional light emits light in a specific direction with no fall-off in intensity with distance.

Light decreasing over distance is known as light attenuation, and it is the property of a point light that creates the point light affect. In other words, if you take a directional light and apply attenuation, you can create the point light effect.

SPOT LIGHTS

A spot light is essentially a point light that is restricted to a direction, often in the shape of a cone, instead of shining in all directions. In real life we use objects to block light's ability to shine in certain directions. In computer graphics various mathematical equations can be used to simulate the spot light effect.

AREA LIGHTS

An area light in computer graphics is an array of lights that collectively cover an area. Areas lights are commonly used to produce soft shadows in a scene by allowing the various surfaces of a scene to be sampled not only by more than one light but also by slightly varying the positions of these lights. This causes the surfaces to be rendered in a way that softens sharp shadows since the lighting contribution for the discreet points throughout an area light source affect those dark areas. In other words, the shadows themselves receive light that gradually lightens them up, giving a soft shadow appearance.

Therefore, if you use a large array of many point lights, the shadows in the scene will receive light from some of the area light points more than others, creating the softening effect. This occurs because single light sources shade pixels so that those pixels are either in light or in shadow, which creates hard shadows. By creating an area of lights and by accumulating the results, the "in shadow" or "in light" question becomes what percentage is in light versus in shadow, which creates the varying shades of gray necessary for soft shadows.

In real life, light bounces around the environment so many times that shadows are soft because the shadows are not actually the result of no light at all, but instead are the result of some surfaces not being lit as much as others. Light bouncing around the scene falls under the topic of global illumination, which is a highly advanced computer graphics topic. Area lights do not bounce off of surfaces and therefore are not considered global illumination. Area lights, basically, are just a lot of lights covering an area.

Area lights are used more in ray-traced scenes than in video game scenes since the number of light sources in an area light necessary to create believable soft shadows can be far more expensive than what is reasonable for a game's real-time requirements. Lighting can become expensive, and area lights can be very costly.

BASIC LIGHTING INFORMATION

A lighting algorithm is the specific mathematical operation performed on a surface to shade it to produce the desired lighting effect. Lighting algorithms can be used to represent the different values of the lighting equation. Commonly this includes, but is not limited to, the following lighting values.

- Ambient
- Diffuse
- Specular

A few lighting algorithms exist in computer graphics. Later in this chapter we'll discuss a very popular diffuse and specular lighting algorithm known as Blinn-Phong.

The lighting equation itself really depends on what attributes you want to include. Many of the lighting terms such as the visibility value, the geometric value, and so on are better suited for advanced graphics books and are terms you'll encounter when talking about topics such as global illumination. In this chapter we'll briefly discuss a few of the common terms that you'll encounter when you begin researching lighting in computer graphics.

AMBIENT LIGHT

The ambient value of the lighting equation is used for various purposes. In simple lighting algorithms, it is often used to add a fill color to lighten the scene. This fill color is a way to

simulate basic area light by using nothing more than a single color value. This does not lead to realistic lighting conditions and is just a value used to brighten the scene.

On the other hand, the ambient value can be used as the occluding factor. This means the ambient value is used to store a percentage of how many surfaces in the scene can possibly block light rays from reaching the surface. This technique is known as ambient occlusion.

In ambient occlusion you essentially trace many rays from the surface being evaluated into the scene. The percentage of those rays that hit some surface in the scene is recorded and used as the ambient occlusion value. This value is multiplied by the other lighting terms to create shadows in the scene. These shadows appear soft when many rays going in many different directions from the surface are used. This is because some points on a surface might be lit differently than others, creating a smooth gradient of gray values across the surface.

Regardless of whether the algorithm used involves nothing more than adding a static color value to the scene or is as complicated as ambient occlusion, the ambient value itself is generally used to simulate light coming to the surface from the environment. Global illumination algorithms go a step further to account for light bouncing off of a surface and evaluate the resulting color bleed onto the surface. An example of this is to take a bright blue ball and place it on a plain sheet of paper. If you place a bright light on top of the ball so that the light is shinning down on it, the white piece of paper should have some tints of blue on it because the blue reflecting off of the ball is reaching the paper. This color bleeding effect is common in global illumination.

DIFFUSE LIGHT

The diffuse lighting value is light that has reached a surface and has reflected off of it evenly, or at least it seems even. When light hits a surface, it is scattered back into the scene. The smoother the surface is, the more light reflects in the direction from which it came. Surfaces such as mirrors are extremely smooth, and lights reflect back so that a mirror image appears on the surface. For surfaces that are not as smooth, light often hits microscopic grooves along the surface that cause the light to scatter in different directions. Since so much light is hitting such a surface from so many different angles, diffusely lit surfaces look as if light is being evenly reflected in all directions.

SPECULAR LIGHT

Specular light is light that hits a smooth surface and reflects sharply in specific directions. With diffuse light, the millions of light rays in nature strike the surface and scatter in so many different directions that objects look evenly lit regardless of the viewing angle as far as light reflection is concerned.

With specular light the sharp reflection of light, instead of the scattering reflection, in many angles is what creates a highlight on the surface. Mirrors are so smooth that the light can create clear reflections on them. For objects that are smooth enough to reflect light more sharply than others, highlights appear that can be seen from certain orientations. Since light reflects more sharply in some directions than others, the appearance of the highlight can change depending on your viewing angle. This differs from diffuse light, because a diffusely lit object looks the same regardless of the viewing angle (aside from shadows, but we'll assume shadows are not present), but an object reflecting specular highlights looks different as the object's orientation or the viewing orientation changes. This is similar to rotating a plastic soda bottle while standing in sunlight. The highlights from the light hitting the object shimmer and change as the orientation changes since the reflection isn't evenly dispersed in all directions.

LAMBERT DIFFUSE MODEL

The Lambertian reflection model is used to simulate light rays striking a surface and reflecting back into the scene, where the brightness of a point on that surface looks the same regardless of where the observer is located. In other words, Lambertian reflection is ideal for diffuse light, where the nature of diffuse light is to illuminate a surface so that it looks the same around all angles.

The Lambert equation for diffuse light is fairly straightforward and very popular. To calculate the diffuse contribution, you need the light vector and the surface normal. With these two pieces of information you can calculate the dot product between them, and the resulting floating-point value is what you use as the diffuse contribution. This is commonly referred to as N dot L, where N is the normal and L is the light vector. You take this dot product value and multiply it by the surface color, material color (e.g., textures etc.), and so forth.

BLINN-PHONG SPECULAR MODEL

The Blinn-Phong reflection model is used to perform real-time lighting in computer graphics. The Blinn-Phong reflection model is actually a modification of the Phong reflection model. A reflection model can be thought of as an algorithm that is used to describe how light reflects off of objects.

The Blinn-Phong reflection model is used to create the specular contribution of the lighting equation. The steps to create the specular contribution using the Blinn-Phong model are as follows.

1. Retrieve the normal vector.

2. Compute the light vector as the light's position minus the vertex position and normalize it.

3. Compute the view vector as the light's position minus the camera's position and normalize it.

4. Compute the half vector that is necessary for the Blinn-Phong algorithm as the light vector plus the view vector.

5. The specular value is the dot product of the normal and the half vector raised to a specific power (this power is known as the specular power and is used for shininess).

To use the Blinn-Phong reflection model, the first pieces of data necessary are the normal, the view vector, and the light vector. The normal is the surface normal. The view vector is the vector that describes the direction from the camera's position to the point being shaded, which allows the specular contribution to be view-based. The last vector, the light vector, is a vector from the light's position to the point being shaded. If these vectors are calculated in the vertex shader and sent to the pixel shader, the results can be interpolated to point not from the original vertex position but from the pixel's point being shaded. You could also calculate them in the pixel shader, but the interpolated values from the vertex shader work just as well.

Once you have these vectors, you calculate a new vector called the half vector. This vector is used instead of finding the reflection vector as is done in the Phong reflection model. It is faster to calculate than the method used in the Phong reflection model. Like the diffuse contribution that is calculated using N dot L, the specular value is found using N dot H, where N is the normal and H is the half vector. Raising this value by a power, known as the specular power, allows us to control the amount of shine an object has.

We'll use the Lambert diffuse and Blinn-Phong reflection models in the demo application later in this chapter.

MATERIALS

Material is a term used to describe the representation of a surface. For example, a brick wall in a video game might have as part of the "brick" material a texture image of bricks, a normal map used for bump mapping to increase the detail of the bricks, a diffuse modifier, and anything else the artist creates to make the wall look like a brick wall.

These days a material is a collection of pieces of data used to create the look of a surface. In the early days of 3D graphics, most games used textures as the main or sole source of the material. Materials can include, but are not limited to, the following.

- Decal color texture map
- Normal map
- Alpha map
- Diffuse color
- Specular color
- Emissive color
- Ambient color
- Vertex shader
- Pixel shader
- Geometry shader

In today's games, materials are complex assets, and some game engines have their own material systems built into the rendering system. With the amount of content that goes into games, materials will most likely continue to grow in complexity as the amount of data necessary to represent certain surfaces increases.

IMPLEMENTING PER-PIXEL LIGHTING

On the companion CD-ROM, in the Chapter 13 folder, is a demo application called Lighting. The Lighting demo uses the Lambertian reflection model for diffuse reflection and the Blinn-Phong reflectance model for specular highlights. The demo builds off of the source code from the OBJ Models demo in Chapter 11, "3D Models." In this section we will discuss the code added to modify the OBJ Models demo to display lighting on an object.

CREATING THE SHADERS

The HLSL code for the effects is stored in LambertBlinnPhong.fx. In the vertex shader, the code starts off by transforming the vertex position as usual, and then it moves on to compute the transformed normal vector to account for the object rotating. It computes the light vector, which is computed by normalizing the light position from the vertex position,

and the view vector, which is the normalized difference of the camera position and the vertex position. The vectors are interpolated across the surface as they are passed to the pixel shader from the vertex shader, which is the same as if we calculated the vectors in the pixel shader using the computed pixel position. We hold off on normalizing until we reach the pixel shader stage since the interpolation could possibly de-normalize the vectors, and we'd have to re-normalize anyway just to keep this from happening.

In the pixel shader all vectors are normalized, and the half vector is computed for the Blinn-Phong algorithm. Once the vectors are normalized, the diffuse contribution is computed by calculating N dot L, and the specular contribution is computed from N dot H raised to a power that represents the shininess factor, which in this demo is 25. Since this demo is just a simple example of lighting, the diffuse and specular values are multiplied by the color white and stored as the output color. We could have multiplied them by a specific object color, light color, texture color, and so forth.

The `LambertBlinnPhong.fx` shader is shown in Listing 13.1.

LISTING 13.1. THE LAMBERTBLINNPHONG.FX SHADER

```
float4 lightPos;
float4 eyePos;


DepthStencilState DepthStencilInfo
{
   DepthEnable = true;
   DepthWriteMask = ALL;
   DepthFunc = Less;
};


cbuffer cbChangesEveryFrame
{
   matrix World;
   matrix View;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};


struct VS_INPUT
{
   float4 Pos : POSITION;
   float3 Norm : NORMAL;
   float2 Tex : TEXCOORD;
};

struct PS_INPUT
{
   float4 Pos : SV_POSITION;
   float3 Norm : NORMAL;
```

```
    float3 LightVec : TEXCOORD0;
    float3 ViewVec : TEXCOORD1;
};


PS_INPUT VS(VS_INPUT input)
{
    PS_INPUT output = (PS_INPUT)0;

    float4 Pos = mul(input.Pos, World);
    Pos = mul(Pos, View);
    output.Pos = mul(Pos, Projection);

    output.Norm = mul(input.Norm, World);
    output.Norm = mul(output.Norm, View);

    output.LightVec = lightPos.xyz - Pos.xyz;
    output.ViewVec = eyePos.xyz - Pos.xyz;

    return output;
}


float4 PS(PS_INPUT input) : SV_Target
{
    float3 normal = normalize(input.Norm);
    float3 lightVec = normalize(input.LightVec);
    float3 viewVec = normalize(input.ViewVec);
    float3 halfVec = normalize(lightVec + viewVec);
    float diffuse = saturate(dot(normal, lightVec));
    float specular = pow(saturate(dot(normal, halfVec)), 25);

    float4 white = float4(1, 1, 1, 1);

    return white * diffuse + white * specular;
}


technique10 BlinnPhongSpecular
{
    pass P0
    {
        SetDepthStencilState(DepthStencilInfo, 0);

        SetVertexShader(CompileShader(vs_4_0, VS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, PS()));
    }
}
```

THE MAIN SOURCE FILE

In the main source file, two new objects are added to the global section: one to pass the light position to the shaders and one to pass the camera position. The end of the global section with the new variables in the Lighting demo are shown in Listing 13.2. In the `InitializeDemo()` function, these effect variables are obtained so we can use them to actually pass data to the shaders. The `InitializeDemo()` function is partially shown in Listing 13.3. The only new code in it is the effect variable calls that obtain access so we can set the light and camera position.

LISTING 13.2. PARTIAL LOOK AT THE LIGHTING DEMO'S GLOBAL VARIABLES

```
…

ID3D10Effect *g_shader = NULL;
ID3D10EffectTechnique *g_lightingTech = NULL;
ID3D10EffectVectorVariable *g_lightPosEffectVar = NULL;
ID3D10EffectVectorVariable *g_eyePosEffectVar = NULL;
ID3D10EffectMatrixVariable *g_worldEffectVar = NULL;
ID3D10EffectMatrixVariable *g_viewEffectVar = NULL;
ID3D10EffectMatrixVariable *g_projEffectVar = NULL;

D3DXMATRIX g_worldMat, g_viewMat, g_projMat;


// Scene rotations.
float g_xRot = 0.0f;
float g_yRot = 0.0f;
```

LISTING 13.3. PARTIAL LOOK AT THE `INITIALIZEDEMO()` FUNCTION

```
bool InitializeDemo()
{
  // Load the shader.

  DWORD shaderFlags = D3D10_SHADER_ENABLE_STRICTNESS;

#if defined( DEBUG ) || defined( _DEBUG )
   shaderFlags |= D3D10_SHADER_DEBUG;
#endif

   ID3D10Blob *errors = NULL;

   HRESULT hr =
D3DX10CreateEffectFromFile("LambertBlinnPhong.fx",
      NULL, NULL, "fx_4_0", shaderFlags, 0, g_d3dDevice, NULL,
      NULL, &g_shader, &errors, NULL);

   if(errors != NULL)
   {
      MessageBox(NULL, (LPCSTR)errors->GetBufferPointer(),
                 "Error in Shader!", MB_OK);

      errors->Release()
   }
```

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

    g_lightingTech = g_shader->GetTechniqueByName(
        "BlinnPhongSpecular");

    g_worldEffectVar = g_shader->GetVariableByName(
        "World")->AsMatrix();

    g_viewEffectVar = g_shader->GetVariableByName(
        "View")->AsMatrix();

    g_projEffectVar = g_shader->GetVariableByName(
        "Projection")->AsMatrix();

    g_lightPosEffectVar = g_shader->GetVariableByName(
        "lightPos")->AsVector();

    g_eyePosEffectVar = g_shader->GetVariableByName(
        "eyePos")->AsVector();

    …

    return true;
}
```

In the `Update()` function the world matrix is rotated along the X and Y axes so we can see the specular lighting changes as the orientation moves in real time. In the `RenderScene()` function the only new code is the code that passes the light and camera positions to the shaders. The light position is located 5 units along the negative Z axis, while the camera position is located at the origin. The object itself is positioned 6 units along the Z axis, which can be seen in the `Update()` function.

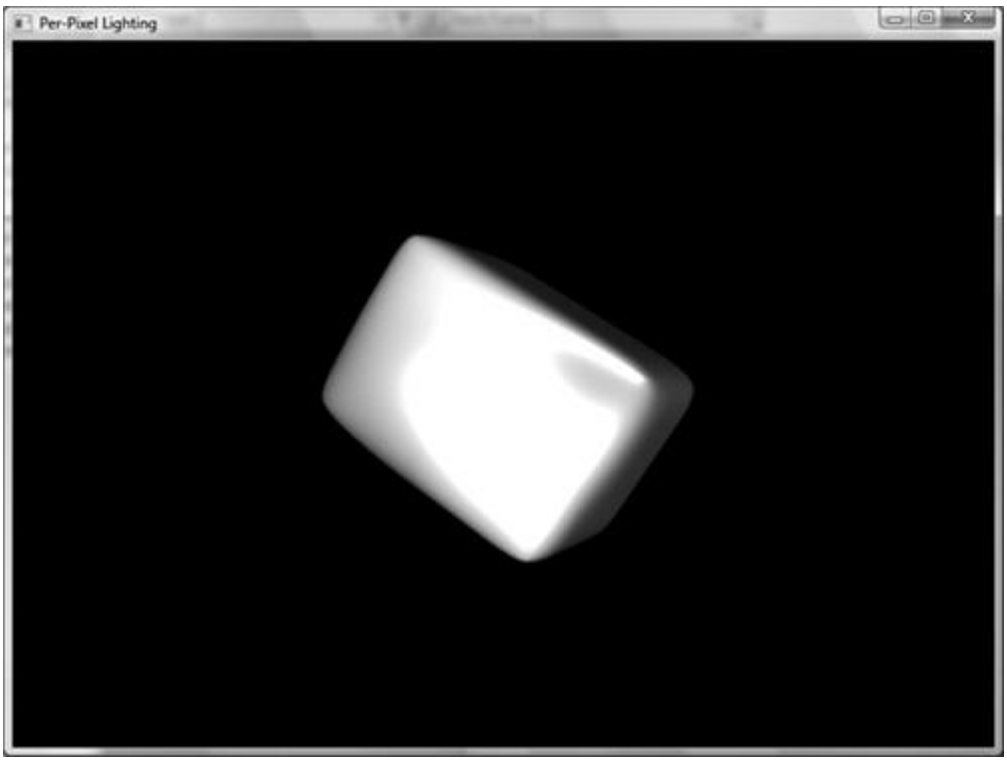
The `Update()` and `RenderScene()` functions are shown in Listing 13.4. Figure 13.3 shows a screenshot from the running demo.

LISTING 13.4. THE `UPDATE()` AND `RENDERSCENE()` FUNCTIONS FROM THE LIGHTING DEMO

```
void Update()
{
    g_xRot += 0.0001f;
    g_yRot += 0.0002f;

    if(g_xRot < 0) g_xRot = 359;
    else if(g_xRot >= 360) g_xRot = 0;

    if(g_yRot < 0) g_yRot = 359;
    else if(g_yRot >= 360) g_yRot = 0;

    D3DXMATRIX trans, rotX, rotY;
```

```
   D3DXMatrixRotationX(&rotX, g_xRot);
   D3DXMatrixRotationY(&rotY, g_yRot);
   D3DXMatrixTranslation(&trans, 0, 0, 6);

   g_worldMat = (rotX * rotY) * trans;
}


void RenderScene()
{
   float col[4] = { 0, 0, 0, 1 };

   g_d3dDevice->ClearRenderTargetView(g_renderTargetView, col);
   g_d3dDevice->ClearDepthStencilView(g_depthStencilView,
                                      D3D10_CLEAR_DEPTH, 1.0f, 0);

   g_d3dDevice->IASetInputLayout(g_layout);
   g_d3dDevice->IASetPrimitiveTopology(
      D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

   D3D10_TECHNIQUE_DESC techDesc;
   g_lightingTech->GetDesc(&techDesc);

   unsigned int stride = sizeof(DX10Vertex);
   unsigned int offset = 0;

   float lightPos[4] = { 0, 0, -5, 1 };
   float eyePos[4] = { 0, 0, 0, 1 };

   g_worldEffectVar->SetMatrix((float*)&g_worldMat);
   g_lightPosEffectVar->SetFloatVector(lightPos);
   g_eyePosEffectVar->SetFloatVector(eyePos);

   for(int m = 0; m < (int)g_meshes.size(); m++)
   {
      g_d3dDevice->IASetVertexBuffers(0, 1,
         &g_meshes[m].m_vertices, &stride, &offset);

      for(unsigned int i = 0; i < techDesc.Passes; i++)
      {
         g_lightingTech->GetPassByIndex(i)->Apply(0);
         g_d3dDevice->Draw(g_meshes[m].m_totalVerts, 0);
      }
   }

   g_swapChain->Present(0, 0);

   Update();
}
```

FIGURE 13.3. SCREENSHOT FROM THE LIGHTING DEMO.



ADDITIONAL LIGHTING TOPICS

A few additional lighting-related topics are important to video game graphics programming. In this section we'll briefly discuss the following topics and their roles in game graphics.

- Bump mapping
- Light mapping
- Shadows

BUMP MAPPING

Bump and normal mapping are very popular techniques that can be seen in many of today's video games. Their goal is to simulate fine detail on surfaces that are actually flat. In other words, it is a way to simulate detail where that detail does not exist, similar to the original purpose of texture mapping.

A bump map is a texture where a height map is used to represent the pixel-level depth of the detail. This height map is converted into an image of normal vectors, where the normals can vary from pixel to pixel. When lighting a surface using this per-pixel normal data from a bump map image, you can slightly adjust the shading of a surface to give the impression of depth and detail even though it is really just a lighting trick used to simulate that detail.

Bump and normal mapping are essentially clever extensions to per-pixel lighting, where instead of using the interpolated normal of vectors across a surface like we did for the Lighting demo, we fetch the normals from a texture known as a bump or normal map. The terms *bump map* and *normal map* are sometimes used to mean the same thing. These days the term *normal map* is used more often. Although the terms are sometimes used to refer to the same thing, how an image is created determines if it is a bump or a normal map.

Bump maps are usually created from height maps (i.e., grayscale images of a texture or pattern) that are converted to an image of normal vectors, where the X, Y, and Z of the normal are stored in the R, G, and B channels of the image. A normal map is different in that a normal map image is created by examining the difference between a low-polygonal object and a high-polygonal object and capturing its curvature information. When you apply the normal map to the low-polygon object, it can give the appearance of having high detail like the high-polygonal object, even though it's just a simulation and that detail isn't there. It's like taking an object with 1 million polygons and a lower-resolution version with just 10,000. As long as the basic shape and curvature are similar—in other words, you are not trying to normal map a sphere on a box—you can make the low-polygon object look like the high-polygon one. This has been made popular by video games such as *Crysis, Gears of War 2*, and many other titles.

Essentially, the difference between bump and normal maps is how they are computed. The shaders themselves are the same, and usually a texture known as texture-space (also known as tangent-space) bump or normal mapping is used to render the surface consistently. A bump map is usually created from another image such as a height map, while a normal map is created using two geometric objects. The goal for both is to use those pixel-level normal vectors to create the illusion of detail across the surface so that the detail does not actually have to be there, which can lead to performance gains. Like with the example of the 1–million-polygon object versus the 10,000-polygon object, if you can capture that detail and display it using less while making it look exactly or almost the same, the performance gains will be enormous.

Sometimes the terms bump mapping *and* normal mapping *are used interchangeably.*

LIGHT MAPPING

Light mapping is a technique used to display lighting in a scene that is not calculated at run-time but is precomputed and displayed during execution. The idea behind light mapping is to use textures to represent the illumination of a surface. That way lighting can be displayed in a scene by using textures instead of by calculating the actual illumination. What is important about this is that it allows complex and extremely expensive lighting and shadowing algorithms to be precomputed and displayed at run-time at a performance acceptable for games.

Light mapping is a way to display static lighting in a scene using algorithms that are, at this time, either impossible or impractical to do in real time. This allows for very realistic renderings without a performance hit. The main downfall to this method is that the scene's geometry and lights that are light mapped are static since they are precomputed and can't change in position or orientation without being computed again. Light mapping is a great way to display global illumination in a video game's scenes.

SHADOWS

Shadows are created in nature by a lack of light striking some surfaces in relation to others. In computer graphics, shadows do not come for free. It is not enough to shine lights in a scene to develop shadows in graphics. To create shadows in computer graphics we must account for the visibility value of the lighting equation. This allows us to determine how much light reaches a surface by testing the visibility of a point on a surface to each light source in the scene. In other words, we must take steps to calculate shadows because

shadows themselves are independent of the lighting algorithms discussed. They are their own effects. Just because a scene has lighting does not mean it automatically has shadows.

The following are the most common shadowing topics you'll likely encounter in computer graphics.

- Faking shadows
- Shadow volumes
- Shadow mapping
- Soft shadows
- Global illumination

Shadows can be faked in a number of ways. Imposters, which are stand-in geometry (for shadows they are often solid-color pieces of geometry that follow the character), can be used to represent shadows or can be painted by artists into textures. Faking shadows is computationally cheaper because shadowing algorithms are usually more expensive. The problem with fake shadows is that they are not realistic and are not ideal in most cases.

Shadow volumes are a shadowing technique used in games such as Id Software's *Doom 3*. The main concept behind shadow volumes is that a volume of geometry is created by extruding polygons around the silhouette of an object in a direction dictated by the position of a light. In other words, the geometry is extruded in the direction of the light vector. A shadow volume is then rendered to graphics buffers such as the stencil buffer of the graphics API and used as a rendering mask to determine which areas of the scene are in light or shadow. Pixels in shadow are rendered in the shadow color, while the rest of the screen is rendered normally using lighting. By rendering the shadow volume to the stencil buffer, the pixels that make up that buffer can be used to determine where the shadows exist.

Shadow mapping is a shadowing technique that works by rendering the scene from the light's perspective and storing only depth values into a texture that is referred to as a shadow map. During the rendering of the scene, the depths from the shadow map are projected onto the scene. The depth of each rendered pixel is compared to the projected shadow map depth to see if there is a surface between the one being rendered and the light source. If the test passes and the depth from the shadow map is closer than the depth of the pixel being rendered, then a shadow color is used; otherwise, rendering proceeds as usual.

Soft shadows are a style of shadow, not a specific technique or algorithm. Soft shadows are not hard. In other words, soft shadows often display a gradient of gray values from light to dark across the shadow's surface. Soft shadows are often light, subtle, and blurry. To create soft shadows you can fake it by blurring the shadow map, for example, or you can go through the extra processing to create more dynamic soft shadows. One way this is done is by calculating the penumbra and umbra that make up the hard and soft areas of the shadow and using that data for rendering. Some algorithms use these two pieces of data in shadow mapping, shadow volumes, and so forth to create soft shadows.

The last item on the list is global illumination. Global illumination is the act of representing illumination in a scene as it interacts with the environment on a global level. This includes surfaces being lit by light sources, surfaces being lit by light bouncing off of other surfaces (sometimes leading to color bleeding, as mentioned earlier in this chapter), and so forth. Global illumination techniques and algorithms produce some of the most believable scenes in computer graphics because they attempt to account for much of what we see in nature. Global illumination is usually stored in static images such as light maps since global

illumination in real time is not suitable for video games at this time. (Global illumination is not calculated in real-time because it can take minutes, hours, or days to calculate scenes, whereas in a video game it would have to take milliseconds at the most. It takes too long to calculate global illumination for it to be done on-the-fly in real-time.)

SUMMARY

Lighting and shadows are very important in computer graphics, especially in terms of adding realism. In this chapter you've received a brief overview of real-time lighting in Direct3D. As you begin to research more advanced topics in computer graphics, you will undoubtedly encounter various other lighting and shadowing topics.

The following were discussed in this chapter:

- Per-vertex lighting
- Per-pixel lighting
- Ambient lighting
- Ambient occlusion
- Materials
- Diffuse lighting
- Specular lighting
- Shadows

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. Define per-vertex lighting.
2. Define per-pixel lighting.
3. Describe directional lights in computer graphics.
4. Describe point lights in computer graphics.
5. Describe spot lights in computer graphics.
6. Describe area lights in computer graphics.

7. What is global illumination?

8. What is diffuse light? How does it differ from ambient light?

9. What is specular light? How does it differ from diffuse and ambient light?

10. Describe the Blinn-Phong lighting algorithm.

11. What does the term *material* mean in modern computer graphics?

12. List two ways shadows can be faked in games.

13. Generally describe the shadow volumes technique.

14. Generally describe the shadow mapping algorithm.

15. Describe two ways to produce soft shadows.

CHAPTER EXERCISES

Exercise 1: Build off of the Lighting demo and add the ability to specify the light color through a uniform variable that is supplied by the application.

Exercise 2: Build off of the Lighting demo and add three more lights to the shader.

Exercise 3: Build off of the Lighting demo and render multiple lights using a multipass approach. Use color blending to blend the results into one.

14. CONCLUSIONS

In This Chapter

- Improvements to the Game
- Additional Techniques and Topics
- Moving Forward

Game development is a very challenging and rewarding industry to participate in. From the perspective of a hobbyist or student making video games, making that first game playable brings a great sense of accomplishment and pride. Making any type of game, from the hard-core genres to the casual game types, is never easy, and the completion of any type of game brings a lot more than something that can be shown to your friends. It brings experience, knowledge, and unique insight that is difficult at best to learn from a book. It also feels really good to have something playable that you've created through hardware and dedication.

With every game you create, you are placing yourself in a position that helps you become the professional game developer you've always wanted to be. This is true regardless of the role you play on the development team. If you can design and implement a game from start to finish, even if the game turns out to be a dud, it will never be in vain, as long as you've learned something in the process. The more you learn, the better your chances are for creating a successful title in the future. There is no substitute for experience when it comes to any profession.

It is important to know where your game falls short and to understand all aspects (both positive and negative) of your project. This is very valuable because if you don't see both the good and bad in your games, how do you learn and grow in an industry that is fast moving, fast paced, and dynamic? The goal of this chapter is to briefly look at what you can do to improve this game project or start a new one.

IMPROVEMENTS TO THE GAME

The purpose of this chapter is to look at what we accomplished throughout this book and what you can do after you're done with this book.

GAME-PLAY IDEAS

Placing effort in the game experience is essential to making the game more fun for those playing it. When starting off in game development, the idea is often to create a simple casual game as a way to gain experience in designing and finishing a video game. Therefore, most improvements that you make to your initial ideas will include looking at all aspects of the game and trying to figure out what could be enhanced or added without going overboard, without unintentionally compromising the original design, or without being unnecessary. Some improvements that can be added to the game include:

- Additional controls
- New weapons and items to improve the game-play
- Adding strategy
- Adding difficulty

Not every gamer is the same, and the controls offered can have an impact on how gamers play your game. Giving players the option to customize their controls can be a very useful and straightforward improvement that can be a welcome addition to your game. In Bungie's *Halo 3* for example, the button layouts affect the game-play so much that player tactics can change when using one type rather than the other. It is no wonder that many aggressive and serious players choose *Halo 3*'s bumper jumper control type, which gives them reflex and control possibilities that affect how they approach the game.

New weapons and items can be added to improve the game-play and allow for shifts in the game's momentum and pacing with great effect. Such items can include invisibility, extra weapon damage, health regeneration, and so forth. Nothing that is added should make the game either too easy or unnecessarily difficult, and balance always needs to be taken into consideration in these areas.

When it comes to adding strategy, the game-play will change when a player is given the opportunity to dynamically make decisions in the moment that can affect the pacing of the game. This can include using power-ups wisely, picking up weapons based on their effect on the enemies currently being encountered, taking advantage of obstacles in the game world (e.g., using them for cover from enemy fire), and so forth.

Difficulty can be added to the game by tweaking ammo and health, weapon rate of fire, weapon and enemy accuracy, enemy and player speeds, power-up and item durations and strength, and so forth. This can be done during testing while you observe how the different difficulty levels stack up to the skill and abilities of those playing your game. It is possible that what you find to be difficult is fairly easy for the average gamer or that what you find to be easy is more difficult than you thought it would be for the target audience.

GAME DESIGN IMPROVEMENTS

Making changes to the game's design is probably not the best course of action. You could probably get away with designing either a sequel that implements your design improvements or even an entirely different game, depending on what you have in mind. If you find your game lacking in the fun and interaction department and want to improve it, there are a few things you can do without taking on too much as a beginner.

- **Follow the leader.** Look at what other games of the same genre do well and, if feasible, try to incorporate those things in your game. You can get ideas for how to make the enemies more challenging, balance weapons, intensify the game experience, and so forth.

- **Get feedback for real-world players.** Allow others to play your game and suggest what they would like to have changed or tweaked. Minor tweaks can lead to significant changes in the way people play your game and could contribute to the overall improvement of your project. It is difficult for you to make positive changes to your game without first seeing for yourself how people interact with your game. Sometimes players will experience your game in ways you didn't know were possible. This insight is very important.

- **Properly test your game.** You can discover flaws in your game's design (e.g., level design, weapon design, etc.) that can make or break your game's success. It is important to test your game and fix anything that can break your game. This includes not just bugs but also exploits, unbalanced weapons and equipment, unbalanced level design (important for multiplayer games), and so forth.

If you find yourself making changes that completely change the project, we recommend considering making a new game. If you are making games for fun and for the experience, always be open to creating more games once you've finished one.

ADDING TO THE GAME

Many features can be added to the game. It is important when coming up with new features to not go overboard and try to do more than is reasonable. When working on a game that you are creating for yourself, it is very tempting to continually add or aspire to add features, but doing so can create more negatives than positives. Here are a few things to keep in mind when extending the game you have created.

- If the changes you are making are dramatic, consider making a sequel or a different game altogether. That way you can have multiple items for your resume and can offer what you've learned from each one, which can help a lot during an interview.

- Be mindful of your limits. It is very tempting to create the next *Halo* or the next *Gears of War*, but is it realistic? If you try to do too much too fast, you can end up wasting a lot of time working on a project that is above your skill and experience level.

- Think carefully about what you add or change in your game. If you add features that do not positively affect the game or the experience of those playing it, you may be adding

them at the expense of features that would have made for a stronger product. No game has to do everything all at once, and it often is overkill to attempt such a goal.

- Plan carefully. It is never a good idea to start or continue a project without hard guidelines, goals, and steps that you want to take. Without a solid plan of action, it is easy for the hobbyist or student game developer to enter a loop where there is no defined end to the project, and features are constantly being added for the sake of adding them. Once a project is done, don't be afraid to end it and move on to the next project when it comes to creating games for experience.

A number of other features can be added to the game that would have a positive impact on the project and can also benefit your resume. These additional features are not necessarily the easiest for beginner but can serve as the next step. These can include the following.

- **Multiplayer support.** Many games today are as popular as they are in part or entirely because of their multiplayer support. *Halo 3, Gears of War 2*, and *Call of Duty 5* are a few examples where multiplayer is essential to the game's experience, especially for games that are competitive.

- **Improved use of audio effects.** Audio has always been a very powerful way to immerse a gamer in the experience. Making the most of the audio and sounds in a game can make the game experience more exciting and intense.

- **Improved graphics and interface design.** Graphics have played an important role in video games for a long time and will always be important to gamers. If the game looks good, it could attract the eye of many gamers who would otherwise have missed out on the title. Of course, how far you go with this depends on time, skill, and necessity.

- **Dynamic environments.** These can help add to the intensity and experience of the game-play. This can include background explosions, rocks falling from mountain tops, and so forth.

A lot can be added to a game project, and we recommend that you think about what you would like to see in the game and attempt to implement it without going overboard. Remember, a degree might help in some fields, but experience is the most valuable trait when starting out in the games industry. It could never hurt to be able to demonstrate to potential employers what you can do and what skills you possess, so try to diversify your experience.

ADDITIONAL TECHNIQUES AND TOPICS

Some of these topics are fairly common to almost all games for reasons we will explore next, and their uses and benefits cannot be ignored when creating more complex video games.

In this section we will briefly look at each of these topics and how they can play a role in games in general. These topics include but are not limited to the following.

- Scene management

- Artificial intelligence

- Game physics

- Networking

SCENE MANAGEMENT

Scene management is a general term that often describes steps taken algorithmically to manage the game's data and information in a way that optimizes their use. This can come in a number of forms, and all games include some form of scene management, especially 3D games.

Some of the more common forms of scene management include the following.

- State management
- Level-of-detail
- Game updates
- Optimizing assets
- Geometry culling

State management refers to managing how game states and changes on a per-frame basis occur and usually is a topic dealing with the management of graphics API state switches. In graphics APIs, changing between some assets such as textures and shaders can prove fairly expensive. If these changes occur many times per-frame, this could possibly have an impact on the game's overall frame rate and performance. By switching states only when necessary, a game developer can attempt to avoid some of the costly overhead associated with this action. This can include sorting all objects that need to be rendered by their texture so that all objects with texture A are rendered first, followed by all objects that use texture B, and so forth. Although it is impossible to eliminate all state changes in most complex games, it is possible and worthwhile to look into avoiding them as much as possible.

Level-of-detail refers to displaying versions of an object based on certain conditions in the game world. For example, as objects move further away from the camera, a lot of their detail cannot be seen. If you have a crack on a rock, it is possible that such a small detail cannot be seen across the level, even if the rock itself can be seen. If we continue using camera distance as an example, you can imagine that as objects move away they appear smaller on the screen. Using fewer pixels to make up the object means less detail, and if an object can be rendered using a version of the model that has less polygons, smaller resolution textures, and so forth, then a game can improve performance by processing less data while at the same time being completely transparent to the gamer. If the gamers cannot see detail as objects move away anyhow, they are not likely to notice that the object they are seeing at a distance is not the same model they see up close. Sometimes it is possible to notice the sudden change when high-resolution objects are used in place of lower-resolution ones. This type of geometry switching is very common in major 3D games such as *Assassins Creed* by Ubisoft, where if you look carefully enough from the right elevation, you can tell that the textures and geometry seen in the distance are of a much lower level-of-detail than if you were up close to them.

Game updates refer to executing algorithms that do not necessarily need to be updated as often as a frame-by-frame action such as rendering. One example of this can be seen in artificial intelligence. Depending on the game, it is unlikely that each of the AI characters will need to think for every single frame that is processed. Therefore, one might schedule AI updates to happen every *x* number of frames or after a certain amount of time has passed. Alternatively, maybe the game only updates a few characters each frame until all characters have been updated. The same can be true for physics and collision detection, where the frame-by-frame differences are so small that they don't have to take place each frame for all objects in the game environment.

Optimizing assets is a general term referring to actions such as compressing textures, adjusting texture resolution to smaller sizes wherever possible, optimizing the geometry of a polygonal model (reducing triangle count, using index geometry, etc.), optimizing shader instructions and taking advantage of the hardware wherever possible (e.g., using dynamic branching for early-out execution), and so forth.

Geometry culling is any technique that can determine if an object or section of geometry should or should not be rendered. For example, if the game determines that object A is behind the player and out of view, then the game would not bother passing that object to the graphics hardware since the object will not be rendered. This can prevent unnecessary transfers of data and CPU/GPU processing for objects that have no effect on the final rendered scene. Geometry culling commonly takes one of the following forms.

- Occlusion culling
- View frustum culling
- Geometry partitioning

Occlusion culling is a technique that is used to determine if one object is blocking the view of another object in the scene from the camera's point of view and, if so, discards it from the rendering process. For example, if your scene had two buildings, where one building completely blocked the view of the second (even if both are in the view range), then it would be wasteful to process and attempt to render the second building since it will have no impact on the final rendered scene.

View frustum culling is a technique in which you don't draw any geometry that is not within the view range of the camera. This essentially means rendering any object we can actually see in the direction we are looking, but not those outside that volume. Most 3D games have a lot of data in their scenes, only a section of which can be visible at any given time depending on the position and direction of the camera. No complex 3D scene seen in any major video game would be possible without culling because there is so much information that it would overwhelm even the most powerful of today's computers. This topic along with occlusion culling and geometry partitioning are discussed in the book *Game Graphics Programming*, by this author. The view-frustum culling demo in that book shows that attempting to draw several thousand objects at once can have a huge impact on the demo's performance, but that culling out objects that can't be seen results in an obvious leap in frame rate.

Geometry partitioning is the technique of taking a set of geometry, such as the level's entire geometry, and splitting it up into smaller, more manageable chunks. These chunks can be view frustum culled to avoid drawing sections of the game level that can not be seen. Geometry partitioning is important not only for rendering geometry but also for performing other updates such as physics and collision detection. For physics, if you can determine a small area of the level where an object is located, then the object can be tested for collision on the geometry in that small chunk of the level instead of every piece of geometry, which would not only be very costly but would be impractical with today's hardware and scenes.

Geometry partitioning uses what is known as data structures and algorithms. A data structure is essentially a way data is stored in memory, while an algorithm is an operation that occurs on a data structure. The simplest data structure is the array, where data is stored sequentially in memory, and the simplest algorithm (using arrays as an example) is the insertion algorithm, which places values in the array's elements. Each data structure can have algorithms that are common among other data structures (e.g., insertion, deletion, sorting, searching, etc.) and some that are unique to the specific data structure being used (e.g., hashing keys in a hash table, node traversal in a tree, etc.). Some common data structures used in game development include the following.

- Octrees
- Quad trees
- BSP trees
- Portals
- Scene graphs

Some data structures that are common to general application development include:

- Arrays
- Link lists
- Stacks
- Queues
- Hash tables
- Trees (binary, k-dimensional, etc.)
- Graphs (technically a type of tree)

We recommend that you study various data structures and algorithms, for they can prove quite useful when you are developing complex software applications, especially for video games.

ARTIFICIAL INTELLIGENCE

Artificial intelligence depends greatly on the game that is being developed. This was briefly touched on in Chapter 12, "Animations," which discussed how using nothing more than animation techniques can be a powerful tool in faking the appearance of virtual intelligence. Other fairly straightforward actions can include the following.

- Playing situation-based audio
- Playing situational animations
- Animation paths

Playing situation-based audio can have a profound impact on the game experience. For example, in *Unreal Tournament III*, during a match AI game characters can call out different things such as "I'm under fire!" or "Enemy flag carrier is near!" In a game, these situational facts can be easily determined (e.g., true or false if the character is being attacked, true or false if an enemy with the flag is within a certain distance, etc.), and the game can therefore play a predefined audio clip randomly when they happen. This can give the illusion that the AI characters are really talking to you and are intelligent, whereas in reality it is just an example of clever design.

Playing situational animations can also have a profound impact on the realism the game is trying to convey to the player. For example, in *Metal Gear Solid 4*, if an enemy thinks he spotted or heard something, he will move toward the source of the sight or sound. During this movement the enemy is often moving slowly and displaying a set of animations to give

the appearance of being cautious and alert. Again, like with situation-based audio, this is merely an example of creative and clever design.

The use of animation paths was discussed in Chapter 12 and can be as simple as moving a character linearly throughout the game world while playing an animation such as a walking movement to give the appearance that the character is a living, breathing inhabitant of the virtual world. In its simplest form, this technique doesn't even have to use any AI processing, other than to check if some condition is true to break the character out of its pattern, such as an attack or an enemy character moving within the character's view.

By mixing and matching these techniques as well as other AI techniques that are common to video games, you can easily find yourself designing complex behavior that is not always easy to efficiently implement. Not only must nonplayable characters give the illusion that they have intelligence, but they must demonstrate it in some games in the form of path finding, squad-based behaviors, strategy, and so forth. Artificial intelligence can become quite challenging as its design becomes more complex.

GAME PHYSICS

Game physics can be tricky. On the one hand, the feature can really add to the experience of a video game. On the other hand, it can be overkill and completely unnecessary for some projects.

Game physics can be far more complex than moving objects linearly and bounding geometry collisions and can include topics such as point masses, rigid bodies, and soft bodies. Rigid and soft bodies are beyond the scope of this book. They are great for games with objects that must collide and interact with the scene and environmental forces realistically, as seen in Valve's *Portal*. Soft bodies are used for objects that can be deformed and morphed in a scene. A very common example of a soft body is a piece of cloth such as a large curtain in a game scene.

Instead of attempting to add physics features to a game, we could probably do more by adding visual effects to the avatars when moving forward, backward, and side to side. One simple example of this is to rotate the player's ship a little as it moves side to side to give it a leaning appearance.

NETWORKING

The online component of commercial video games is very important to the success of many titles. Playing with other gamers can add to the fun factor and extend the life of the game. The online component has become a standard feature in today's commercial games and very few don't offer online connectivity.

Networking and multiplayer gaming can be very beneficial to your future game projects. With online multiplayer games, things can become more complex. When transmitting information across a network, it takes time for the data to go from the sending machine to the receiving machine. This time is measured in milliseconds and is called latency or lag. The challenge with many online games is that this delay can affect the updating of game objects on all machines connected to the gaming session. Locally, this is not an issue since the information does not have to leave the machine, but across the Internet things become complex. Other issues that need to be addressed in an online game are:

- What to do with packets (transmitted information) that are dropped (never received)
- What to do about and during long periods of time between the receipt of new information

- How to ensure that all machines connected to the game are using the same or reasonably accurate information
- How to improve networking performance with the data sent across the network
- How to send all the information necessary for the game while staying within the user's hardware and bandwidth limits
- How to prevent cheating when it comes to the information received from another machine across the Internet

MOVING FORWARD

Congratulations on making it through this book. We hope you have already begun work on your own DirectX 10–based gaming project. To take your potential to new heights, there are a few very important general areas of study that should follow this book. These include formally learning video game design and learning about game engines and technology. Although these topics are separate professions on their own, as an individual or as a member of a small team working on a game, it is important that you understand more than just game and graphics programming.

GAME DESIGN

Game design is a very complex and challenging subject, and it will become more complex as games evolve in future generations. Although we all would like to think we can design gold, the truth is that game design is a process best learned through experience and a lot of dedication and hard work.

Designing a video game is more than creating a cool leading character or cool quests. There is also level and environment design, back and plot stories, puzzles, weapons and items, controls and game mechanics, and much more. Game design is all about creating a game that people would like to play. Unfortunately, making a game fun and engaging is one of those things that sounds easier than it is. Poor design can lead to repetitive game-play, bad controls, bad story, uninspired characters, and an overall negative experience. It is far easier to design a bad game than it is to design a good one.

GAME ENGINES

If you are going to work in the games industry, even as a hobby, you are inevitably going to encounter the term *game engine*. A game engine is a framework that game developers use as a foundation when developing their games. This allows game developers to abstract parts of their game into a framework (e.g., rendering algorithms, input detection, and so forth) that can be used in multiple projects or licensed to other developers.

A game engine essentially takes services that need to be performed and places them in a high-level framework. In a game engine you might have the following features and abilities.

- Material system
- Code for streaming information from a source such as a DVD
- High-level rendering system that can efficiently display the geometry given with various materials
- Audio system (e.g., playing sounds, mixing effects, streaming audio, etc.)

- Networking system
- Physics system
- Cinematic system (i.e., real-time cut-scenes)
- Scripting language and tools (e.g., virtual machine, compiler, text editor, etc.)
- Game tools and editors (e.g., level editor, material editor, etc.)

Some development companies offer middleware solutions to other game developers, and this has proven to be a big business. One of the many main reasons a company would license a game engine is to save the time and resources required to develop the technology themselves. If the technology already exists and is proven, then licensing it could be more beneficial than trying to develop a game engine.

XNA

XNA is a game development framework created by Microsoft for the development of games on Windows-based PCs and the Xbox 360 home gaming console. XNA is not a replacement for DirectX; rather it is built on top of it. XNA uses C# and the Visual Studio toolset and is available for free to anyone interested in using it to create their own video games.

XNA can be a great way for hobbyist and independent game developers to get started in video game development. Microsoft has made it easy for anyone to join their Creators Club Web site, where they can upload their games for review (which can land the game on the Xbox 360) and review and play games created by other members. Although XNA is a framework, it is not a game engine like Epic's Unreal 3.0 engine. XNA is not very high level and is more general than what would be considered a game engine. Also, it does not have the technology that some commercial game engines have such as streaming data from a disk, a level editor, information and game management data structures, and so forth.

XNA is something to consider if you are a hobbyist or small game developer because it is a useful tool. It is also one of the best ways to get your smaller game ideas into the Xbox Live marketplace.

SUMMARY

The following elements were discussed in this chapter:

- Possible game-play improvements
- Possible game-design improvements
- Possible additions to a basic game
- Scene management
- Geometry culling
- Artificial intelligence, game physics, and networking
- Game engines

- XNA

CHAPTER QUESTIONS

Answers to the following chapter review questions can be found in Appendix A.

1. List the four examples of what can be added to improve game-play. Explain each one.

2. List the three examples of what can be added to improve game design. Explain each one.

3. Why would it not be the best idea to make major changes to a game design once the game has been created?

4. What is the benefit to adding multiplayer support to a game project?

5. Define the general term *scene management.*

6. Define level-of-detail and explain why it is so useful in video games.

7. What is geometry culling?

8. What are some examples given in this chapter that can be used to cull geometry from rendering?

9. List and describe four topics one would have to take into consideration when taking a game online.

10. What is a game engine?

11. List five possible features of a game engine that were mentioned in this chapter.

12. Why would a developer license a game engine?

13. What is XNA?

14. For what platforms is XNA available?

15. For what main reason stated in this chapter would an individual or a small development team look into using XNA?

APPENDIX A. ANSWERS TO CHAPTER QUESTIONS

Chapter 1 Answers

Chapter 2 Answers

Chapter 3 Answers

Chapter 4 Answers

Chapter 5 Answers

Chapter 6 Answers

Chapter 7 Answers

Chapter 8 Answers

Chapter 9 Answers

Chapter 10 Answers

Chapter 11 Answers

Chapter 12 Answers

Chapter 13 Answers

Chapter 14 Answers

CHAPTER 1 ANSWERS

1. What is DirectX? When was DirectX released and for what operating system?

Answer: DirectX is a multimedia technology created by Microsoft, originally for Windows 95, in the early to mid 1990s.

2. What does COM stand for?

Answer: Component Object Model.

3. List at least four APIs that make up DirectX 10.

Answer: Direct3D, XInput, XACT, DInput, DirectSound.

4. List two APIs that are no longer part of DirectX 10 but were part of previous versions of DirectX.

Answer: DirectDraw, DirectPlay, DirectShow, DirectMusic.

5. What are XINPUT and XACT? How do they fit into the DirectX technology?

Answer: XInput is used to detect input from game controllers such as Xbox 360 game pads, steering wheels, and so forth. It is a replacement for DirectInput when using game controllers. XACT is a new high-level audio technology that replaces DirectSound. Actually, XAudio2 is the replacement for DirectSound, and XACT3 is a high-level tool built from XAudio2.

6. Describe what a shader is and why it is so important to graphics programmers.

Answer: A shader is a piece of code executed by the graphics hardware during the rendering pipeline to manipulate data in a way desired by the graphics programmers. They are used to allow developers to write their own effects and GPU-based code that can be executed by an application.

7. Describe what a vertex shader is. Describe how the vertex shader is used in relation to the geometry and pixel shaders.

Answer: The vertex shader is code executed on each rendered vertex in the rendering pipeline. The vertex shader sits before the geometry shader and is the first out of the three DirectX 10 shaders.

8. Describe what a geometry shader is. Describe how the geometry shader is used in relation to the vertex and pixel shaders.

Answer: A geometry shader is executed on each rendered primitive in the rendering pipeline. The geometry shader falls after the vertex shader but before the pixel shader.

9. Describe what a pixel shader is. Describe how the pixel shader is used in relation to the vertex and geometry shaders.

Answer: A pixel shader is executed on each rendered pixel. The pixel shader comes after the vertex shader if no geometry shader exists. If a geometry shader does exist, then the pixel shader will follow the geometry shader instead of the vertex shader.

10. What is the difference between managed and unmanaged code? List at least three programming languages that are supported by the .NET managed environment.

Answer: Managed code is code that is compiled into an intermediate language by using a

shared unified set of class libraries. In a managed environment, a run-time-aware compiler takes the intermediate code and translates it to native code during the application's execution. During translation, things such as array bounds checking, garbage collection, type safety, exception handling, and so forth are handled. Languages include .NET C++, .NET C#, .NET J#, .NET Visual Basic, and .NET JScript.

11. What does MDX stand for? What does XNA stand for?

Answer: Managed DirectX is also commonly known as MDX. XNA does not stand for anything; it is the name of one of Microsoft's more recent game development technologies.

12. What is WGF? What new name did it get?

Answer: Windows Graphics Foundation. DirectX.

13. What was the code name for Windows Vista? What was the codename for the original Xbox?

Answer: The code name was Longhorn for Vista and DirectXbox for the Xbox.

14. List the three high-level shading languages discussed in this chapter.

Answer: GLSL, HLSL, and Cg.

15. List four of the five features we've discussed for Windows Vista.

Answer: The Windows Game Explorer, Windows LIVE, flexible parental controls, DirectX 10, and Windows/Xbox connectivity.

16. True or false: The first version of DirectX was released for Windows 3.1.

Answer: False

17. True or false: All versions of DirectX, versions 1 through 10, have been released.

Answer: False (DirectX 4 was not released.)

18. True or false: Direct3D 10 uses Shader Model 4.0 for programmable shaders along with a fixed-function pipeline.

Answer: False

19. True or false: Geometry shaders were first introduced in Shader Model 3.0 and are now being used in Direct3D 10 and OpenGL 3.0.

Answer: False

20. True or false: XNA is a high-level framework built from DirectX.

Answer: True

APPENDIX A. ANSWERS TO CHAPTER QUESTIONS

Chapter 1 Answers

Chapter 2 Answers

Chapter 3 Answers

Chapter 4 Answers

Chapter 5 Answers

Chapter 6 Answers

Chapter 7 Answers

Chapter 8 Answers

Chapter 9 Answers

Chapter 10 Answers

Chapter 11 Answers

Chapter 12 Answers

Chapter 13 Answers

Chapter 14 Answers

CHAPTER 1 ANSWERS

1. What is DirectX? When was DirectX released and for what operating system?

Answer: DirectX is a multimedia technology created by Microsoft, originally for Windows 95, in the early to mid 1990s.

2. What does COM stand for?

Answer: Component Object Model.

3. List at least four APIs that make up DirectX 10.

Answer: Direct3D, XInput, XACT, DInput, DirectSound.

4. List two APIs that are no longer part of DirectX 10 but were part of previous versions of DirectX.

Answer: DirectDraw, DirectPlay, DirectShow, DirectMusic.

5. What are XINPUT and XACT? How do they fit into the DirectX technology?

Answer: XInput is used to detect input from game controllers such as Xbox 360 game pads, steering wheels, and so forth. It is a replacement for DirectInput when using game controllers. XACT is a new high-level audio technology that replaces DirectSound. Actually, XAudio2 is the replacement for DirectSound, and XACT3 is a high-level tool built from XAudio2.

6. Describe what a shader is and why it is so important to graphics programmers.

Answer: A shader is a piece of code executed by the graphics hardware during the rendering pipeline to manipulate data in a way desired by the graphics programmers. They are used to allow developers to write their own effects and GPU-based code that can be executed by an application.

7. Describe what a vertex shader is. Describe how the vertex shader is used in relation to the geometry and pixel shaders.

Answer: The vertex shader is code executed on each rendered vertex in the rendering pipeline. The vertex shader sits before the geometry shader and is the first out of the three DirectX 10 shaders.

8. Describe what a geometry shader is. Describe how the geometry shader is used in relation to the vertex and pixel shaders.

Answer: A geometry shader is executed on each rendered primitive in the rendering pipeline. The geometry shader falls after the vertex shader but before the pixel shader.

9. Describe what a pixel shader is. Describe how the pixel shader is used in relation

to the vertex and geometry shaders.

Answer: A pixel shader is executed on each rendered pixel. The pixel shader comes after the vertex shader if no geometry shader exists. If a geometry shader does exist, then the pixel shader will follow the geometry shader instead of the vertex shader.

10. What is the difference between managed and unmanaged code? List at least three programming languages that are supported by the .NET managed environment.

Answer: Managed code is code that is compiled into an intermediate language by using a shared unified set of class libraries. In a managed environment, a run-time-aware compiler takes the intermediate code and translates it to native code during the application's execution. During translation, things such as array bounds checking, garbage collection, type safety, exception handling, and so forth are handled. Languages include .NET C++, .NET C#, .NET J#, .NET Visual Basic, and .NET JScript.

11. What does MDX stand for? What does XNA stand for?

Answer: Managed DirectX is also commonly known as MDX. XNA does not stand for anything; it is the name of one of Microsoft's more recent game development technologies.

12. What is WGF? What new name did it get?

Answer: Windows Graphics Foundation. DirectX.

13. What was the code name for Windows Vista? What was the codename for the original Xbox?

Answer: The code name was Longhorn for Vista and DirectXbox for the Xbox.

14. List the three high-level shading languages discussed in this chapter.

Answer: GLSL, HLSL, and Cg.

15. List four of the five features we've discussed for Windows Vista.

Answer: The Windows Game Explorer, Windows LIVE, flexible parental controls, DirectX 10, and Windows/Xbox connectivity.

16. True or false: The first version of DirectX was released for Windows 3.1.

Answer: False

17. True or false: All versions of DirectX, versions 1 through 10, have been released.

Answer: False (DirectX 4 was not released.)

18. True or false: Direct3D 10 uses Shader Model 4.0 for programmable shaders along with a fixed-function pipeline.

Answer: False

19. True or false: Geometry shaders were first introduced in Shader Model 3.0 and are now being used in Direct3D 10 and OpenGL 3.0.

Answer: False

20. True or false: XNA is a high-level framework built from DirectX.

Answer: True

CHAPTER 2 ANSWERS

1. What is Direct3D? What is DirectDraw? How are Direct3D and DirectDraw related?

Answer: Direct3D is the 3D rendering API that is part of DirectX. DirectDraw was used for low-level graphics, mainly 2D, in DirectX. DirectDraw and Direct3D were combined into one API.

2. What does HAL stand for?

A. Hardware Application Layer

B. Hardware Abstraction Layer

C. It is not short for anything

D. None of the above

Answer: B

3. What does HEL stand for?

A. Hardware Emulation Layer

B. Hardware Experience Layer

C. It is not short for anything

D. None of the above

Answer: A

4. What is the name of the Direct3D 9 version from Vista?

A. Direct3D 9V

B. Direct3D 9 Vista

C. Direct3D 9L

D. Vista is Direct3D 10 only

Answer: C

5. What does REF stand for in Direct3D?

Answer: Reference mode

6. Describe page flipping.

Answer: Page flipping is where you render to one buffer, display it, render to another buffer, then display that, and repeat.

7. Describe double buffering.

Answer: Double buffering is where the contents from one buffer are copied to another before being displayed.

8. What are swap chains? How do they differ from Direct3D 9 and Direct3D 10?

Answer: A swap chain is an object that is made up of various rendering buffers that is tied to a specific window. In Direct3D 10 you have to create a swap chain for the main rendering, but in Direct3D 9 you do not have to manually do this unless you want additional swap chains.

9. What are render target views? What is their purpose in Direct3D?

Answer: Render targets are surfaces that are rendered to. In Direct3D 10 you have to create a render target even for the back buffer, although in Direct3D 9 you do not have to manually do this for the back buffer.

10. When drawing text, what flag can be used to calculate the rectangle of the text? What is the side effect of using this flag when it comes to drawing text?

Answer: The flag used is `DT_CALCRECT`. The side effect is that the function needs to be called twice.

11. List three high-level programmable shading languages.

Answer: Cg, HLSL, GLSL.

12. What is T&L? When was T&L added to Direct3D?

Answer: T&L is transformations and lighting. It was added in DirectX 7.

13. List four purposes to displaying text in a video game.

Answer: Displaying player information, game play information, player-to-player communications, timers.

14. List the two functions needed to create and display text using Direct3D 10. Describe each of the parameters the functions take.

Answer: `D3DX10CreateFont()` and `DrawText()`. The `D3DX10CreateFont()` function takes as its first parameter the Direct3D rendering device. The second and third parameters are the font's size in logical units. The fourth parameter is the weight, which controls the boldness of the font. The fifth parameter controls the number of mip map levels. The sixth parameter is a flag for whether italics are to be used with the font or not. The seventh parameter is the character set, which can be `ANSI_CHARSET` if using an ANSI strings, or you can use Unicode strings. The eighth parameter is the output precision, which controls how Windows decides how to match desired font sizes with the actual fonts. The ninth and tenth parameters are used for matching the font's desired quality with the font's default quality and to set the font's pitch and family indexes. The last two parameters are used to specify the name of a font that is installed on your system and the output address for the font object to be created by this function. The `DrawText()` function takes as its first parameter a Direct3D 10 sprite object that contains the string to be drawn. The second parameter is the string that is to be displayed to the screen. The third parameter is the number of characters in the string. The fourth parameter is a rectangle area that specifies the region in which the text can be drawn. The fifth parameter is the format of

how the text should be displayed. The last parameter is the RGBA color with which the text should be displayed.

15. True or false: HAL and HEL were introduced in Direct3D 10.

Answer: False

16. True or false: Direct3D supports software rendering and hardware graphics.

Answer: True

17. True or false: Direct3D 9 is the first API to do away with the fixed-function pipeline.

Answer: False

18. True or false: Page flipping copies data from one buffer to another when it is time to display a rendered scene.

Answer: False

19. True or false: A swap chain in Direct3D 10 is tied to the window.

Answer: True

20. True or false: Render targets inform Direct3D where to store the results of a rendering.

Answer: True

CHAPTER 3 ANSWERS

1. Define a primitive.

Answer: A primitive is a simple shape that can be rendered.

2. List three types of primitives that Direct3D supports.

Answer: Points, lines, and triangles.

3. List the different types of lines. Describe each one.

Answer: Line lists, which are lists of individual lines, and line strips, which are connected lines that build off of the end point of the previous line.

4. List the different types of triangles. Describe each one.

Answer: Triangle lists, which are lists of individual triangles, triangle strips, which are connected triangles that build off of the end edge of the previous triangle, and triangle fans, which are triangles that connect to a common point.

5. What is a vertex? What is a vector? How are the two similar?

Answer: A vertex is a point of a primitive that makes up its shape's definition. Vertices often have various attributes including positions, normals, texture coordinates, and so on. A vector is a direction in space. For example, a normal is a vector because it describes a direction.

6. What is an input layout?

Answer: An input layout is the layout of the data being sent to Direct3D.

7. What is a vertex buffer?

Answer: A vertex buffer is a buffer that stores vertex information that is to be rendered to Direct3D.

8. What is the input assembler, and how is it used to set up geometry in Direct3D?

Answer: The input assembler is used to bind data to Direct3D to prepare it for rendering through various function calls.

9. What are indices, and how are they used in the rendering of geometry?

Answer: Indices are array indexes that are used to specify the primitives of an object from a list of vertices, where the orders of the vertices themselves are unimportant.

10. Describe techniques.

Answer: A technique is an implementation of an effect in a shader file. You can have different technique declarations that perform the same graphical effect. For example, you can perform an effect that uses code targeted to different

platforms.

11. True or false: A vertex is a point's position.

Answer: False (Vertices can have a position but also many more attributes.)

12. True or false: There are eight bytes in a bit.

Answer: False (There are eight bits in a byte.)

13. True or false: Indices are vertex index positions of each primitive's vertex points.

Answer: True (They are array positions.)

14. True or false: Rasterizer states control how Direct3D is set up.

Answer: False (They control rasterizer operations, not API setup.)

15. True or false: The alpha channel is often a control for transparency.

Answer: True

16. True or false: HLSL is Direct3D's high level programmable shading language.

Answer: True (when discussing shading languages that are part of the API's framework)

17. True or false: Back face culling is the ability to not draw polygons far away from the camera.

Answer: False

18. True or false: The fill mode controls how the surface is shaded.

Answer: True

19. True or false: A technique defined in an HLSL shader is an effect with one or more passes.

Answer: True

20. True or false: *Topology* is a term used to describe the primitive type of geometry.

Answer: True

CHAPTER 4 ANSWERS

1. Define programmable shaders.

Answer: A shader is code that is executed on the GPU.

2. What does HLSL stand for?

Answer: High-Level Shading Language.

3. What does GLSL stand for?

Answer: OpenGL Shading Language.

4. What is a shader model? What are the versions discussed in this chapter?

Answer: A shader model is a version of shading technology. Almost every generation of DirectX had one or two shader models.

5. What is the difference between low-level and high-level shaders?

Answer: Low-level shaders use an assembly type language for their syntax, while high-level shaders use a high-level language such as C++.

6. List three of the issues discussed in this chapter that occur when working with low-level programmable shaders.

Answer: Harder to read, harder to maintain, and takes more time to develop.

7. Define a vertex shader. What stage(s) accepts the vertex shader's output as input?

Answer: If present, the geometry shader takes the vertex shader's output as its input, but if not, the pixel shader takes the vertex shader's output as input.

8. Define a geometry shader. What stage(s) accepts the geometry shader's output as input?

Answer: A geometry shader is a shader that is executed for each primitive. The pixel shader takes the output from the geometry shader if one is present.

9. Define a pixel shader. What stage(s) accepts the pixel shader's output as input?

Answer: The output of the pixel shader goes to the rendering surface that is to be displayed.

10. What is the input layout of the input assembler?

Answer: The input layout describes the input structured and used to define a vertex.

11. What data type is used for effect shaders in Direct3D 10?

Answer: ID3D10Effect.

12. What is the fixed-function pipeline?

Answer: It is a set of fixed algorithms and rendering states that are part of the rendering API.

13. List three of the limitations of the fixed-function pipeline that were discussed in this chapter.

Answer: Developers had no control over what a graphics API offered and had to either use what was in the API or use clever tricks to attempt to create effects not supported by the tool. Developers also had no direct control over when new features were added, and individuals could not modify the algorithms that make up the fixed-function pipeline.

14. What does it mean to have a unified shader core (architecture)?

Answer: Unified shader core means that each of the shader types has a unified instruction set, whereas previous versions did not.

15. Between which stages does the geometry shader sit?

Answer: It sits between the vertex shader and the pixel shader.

16. List the various data types in the vector type.

Answer: Any of the scalar types (e.g., float, half, int, bool, double, and uint).

17. List the various data types in the scalar type.

Answer: float, half, int, bool, double, and uint.

18. List the various data types in the matrix type.

Answer: Any of the scalar types followed by the number of rows and columns (e.g., `float3×3`, `float4×4`, `int3×4`, and so forth).

19. List the seven data types in the sampler type.

Answer: texture, `Texture1D`, `Texture1DArray`, `Texture2D`, `Texture2DArray`, `Texture3D`, and `TextureCube`.

20. What is a constant buffer, and what use does it have in shaders?

Answer: A constant buffer tells the API how the variable stored inside of it is used by the shaders. This allows constant buffer variables to be optimized based on their use.

21. List and define the three constant buffer usages discussed in this chapter.

Answer: `cbChangeOnResize`, `cbChangesEveryFrame`, and `cbNeverChanges`

22. What is a texture buffer?

Answer: Texture buffers are specialized buffers for textures.

23. Define semantics.

Answer: The semantics of a variable are an optional piece of information used to bind shader inputs and outputs by the HLSL compiler.

24. What is the `Buffer` type used for in the HLSL syntax?

Answer: A `Buffer` object is treated like an array, where you can store information into the object and read information from it using the `Load()` function.

25. List and define four of the storage classes with which a variable can be defined.

Answer: Extern, static, shared, and uniform.

26. Static cannot be used for what type of variables in an HLSL effect shader?

Answer: Static cannot be used for global variables.

27. Define dynamic branching.

Answer: Dynamic branching is when instructions are used to change the flow of execution.

28. What does the `SV` in `SV_POSITION` stand for?

Answer: System value.

29. Define a perspective projection as discussed in this chapter.

Answer: Perspective projection adds perspective to surfaces that are rendered to give them the appearance of depth as objects move away from the viewer.

30. Define an orthogonal projection as discussed in this chapter.

Answer: Orthogonal projection does not account for depth as surfaces move away from the viewer. In other words, surfaces appear to be the same distance apart even when they are not.

CHAPTER 5 ANSWERS

1. What are projection transformations?

Answer: They are matrices that transform vertices by applying a projection (perspective or orthogonal) to the geometry.

2. Describe orthogonal projection.

Answer: Orthogonal projection projects geometry onto a plane to where the depths of the geometry do not change with distance.

3. Describe perspective projection.

Answer: Perspective projection projects geometry onto a plane where objects appear smaller with distance.

4. In what shader does transformation often take place?

A. Geometry

B. Vertex

C. Pixel

Answer: B

5. Multiplying a vector and a matrix together is known as what?

A. Vector-matrix multiplication

B. Vector transform

C. Concatenation

D. None of the above

Answer: B

6. What are the two types of coordinate systems? Describe each.

Answer: The left-handed system, where the X, Y, and Z axes point in the right, up, and forward directions, respectively, and the right-handed system, where the X, Y, and Z axes point in the left, up, and back directions.

7. What is the purpose of a world matrix?

Answer: To transform geometry from its local position to its world position.

8. What is the purpose of the view matrix?

Answer: The view matrix is used to represent a virtual camera.

9. What is the name of the concatenation result of the projection, world, and view matrices?

Answer: MVP, or model-view-projection matrix.

10. What three elements can be used to build a view matrix in Direct3D?

Answer: Camera position, look-at position, and up direction.

11. True or false: Matrices can be concatenated together.

Answer: True

12. True or false: Vectors can be concatenated into a matrix.

Answer: False

13. True or false: There are generally three types of projections.

Answer: False

14. True or false: World and local are two different names for the same type of matrix.

Answer: True

15. True or false: The order in which matrices are multiplied matters.

Answer: True

CHAPTER 6 ANSWERS

1. What is the purpose of texture mapping?

Answer: It simulates detail on surfaces without adding the actual detail.

2. List three ways textures can be used in computer graphics.

Answer: Color decals, bump maps, and specular maps.

3. What are texture coordinates, and why are they necessary for texture mapping?

Answer: They are used to define how the texture image should be projected onto the surface's plane.

4. What is the purpose of texture filtering?

Answer: Texture filtering samples data from nearby pixels in a texture and combines the results.

5. What is the min texture filter?

Answer: Min stands for minifying and represents cases where the texture is moving away from the viewer.

6. What is the mag texture filter?

Answer: Mag stands for magnification and represents cases where the texture is being magnified on screen (moving closer to the viewer).

7. What are mip maps?

Answer: Multi-resolution maps (i.e., multiple versions of the texture at different resolutions).

8. Describe point filtering.

Answer: Point filtering (also known as nearest-neighbor filtering) samples the closet pixel at the sample point defined by the texture coordinates.

9. Describe bilinear filtering.

Answer: Bilinear filtering averages nearby pixels around the sample point defined by the texture coordinates.

10. Describe trilinear filtering.

Answer: Trilinear filtering averages nearby pixels around the sample point defined by the texture coordinates and averages between the mip map levels.

11. What are 1D textures?

Answer: 1D texture images are images with data along a single axis (e.g., just a width but no height or depth).

12. What are 2D textures?

Answer: 2D textures are images with an X and Y axis.

13. What are 3D textures?

Answer: 3D textures have a width, height, and a depth.

14. What are cube maps? How do they differ from 3D textures?

Answer: A cube map is a set of six 2D images that combine to create a texture whose pixels can be accessed using 3D texture coordinates. 3D textures have a volume, while cube maps do not.

15. What are sphere maps? How do they differ from 2D textures?

Answer: A sphere map is used for spherical mapping. The data is stored in a 2D texture, but the projection used causes the image to map differently.

16. What is multi-texturing?

Answer: Texturing a surface with multiple images.

17. What is multi-sampling?

Answer: Sampling of data multiple times and averaging the results.

18. What are the S3TC texture compression formats? How do they differ from each other?

Answer: DXT1 through DXT5. Each version compresses data differently than the one that came before it, but each of them uses a lossy compression algorithm.

19. What is the difference between lossy and lossless compression?

Answer: Lossy compression results in quality loss, while lossless compression retains the quality of the original data.

20. Why is it a bad idea to compress data that is already compressed when using a lossy algorithm?

Answer: Because lossy compression results in a loss of quality, and compressing already compressed data will result in worse quality.

CHAPTER 7 ANSWERS

1. List at least five ways textures can be used in game graphics outside of directly coloring a surface.

Answer: Bump and normal mapping, specular mapping, alpha mapping, environment mapping, and displacement mapping.

2. What is alpha mapping?

Answer: Specifying alpha values on the pixel level and storing it in an image.

3. What are the two ways alpha blending can be enabled in Direct3D 10?

Answer: Through the Direct3D device object or in the HLSL file.

4. What two ways to store alpha map values were discussed in this chapter?

Answer: They can be stored in a separate image or in the alpha channel of an image.

5. Why would you use 1 bit for alpha values rather than using 1 byte?

Answer: The 1-bit value can act as an off/on flag if the alpha values can be either visible or invisible.

6. What is alpha to coverage, and what part does it play in game graphics?

Answer: Alpha to coverage is a term in computer graphics that deals with multi-sampling and refers to the way alpha mapped surfaces are rendered in scenes that have many overlapping polygons.

7. List and describe five of the blend options that can be used for `SrcBlend` in the blend descriptor.

Answer: `D3D10_BLEND_ZERO, D3D10_BLEND_ONE, D3D10_BLEND_SRC_COLOR, D3D10_BLEND_INV_SRC_COLOR, D3D10_BLEND_SRC_ALPHA, and D3D10_BLEND_INV_SRC_ALPHA.`

8. What is a sprite?

Answer: A sprite is a 2D textured surface, usually of a character or an object related to the game or application.

9. What is a point sprite?

Answer: A point sprite is a hardware-accelerated sprite.

10. How does Direct3D 10 point sprite support differ from Direct3D 9?

Answer: Direct3D 10 does not support it directly, but you can create sprites on the hardware using the geometry shader.

11. What is a billboard sprite? How do you calculate a sprite that always faces the camera?

Answer: A billboard is a surface that is rendered to face the viewer. It can be calculated by projecting the geometry using a matrix built from the view information that allows the object to face toward the viewer.

12. How did we create point sprites in Direct3D 10 in this chapter?

Answer: Using the geometry shader, we rendered a list of points from which triangles were generated.

13. Describe the luminance filter algorithm.

Answer: It is used to convert an image to black and white. It does this by calculating the dot product of the color value with the luminance constant.

14. Describe the color inversion filter algorithm.

Answer: The color inversion filter works by negating the values of the color components. Therefore, white becomes black and black becomes white.

15. Describe the sepia tone filter algorithm.

Answer: The sepia tone filter builds off of the luminance filter and adds a brownish tone to it.

CHAPTER 8 ANSWERS

1. What are vectors?

Answer: A vector is a mathematical structure that represents a direction.

2. What structures does the DirectX SDK offer for vectors?

Answer: `D3DXVECTOR2`, `D3DXVECTOR3`, and `D3DVECTOR4`.

3. What is a plane? What main purpose do planes serve as described in this chapter?

Answer: A plane lies infinitely along two axes. They can be used for collision detection.

4. What is a matrix?

Answer: A matrix is a mathematical structure that is used to transform vectors from one space to another.

5. What is the difference between a 3 × 3 and 4 × 4 matrix?

Answer: A 3 × 3 matrix has three rows and three columns, while a 4 × 4 matrix has four rows and columns.

6. What is bounding geometry and how is it used in computer graphics?

Answer: A piece of bounding geometry is used to surround objects in an effort to act as a simplified representation of the object. One use is collision detection.

7. What coordinate system does Direct3D use traditionally?

Answer: Left-handed.

8. What is the model matrix?

Answer: The model matrix is used to represent the local transformation of objects.

9. What is the view matrix?

Answer: The view matrix is a transformation that is used to represent the viewer.

10. What is the projection matrix?

Answer: The projection matrix adds either perspective or orthogonal projection to rendered objects.

11. What is the MVP matrix?

Answer: It is the model-view-projection matrix, which is the concatenation of the model, view, and projection matrices.

12. What three properties does Direct3D use to create a view matrix that represents a camera?

Answer: The position, look-at position, and up direction vectors.

13. What is a ray, and what two components make up a ray object?

Answer: A ray is a structure with an origin and a direction, both of which can be represented as vectors.

14. How can you limit a ray's infinite direction?

Answer: By using the length of the intersection and testing it against the maximum length desired.

15. What is a quaternion used for? List two benefits of using quaternion rotations versus matrices.

Answer: A quaternion is used for efficient rotations. Computationally, they are faster than matrices, and they consume less memory since a quaternion is made up of four floating-point variables, while a matrix can have 16.

CHAPTER 9 ANSWERS

1. List the various sound technologies that can be found in the various game development SDKs provided by Microsoft.

Answer: XAudio2, XACT3, XAPO, XAPOFX, X3DAudio, and DirectSound.

2. Describe the XAudio2 API. What does this technology replace?

Answer: It is a replacement for DirectSound. XAudio2 is a low-level sound API for the PC and the Xbox 360.

3. Describe XACT3. What does this technology replace?

Answer: XACT3 is also a replacement for DirectSound, but it is a high-level audio API built from XAudio2.

4. Describe the audio effects technologies.

Answer: An audio effect is an object that takes incoming audio data and performs some operation on the data before passing it on.

5. Describe Direct Sound and its current role as an audio technology.

Answer: Direct Sound is an audio API that was updated in the DirectX SDK until DirectX 8. Newer audio APIs have since taken its place, and the API is deprecated.

6. What does XACT stand for?

Answer: The Microsoft Cross-Platform Audio Creation Tool (XACT).

7. List the audio formats supported by XACT3 and XAudio2.

Answer: WAV, AIFF, ADPCM, XMA, and xWMA.

8. What is endian order, and how does it affect audio files (or files in general) and networking data?

Answer: The endian order is the byte ordering of multi-byte variables. If data is transmitted or stored in a different order from the machine reading it, then the destination machine will have to swap the byte ordering to use the correct value.

9. What is a wave bank?

Answer: A wave bank is a file with a collection of audio files that are to be played by the application.

10. What is a sound bank?

Answer: A sound bank holds a list of sounds and sound cues that reference audio from the wave banks.

11. What is a sound cue, and what is it used for in XACT3?

Answer: A sound cue is used to play a sound from the XACT sound bank.

12. What is the main difference between XACT3 and XAudio2 discussed in this chapter?

Answer: XACT3 is high-level, while XAudio2 is low-level for audio API.

13. What is the mastering voice, and what is it used for in XAudio2?

Answer: A mastering voice is the voice that is audible. It sends data it receives from source and submix voices to the audio hardware. The mastering voice is the only voice that allows you to hear anything, so you must create this voice in XAudio2 to hear anything.

14. What is the source voice, and what is it used for in XAudio2?

Answer: A source voice is used to send sound data to the other types of voices, and it represents an audio stream of data.

15. True or false: XAudio2 internally takes care of the endian issue for programmers.

Answer: False

CHAPTER 10 ANSWERS

1. What is DirectInput?

Answer: DirectInput is an input API for using keyboards, mice, and game controllers in applications.

2. What advantage does DirectInput offer when you are using standard keyboards and mice when compared to Win32 functions as discussed in this chapter?

Answer: DirectInput allows special device features to be accessed and used in applications.

3. What is XInput?

Answer: XInput is a newer input API for game controllers.

4. What are the benefits to using XInput over DirectInput as described in this chapter?

Answer: XInput is easier and faster to set up, is updated (DirectInput has not been updated since DirectX 8), and supports any Xbox 360–compatible input device.

5. What are the steps necessary to set up XInput in code? How is XInput enabled and disabled?

Answer: No steps are necessary to set up XInput in code. By default it is enabled, but it can be disabled and enabled by calling `XInputEnable()`.

6. What XInput function is used to obtain the state of a controller device?

Answer: `XInputGetState()`.

7. What field is used to detect button presses on an Xbox controller? How do the button flags factor in when determining if a button is pressed?

Answer: `wButtons`.

8. What fields are used to store the thumb stick locations of an Xbox controller?

Answer: `sThumbLX`, `sThumbRX`, `sThumbLY`, `sThumbRY`.

9. What are the minimum and maximum pressure values a trigger can be on an Xbox controller?

Answer: Minimum of 0 and maximum of 255.

10. List the steps to setting the motor speed in an Xbox controller.

Answer: Create an `XINPUT_VIBRATION` object and call `XInputSetState()`.

11. What is the minimum and maximum power the motors can move at in an Xbox controller?

Answer: 0 to 65,535.

12. What XInput function is used to obtain the controller device's capabilities?

Answer: `XInputGetCapabilities()`.

13. List the three Xbox controller subtypes that were discussed in regard to

obtaining device capabilities.

Answer: Arcade stick, gamepad, and steering wheel.

CHAPTER 11 ANSWERS

1. What object is used to create a file stream for input in C++?

Answer: `ifstream`.

2. What object is used to create a file stream for output in C++?

Answer: `ofstream`.

3. How does the `ReadSome()` function work from the input file stream class?

Answer: `ReadSome()` reads characters from a buffer into an array until all characters have been read or until the memory buffer associated with the stream runs out.

4. What flag is used to create a file stream that is used for binary files?

Answer: `ios_base::binary`.

5. What key reason was given for the use of `seekg()` and `tellg()` from the Files demo?

Answer: To use the two together to obtain the file size.

6. Define a token.

Answer: A block of characters between delimiters.

7. Define a delimiter.

Answer: A character that is used to separate tokens in a stream of characters.

8. Describe how the `TokenStream` class extracts the next token in the data stream.

Answer: It reads characters until it finds a delimiter. Once a delimiter is found, the text that was read is returned to the caller.

9. Describe how the `TokenStream` class extracts the next line in the data stream and how it differs from token extraction.

Answer: It reads characters until a delimiter is reached, at which point the characters that were read are returned as a string.

10. Describe the OBJ file format. How are faces represented in an OBJ file?

Answer: The OBJ file format is a text file that specifies geometry information on each line of the file. The beginning of each line starts with a keyword that describes the data on the line, followed by the data itself.

CHAPTER 12 ANSWERS

1. What is the benefit of using time-based calculations over frame-based ones?

Answer: The simulations can remain more consistent and independent of the frame rate.

2. Describe linear interpolation.

Answer: To interpolate between two numbers we can use linear interpolation. To perform linear interpolation we need three pieces of information. The first two pieces of information are the two numbers between which we are interpolating. The third piece of information needed is a scalar value that is a percentage from 0 to 1, with 1 being 100%. Using 0% basically says we are at point A, and using 100% means we are at point B. Any value between 0% and 100% places the value somewhere between point A and point B.

3. Write down the equation for linear interpolation.

Answer: Final = (B – A) * dt + A.

4. Write down the equation for cubic Bezier curves as described in this chapter.

Answer: Final = A * (1 – S)3 + C1 * 3 * S * (1 – S)2 + C2 * 3 * S2 * (1 – s) + B * S3.

5. What are key-frame animations when it comes to character animation?

Answer: Key-frame animations are used to display a character in different poses, where each frame of animation is a different pose.

6. Describe bone and skeleton animation.

Answer: Bone animation uses a hierarchy of matrices to transform the geometry of an object to give the appearance of animation.

7. What benefits does bone animation have over key-frame animation?

Answer: Bone animation allows for more realistic simulations with less memory footprint and more efficiency than traditional key-frame data stored as separate meshes.

8. A series or collection of paths is known as what?

Answer: A route.

9. A series or collection of routes is known as what?

Answer: A cut-scene (as described in this book).

10. What is the difference between the absolute and relative matrices in bone animation?

Answer: The relative matrix is relative to the bone it represents, while the absolute matrix takes into account the relative matrix of the bone and the absolute matrix from its parent bone.

CHAPTER 13 ANSWERS

1. Define per-vertex lighting.

Answer: Per-vertex lighting is lighting done on each vertex of each primitive.

2. Define per-pixel lighting.

Answer: Per-pixel lighting is lighting that is computed on the pixel level rather than the vertex level.

3. Describe directional lights in computer graphics.

Answer: A directional light is light that shines from a specific direction but has no origin.

4. Describe point lights in computer graphics.

Answer: A point light is a light that emits from a source out into the world in what appears to be an even manner in all directions.

5. Describe spot lights in computer graphics.

Answer: A spot light is a light that emits from a source out into the world but is restricted to shine is certain directions.

6. Describe area lights in computer graphics.

Answer: Area lights are arrays of lights that cover a specific area. The contribution of all of the lights can be used to create soft shadows.

7. What is global illumination?

Answer: It is a term used to refer to algorithms that solve the entire lighting equation. This includes direct light as well as indirect light and visibility (shadows).

8. What is diffuse light? How does it differ from ambient light?

Answer: Diffuse light is used to represent light that scatters evenly across a surface as it hits a surface. Ambient light is used more to simulate light coming indirectly from surrounding surfaces than direct light from the light source.

9. What is specular light? How does it differ from diffuse and ambient light?

Answer: Specular light is used to create highlights on shiny surfaces. Ambient light is used more as fill color, while diffuse light is used to represent light that scatters evenly across a surface. Specular light is light that scatters more along a specific range rather than evenly in all directions.

10. Describe the Blinn-Phong lighting algorithm.

Answer: The Blinn-Phong lighting algorithm is a view-dependent algorithm used for specular highlights. It works by raising N dot H to a specular power, where H is the half vector created from the view vector plus the light vector and N is the surface normal.

11. What does the term *material* mean in modern computer graphics?

Answer: A material is a set of related properties that combine to create a surface's texture. This can include color texture images, normal maps, specular maps, and shaders.

12. List two ways shadows can be faked in games.

Answer: By using imposters or by being hand drawn to give the impression that the shadows are calculated.

13. Generally describe the shadow volumes technique.

Answer: Shadow volumes are extruded pieces of geometry that are rendered to the stencil buffer to mask away areas of the screen that are in shadow. When the scene is rendered a solid color is usually rendered in the shadowed regions while normal rendering is used for the un-shadowed regions.

14. Generally describe the shadow mapping algorithm.

Answer: Shadow mapping works by rendering the scene's depths into a texture from the light's point of view and then using those results with projection mapping to test if the depths in the shadow map are closer to the light than the depths of the surfaces being rendered. If so, then the surface pixel being rendered is in shadow.

15. Describe two ways to produce soft shadows.

Answer: They can be faked (such as blurring a shadow map using a general blurring algorithm) or they can be calculated in real time (such as blurring based on the penumbra or umbra).

CHAPTER 14 ANSWERS

1. List the four examples of what can be added to improve game-play. Explain each one.

Answer: Improved audio effects, which will enhance the player's experience and improve the audio quality of the game; networking support, which will add to the game's re-playability and add to the game's lifespan; dynamic environments, which will give the scene more realism and improve the game-play experience; and improved visuals, which will give the player something visually stimulating that will improve the visual look of the game.

2. List the three examples of what can be added to improve game design. Explain each one.

Answer: Think carefully before adding anything to or changing the design. If it is to be done, take care in how it is brought about or consider creating a sequel or a different game with those new ideas. Be mindful of your limits and try not to do anything that is above your abilities, or you may be setting yourself up for failure. When making the game, go in with some type of game plan and goals so that you are not simply making it up as you go along.

3. Why would it not be the best idea to make major changes to a game design once the game has been created?

Answer: Because those changes can prove difficult to implement, and unforeseen issues can arise from them.

4. What is the benefit to adding multiplayer support to a game project?

Answer: It can enhance the experience of the gamers.

5. Define the general term *scene management.*

Answer: *Scene management* is a general term that often describes steps taken algorithmically to manage a game's data and information in a way that optimizes their use.

6. Define level-of-detail and explain why it is so useful in video games.

Answer: Level-of-detail is the act of displaying different levels of detail of an object based on specific conditions. For example, as an object moves further from the viewer, lower levels of detail of the object can be used since the detail difference is unnoticeable.

7. What is geometry culling?

Answer: The process of eliminating geometry from further processing because it will not ultimately be rendered or would not have an impact on the final scene.

8. What are some examples given in this chapter that can be used to cull geometry from rendering?

Answer: Occlusion culling, back-face culling, and view-frustum culling (which can be used with various data structures used by scene partitioning algorithms).

9. List and describe four topics one would have to take into consideration when taking a game online.

Answer: Latency, which deals with the performance of the networking communications; handling dropped packets, which can cause client machines not to receive important data needed for the game; data security, which is a must for ensuring the integrity of the game's data stays intact; preventing cheating so that players cannot ruin the experience for everyone else; consistency among all connected machines so that some players are not given an unfair advantage because one or more player is not working with the same data; and performance and hardware limits, since these two issues will determine how data is ultimately sent across the network as well as how often.

10. What is a game engine?

Answer: A game engine is a framework of high-level code used to make a video game.

11. List five possible features of a game engine that were mentioned in this chapter.

Answer: A scripting system, a material system, a physics system, a networking system, and code for streaming data from a disk.

12. Why would a developer license a game engine?

Answer: Because licensing proven technology can prove more beneficial to the developer than building it in-house.

13. What is XNA?

Answer: A game development technology that can be used to create PC and Xbox 360 games.

14. For what platforms is XNA available?

Answer: Windows XP, Vista, and the Xbox 360.

15. For what main reason stated in this chapter would an individual or a small development team look into using XNA?

Answer: XNA can be used to develop games for the PC and for the Xbox 360 without any cost to the developer outside of a creator's club subscription if deployment is done on the 360.

APPENDIX B. RECOMMENDED RESOURCES

Recommended Tools

Recommended Books

Recommended Web Sites and Articles

RECOMMENDED TOOLS

Throughout this book we used several free tools for the development of the book's sample source code. These tools are highly recommended and include the following:

- Visual Studio 2005/2008: A Windows XP/Vista–integrated development environment (http://www.microsoft.com/express/)
- DirectX SDK (http://msdn2.microsoft.com/directx/sdk)
- NVIDIA's Melody (and other useful tools) (http://www.developer.nvidia.com/)

RECOMMENDED BOOKS

The books in this section can be of some use to anyone looking to expand their knowledge of the different areas of game development. Reading these books is optional but is recommended for those looking for more detailed knowledge about many of the topics discussed in this book and beyond.

- *Mathematics for 3D Game Programming and Computer Graphics*, Second Edition, Charles River Media (2003)
- *Data Structures for Game Developers*, Charles River Media (2007)
- *Ultimate Game Engine Design and Architecture*, Charles River Media (2006)
- *Game Graphics Programming*, Charles River Media (2008)

RECOMMENDED WEB SITES AND ARTICLES

The Web sites in this section can be of some use to anyone looking to expand their knowledge of the different areas of game development. For additional Web sites and Web articles visit UltimateGameProgramming.com.

- Ultimate Game Programming (http://www.UltimateGameProgramming.com/)
- Microsoft Developer Network (MSDN) (http://msdn.Microsoft.com/)