

A PRACTICAL GUIDE TO  
GRAPHICS PROGRAMMING



REAL-TIME  
3D RENDERING  
with  
**DIRECTX®** and **HLSL**

Paul **VARCHOLIK**

# About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

# Real-Time 3D Rendering with DirectX® and HLSL

## A Practical Guide to Graphics Programming

**Paul Varcholik**



Upper Saddle River, NJ • Boston • Indianapolis • San Francisco  
New York • Toronto • Montreal • London • Munich • Paris • Madrid  
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [international@pearsoned.com](mailto:international@pearsoned.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: 2014933263

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

DirectX, Direct3D, MS-DOS, MSDN, Visual Studio, Windows, Windows Phone, Windows Vista, Xbox, and Xbox 360 are registered trademarks of Microsoft Corporation in the United States and/or other countries.

NVIDIA, GeForce, Nsight, and FX Composer are registered trademarks of NVIDIA Corporation in the United States and/or other countries.

Autodesk, Maya and 3ds Max are registered trademarks of Autodesk, Inc. in the United States and/or other countries.

Rendermonkey is a trademark of Advanced Micro Devices, Inc.

OpenGL is a registered trademark of Silicon Graphics, Inc. in the United States and/or other countries worldwide.

StarCraft and Blizzard Entertainment are trademarks or registered trademarks of Blizzard Entertainment, Inc. in the United States and/or other countries.

COLLADA is a trademark of the Khronos Group Inc.

Photoshop is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

Steam is a registered trademark of Valve Corporation.

Terragen is a trademark of Planetside Software.

Unreal Development Kit and UDK are trademarks or registered trademarks of Epic Games, Inc. in the United States and elsewhere.

Unity Software is a copyright of Unity Technologies.

All other trademarks are the property of their respective owners.

ISBN-13: 978-0-321-96272-0

ISBN-10: 0-321-96272-9

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing: April 2014

**Editor-in-Chief**

Mark Taub

**Executive Editor**

Laura Lewin

**Development Editor**

Songlin Qiu

**Managing Editor**

Kristy Hart

**Senior Project Editor**

Lori Lyons

**Copy Editor**

Krista Hansing Editorial  
Services, Inc.

**Indexer**

Tim Wright

**Proofreader**

Debbie Williams

**Technical Reviewers**

Michael Gourlay  
Joel Martinez  
Budirijanto Purnomo

**Editorial Assistant**

Olivia Basegio

**Cover Designer**

Chuti Prasertsith

**Senior Compositor**

Gloria Schurick

# Praise for *Real-Time 3D Rendering with DirectX and HLSL*

“I designed and taught the technical curriculum at UCF’s FIEA graduate program and was never satisfied with textbooks available for graphics programming. I wish I had Paul Varcholik’s book then; it would make the list now.”

—**Michael Gourlay**, Principal Development Lead, Microsoft

“Modern 3D rendering is a surprisingly deep topic; one that spans several different areas. Many books only focus on one specific aspect of rendering, such as shaders, but leave other aspects with little or no discussion. *Real-Time 3D Rendering with DirectX and HLSL* takes the approach of giving you a full understanding of what a modern rendering application consists of, from one end of the pipeline to the other.”

—**Joel Martinez**, Software Engineer, Xamarin

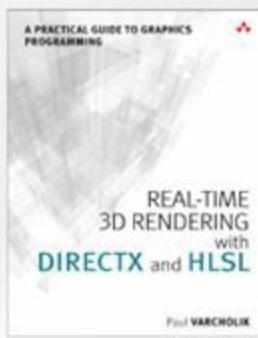
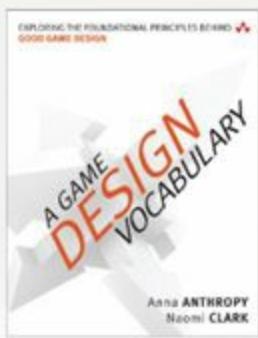
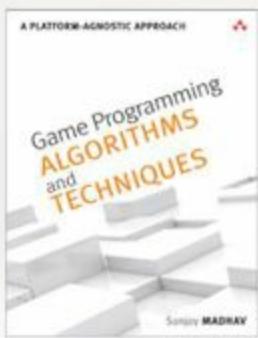
“This practical book will take you on a journey of developing a modern 3D rendering engine through step-by-step code examples. I highly recommend this well-written book for anyone who wants to learn the necessary graphics techniques involved in developing a 3D rendering engine using the latest Direct3D.”

—**Budirijanto Purnomo**, GPU Developer Tools Lead, Advanced Micro Devices, Inc.

“A great tour of the modern DirectX landscape, with a heavy emphasis on authoring HLSL shaders for common game rendering techniques for C++ developers.”

—**Chuck Walbourn**, Senior Design Engineer, Microsoft

# The Addison-Wesley Game Design and Development Series



▲ Addison-Wesley

Visit [informit.com/series/gamedesign](http://informit.com/series/gamedesign) for a complete list of available publications.

## Essential References for Game Designers and Developers

These practical guides, written by distinguished professors and industry gurus, cover basic tenets of game design and development using a straightforward, common-sense approach. The books encourage readers to try things on their own and think for themselves, making it easier for anyone to learn how to design and develop digital games for both computers and mobile devices.



Make sure to connect with us!  
[informit.com/socialconnect](http://informit.com/socialconnect)

**informIT.com**  
the trusted technology learning source

▲ Addison-Wesley

**Safari**  
Books Online

ALWAYS LEARNING

**PEARSON**

# **Contents-at-a-Glance**

## Introduction

### Part I An Introduction to 3D Rendering

[1 Introducing DirectX](#)

[2 A 3D/Math Primer](#)

[3 Tools of the Trade](#)

### Part II Shader Authoring with HLSL

[4 Hello, Shaders!](#)

[5 Texture Mapping](#)

[6 Lighting Models](#)

[7 Additional Lighting Models](#)

[8 Gleaming the Cube](#)

[9 Normal Mapping and Displacement Mapping](#)

### Part III Rendering with DirectX

[10 Project Setup and Window Initialization](#)

[11 Direct3D Initialization](#)

[12 Supporting Systems](#)

[13 Cameras](#)

[14 Hello, Rendering!](#)

[15 Models](#)

[16 Materials](#)

[17 Lights](#)

### Part IV Intermediate-Level Rendering Topics

[18 Post-Processing](#)

[19 Shadow Mapping](#)

[20 Skeletal Animation](#)

[21 Geometry and Tessellation Shaders](#)

[22 Additional Topics in Modern Rendering](#)

[Index](#)

# Contents

[Introduction](#)

## [Part I An Introduction to 3D Rendering](#)

[1 Introducing DirectX](#)

[A Bit of History](#)

[The Direct3D 11 Graphics Pipeline](#)

[Summary](#)

[2 A 3D/Math Primer](#)

[Vectors](#)

[Matrices](#)

[Transformations](#)

[DirectXMath](#)

[Summary](#)

[3 Tools of the Trade](#)

[Microsoft Visual Studio](#)

[NVIDIA FX Composer](#)

[Visual Studio Graphics Debugger](#)

[Graphics Debugging Alternatives](#)

[Summary](#)

[Exercises](#)

## [Part II Shader Authoring with HLSL](#)

[4 Hello, Shaders!](#)

[Your First Shader](#)

[Hello, Structs!](#)

[Summary](#)

[Exercises](#)

[5 Texture Mapping](#)

[An Introduction to Texture Mapping](#)

[A Texture Mapping Effect](#)

[Texture Filtering](#)

[Texture Addressing Modes](#)

[Summary](#)

## [Exercises](#)

### [\*\*6 Lighting Models\*\*](#)

[Ambient Lighting](#)

[Diffuse Lighting](#)

[Specular Highlights](#)

[Summary](#)

[Exercises](#)

### [\*\*7 Additional Lighting Models\*\*](#)

[Point Lights](#)

[Spotlights](#)

[Multiple Lights](#)

[Summary](#)

[Exercises](#)

### [\*\*8 Gleaming the Cube\*\*](#)

[Texture Cubes](#)

[Skyboxes](#)

[Environment Mapping](#)

[Fog](#)

[Color Blending](#)

[Summary](#)

[Exercises](#)

### [\*\*9 Normal Mapping and Displacement Mapping\*\*](#)

[Normal Mapping](#)

[Displacement Mapping](#)

[Summary](#)

[Exercises](#)

## [\*\*Part III Rendering with DirectX\*\*](#)

### [\*\*10 Project Setup and Window Initialization\*\*](#)

[A New Beginning](#)

[Project Setup](#)

[The Game Loop](#)

[Window Initialization](#)

[Summary](#)

[Exercise](#)

## 11 Direct3D Initialization

[Initializing Direct3D](#)

[Putting It All Together](#)

[Summary](#)

[Exercise](#)

## 12 Supporting Systems

[Game Components](#)

[Device Input](#)

[Software Services](#)

[Summary](#)

[Exercises](#)

## 13 Cameras

[A Base Camera Component](#)

[A First-Person Camera](#)

[Summary](#)

[Exercise](#)

## 14 Hello, Rendering!

[Your First Full Rendering Application](#)

[An Indexed Cube](#)

[Summary](#)

[Exercises](#)

## 15 Models

[Motivation](#)

[Model File Formats](#)

[The Content Pipeline](#)

[The Open Asset Import Library](#)

[What's in a Model?](#)

[Meshes](#)

[Model Materials](#)

[Asset Loading](#)

[A Model Rendering Demo](#)

[Texture Mapping](#)

[Summary](#)

[Exercises](#)

## 16 Materials

[Motivation](#)

[The Effect Class](#)

[The Technique Class](#)

[The Pass Class](#)

[The Variable Class](#)

[The Material Class](#)

[A Basic Effect Material](#)

[A Skybox Material](#)

[Summary](#)

[Exercises](#)

## 17 Lights

[Motivation](#)

[Light Data Types](#)

[A Diffuse Lighting Material](#)

[A Diffuse Lighting Demo](#)

[A Point Light Demo](#)

[A Spotlight Demo](#)

[Summary](#)

[Exercises](#)

# Part IV Intermediate-Level Rendering Topics

## 18 Post-Processing

[Render Targets](#)

[A Full-Screen Quad Component](#)

[Color Filtering](#)

[Gaussian Blurring](#)

[Bloom](#)

[Distortion Mapping](#)

[Summary](#)

[Exercises](#)

## 19 Shadow Mapping

[Motivation](#)

[Projective Texture Mapping](#)

[Shadow Mapping](#)

[Summary](#)[Exercises](#)

## [20 Skeletal Animation](#)

[Hierarchical Transformations](#)[Skinning](#)[Importing Animated Models](#)[Animation Rendering](#)[Summary](#)[Exercises](#)

## [21 Geometry and Tessellation Shaders](#)

[Motivation: Geometry Shaders](#)[Processing Primitives](#)[A Point Sprite Shader](#)[Primitive IDs](#)[Motivation: Tessellation Shaders](#)[The Hull Shader Stage](#)[The Tessellation Stage](#)[The Domain Shader Stage](#)[A Basic Tessellation Demo](#)[Displacing Tessellated Vertices](#)[Dynamic Levels of Detail](#)[Summary](#)[Exercises](#)

## [22 Additional Topics in Modern Rendering](#)

[Rendering Optimization](#)[Deferred Shading](#)[Global Illumination](#)[Compute Shaders](#)[Data-Driven Engine Architecture](#)[The End of the Beginning](#)[Exercises](#)[Index](#)

# Acknowledgments

I would like to thank the many people who helped make this book possible. First, to the wonderful team at Pearson, especially Laura Lewin, Olivia Basegio, and Songlin Qiu. You have made this a truly enjoyable experience.

Next, to my technical reviewers, Joel Martinez, Dr. Michael Gourlay, and Budi Purnomo, for your time and expert advice. Your insights have made this book so much better. A special thanks to Michael Gourlay, who contributed the code for the Runtime Type Information (RTTI) and Factory discussions.

I would also like to thank my students and colleagues at the Florida Interactive Entertainment Academy, especially Nick Zuccarello, Brian Salisbury, and Brian Maricle, who contributed 3D models and textures.

Finally, to my wife Janette, who not only provided a seemingly infinite supply of encouragement and patience, but also contributed more than a few of the illustrations.

# About the Author

**Dr. Paul Varcholik** is a programming instructor at the Florida Interactive Entertainment Academy (FIEA), a graduate degree program in game development at the University of Central Florida. Before coming to FIEA, Paul was a lead software engineer at Electronic Arts, where he worked on video game titles including *Madden NFL Football* and *Superman Returns*. Paul is a 20-year veteran of the software industry and has been teaching college courses on software and game development since 1998. Paul has written extensively on topics including robotics, 3D user interaction, and multitouch interfaces. He is also the author of *OpenGL Essentials LiveLessons*, a video series on graphics development using OpenGL.

# Introduction

Graphics programming is the magic behind video games, film, and scientific simulation. Every explosion, dust particle, and lens flare you see on a computer screen is processed through a graphics card. In addition, because modern operating systems use the graphics processing unit (GPU) to draw their content, every pixel you see is rendered through the GPU and through software developed by a graphics programmer. It's a broad topic, but one that has traditionally been the province of a select few. Even to experienced software developers, rendering is often considered a dark art, full of complex mathematics and esoteric tools. Furthermore, the rapid pace of advancement makes modern graphics programming a moving target, and establishing a foothold can be difficult.

That's where this book comes in. In these pages, you'll find an introduction to real-time 3D rendering. I've presented this material in a straightforward and practical way, but it doesn't shy away from more complex topics. Indeed, this book takes you far beyond drawing simple objects to the screen and introduces intermediate and advanced subjects in modern rendering. It is my sincere wish that you find the material in this book approachable, applicable, and up-to-date with modern graphics techniques.

## Intended Audience

This book is intended for experienced software engineers new to graphics programming. The text often uses terminology from the video game industry, but you need not be a game developer to make use of this book. Indeed, the topic of modern rendering reaches well beyond video games and is becoming ever more pervasive in a variety of software-related fields. Regardless of the specific type of software you develop, if you are interested in learning about modern rendering, this book is for you.

This text also assists existing graphics programmers who are new to DirectX or who are familiar with an older version of the library. We cover DirectX, and the library has seen major changes over the last few years. This book also applies to students, hobbyists, and technical artists interested in real-time rendering.

If you are new to programming, not specifically graphics programming, this book might not be what you're looking for. In particular, [Part III, “Rendering with DirectX,”](#) develops a C++ rendering engine and expects a familiarity with that language.

## Why This Book?

Several excellent books on the market explore graphics programming. However, most of these texts focus on only one area: either shader programming or the rendering API (such as DirectX or OpenGL). Mention of the other topic, the other side of the same coin, is often given short shrift—perhaps just a chapter or two.

Modern rendering doesn't exist without shaders, but shaders aren't executed without an underlying graphics application. I know of few books that incorporate both topics in a thorough, integrated fashion, nor one that balances introductory material with intermediate and advanced topics. A sticking point with many books is that they are either so novice that they leave the reader wanting or so advanced that even experienced software engineers have trouble absorbing the material.

You can also find a number of good books on general-purpose game or engine programming. These texts often include material on graphics programming or approaches to organizing 3D models and materials. But these books often have such a broad scope that rendering gets lost in the pages.

The approaches I've mentioned are all valid, and, again, you can find many wonderful books on the market. Yet I'm seeing a gap where an experienced developer who wants to tackle graphics programming is missing a text that offers a full, focused treatment of rendering from both the CPU and GPU sides of the topic. This book aims to fill this gap.

## How This Book Is Organized

This book is organized into four parts:

- **Part I, “An Introduction to 3D Rendering,”** provides an introduction to graphics programming. It includes a discussion about the history of DirectX up to version 11.1 (the version we’re using in this book) and looks at the Direct3D graphics pipeline. [Chapter 2](#) includes a primer on 3D math, along with a detailed look at the DirectX Math API. If you are already familiar with linear algebra and 3D mathematics, you might consider skipping or skimming [Chapter 2](#). However, I encourage you to read the section on DirectX Math (Microsoft’s latest revision of an impressive SIMD-friendly math library focused on graphics-related mathematics). [Part I](#) ends with an exploration of best-of-breed tools for authoring and debugging shaders and graphics applications.
- **Part II, “Shader Authoring with HLSL,”** is all about shaders and programming using the High-Level Shader Language (HLSL). This section begins with the most introductory vertex and pixel shaders and a discussion of semantics and annotations. [Chapter 5](#) examines texture mapping and texture filtering and wrapping modes. [Chapter 6](#) introduces basic lighting models, including ambient lighting, diffuse (Lambert) lighting, and specular highlighting. [Chapter 7](#) details point lights, spot lights, and multiple lights. In [Chapter 8](#), you write shaders involving cube maps, including shaders for skyboxes and environment mapping. [Part II](#) concludes with a potpourri of shaders for fog, color blending, normal mapping, and displacement mapping.
- In **Part III, “Rendering with DirectX,”** we discuss the application side of the house. Throughout this section, you develop a C++ rendering engine and incorporate the shaders you authored in [Part II](#). [Chapters 10–14](#) introduce the core components of the engine: the game loop, time, components, and Windows and DirectX initialization. We also cover mouse and keyboard input, cameras, and text rendering. In [Chapter 15](#), you dive into the topic of 3D models: asset loading and model rendering. And in [Chapter 16](#), you develop a flexible effect and material system to integrate your shaders. [Part III](#) ends with a chapter on CPU-side structures for directional, point, and spot lights.
- **Part IV, “Intermediate-Level Rendering Topics,”** raises the bar a bit and moves to intermediate-level rendering topics. The section begins with a discussion of post-processing techniques (effects typically applied to the entire scene after its initial rendering). This includes shaders for color filtering, Gaussian blurring, bloom, and distortion mapping. In [Chapter 19](#), you implement systems for projective texture mapping and shadow mapping. Then in [Chapter 20](#), you develop a skeletal animation system for importing and rendering animated models. [Chapter 21](#) details geometry and tessellation shaders; you implement a point sprite shader and explore hardware tessellation, a powerful addition to the DirectX 11 graphics pipeline. The

book ends with a survey of additional topics in modern rendering, including rendering optimization, deferred rendering, global illumination, compute shaders, and data-driven engine architecture.

## Prerequisites

This book has no expectation that you are already a game or graphics developer, nor does it expect you to be fluent in 3D mathematics. It simply requires you to be interested in graphics programming and already be familiar with the C++ programming language. If you are an experienced programmer but you are coming from a different language, you may have no trouble with the material in [Parts III](#) and [IV](#). However, there is no concerted effort to discuss C++ syntax.

Furthermore, all code samples are provided for the Microsoft Windows operating system and are packaged for Visual Studio 2012 or 2013. These samples require a graphics card that supports (at least) Shader Model 4. Some of the samples (particularly the demonstrations on compute shaders and tessellation) require a graphics card that supports Shader Model 5.

## Companion Website

This book has a companion website at <http://www.varcholik.org/>. There you'll find all code samples and errata, along with a forum for questions and discussion about the book.

## Conventions in This Book

This book uses a number of conventions for source code, notes, and warnings.

### Note

When something needs additional explanation, it is called out in a “note” that looks like this.

### Warning: Warnings Look Like This

A “warning” points out something that might not be obvious and could cause problems if you did not know about it.

When code appears inside the text, it looks like this.

You will also find exercises at the end of [Chapters 3–22](#) (all but the first two chapters). These exercises reinforce the material discussed in the text and encourage you to experiment.

# Part I: An Introduction to 3D Rendering

[1 Introducing DirectX](#)

[2 A 3D/Math Primer](#)

[3 Tools of the Trade](#)

# Chapter 1. Introducing DirectX

**DirectX is a set of APIs for developing game-or graphics-related applications on Microsoft platforms, including Windows, Windows Phone, Xbox 360, and the new Xbox One. DirectX has been evolving since the mid-1990s and is at the leading edge of modern graphics development. Direct3D is the 3D graphics API within DirectX, and this book focuses its attention there. DirectX also includes systems for 2D graphics, input, audio, text rendering, and general-purpose GPU programming. This chapter provides an overview of DirectX and the DirectX 11 graphics pipeline.**

## A Bit of History

DirectX version 1.0 was released in September 1995, just after the launch of Windows 95. Previously, games were primarily developed on MS-DOS. Microsoft's release was an attempt to move game developers onto the new operating system. Few early adopters of DirectX stepped forward, a situation exacerbated by rapid-fire releases of the library. By February 2000, Microsoft had produced six additional major releases of DirectX and as many minor revisions. But the library steadily improved, and in 2001, Microsoft released its first game console, the Xbox, which supported a revision of DirectX 8. By that time, DirectX had gained a strong following of developers and Microsoft had emerged as a leader in the game and graphics space.

During this same period, 3D graphics hardware was in a similar state of rapid advancement. Consider that, before the mid-1990s, low-cost, consumer-oriented 3D graphics hardware simply didn't exist. Early 3D graphics cards offered fixed-function APIs—feature sets that were specific to the graphics card and could not be modified or extended by a graphics developer. With the 2001 release of the Xbox, the nVidia GeForce 3, and DirectX 8, developers were first widely introduced to programmable shaders. Programmable shaders enable developers to manipulate a 3D object as it passes through the graphics processing unit (GPU), also known as a graphics card, and define its output pixel by pixel. Under DirectX 8, shaders were written in assembly language, but in 2002, with the release of DirectX 9, Microsoft introduced the High-Level Shading Language (HLSL), a C-styled programming language for writing shaders.

In November 2005, Microsoft launched the Xbox 360 and a console cycle that lasted until November 2013 (the longest such cycle since the introduction of game consoles). The Xbox 360 employed a flavor of DirectX 9. A year later, in November 2006, Microsoft released Windows Vista and, with it, DirectX 10. However, during this time, consoles dominated the video game market, and the Xbox 360 supported DirectX 9 only. Thus, few developers embraced DirectX 10, and those who did were limited to the PC platform for DirectX 10 features. Additional factors that likely suppressed large-scale adoption of DirectX 10 were the major changes between DirectX 9 and DirectX 10 and the poor adoption of Windows Vista.

Windows 7 launched in July 2009, followed shortly by DirectX 11. This latest release represented relatively minor changes from DirectX 10 (compared with those of the earlier iteration). DirectX 11 introduced DirectCompute, Microsoft's API for general-purpose GPU (GPGPU) programming, tessellation support, and improved multithreading. Around this time, the PC game market began a resurgence in popularity, due in part to digital distribution platforms like Valve's Steam, as well as aging consoles. These factors almost certainly contributed to the increasing number of developers

who began adopting the updated libraries. Then in fall 2012, Microsoft released Windows 8 and Windows Phone 8, along with DirectX 11.1. This is the version this book focuses on, and all of Microsoft's most recent platforms, including the new Xbox One, support it.

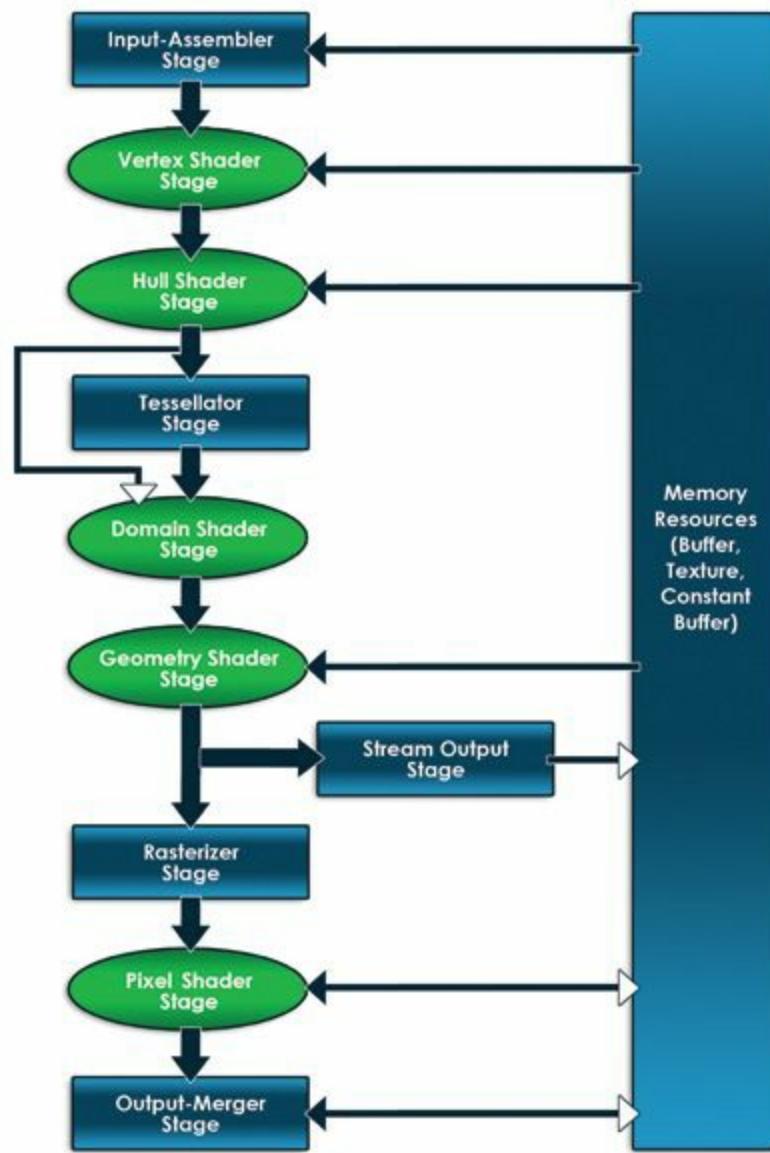
Although this is a book about DirectX, I would be remiss if I didn't mention OpenGL, a competing graphics library that evolved during this same time period. OpenGL is a cross-platform rendering API that Silicon Graphics Inc. first released in 1991. As of this writing, the latest version of OpenGL is the 4.4 specification (released in July 2013), and the Kronos Group manages the API. Although they have significant differences in design, modern OpenGL and DirectX generally support the same rendering capability. Thus, the choice of DirectX or OpenGL is largely a question of platform. DirectX is specific to Microsoft platforms, and graphics vendors widely support it, given the dominance of the Microsoft Windows operating system on desktop computers. OpenGL, on the other hand, is not specific to a particular platform and has gained wide adoption in the mobile development space.

And so, with a bit of history under our belt, we begin.

## The Direct3D 11 Graphics Pipeline

In your computer are generally two processors you'll write code for: the central processing unit (CPU) and the GPU. These components have entirely different architectures and instruction sets. In graphics programming, you write software for both—in a general-purpose language, such as C++, for the application (CPU) side and in HLSL for the GPU side. DirectX is the bridge between the systems. Most texts on the subject of graphics programming focus on either the CPU side or the GPU side, yet they are very much intertwined. In this book, you learn about both.

Direct3D is the API within DirectX that we're primarily concerned with. Put simply, Direct3D is the system you use to draw 3D graphics, and it defines a sequence of steps you use to present graphics to the screen. These steps are known as the Direct3D graphics pipeline (see [Figure 1.1](#)). In this figure, the arrows depict the flow of data from one stage to the next. The large rectangle spanning the right of the figure designates resources within GPU memory, and bi-directional arrows indicate read/write capability between a stage and these resources. Stages that are programmable (using HLSL) are shown as rounded rectangles. The following sections describe each stage in the pipeline.



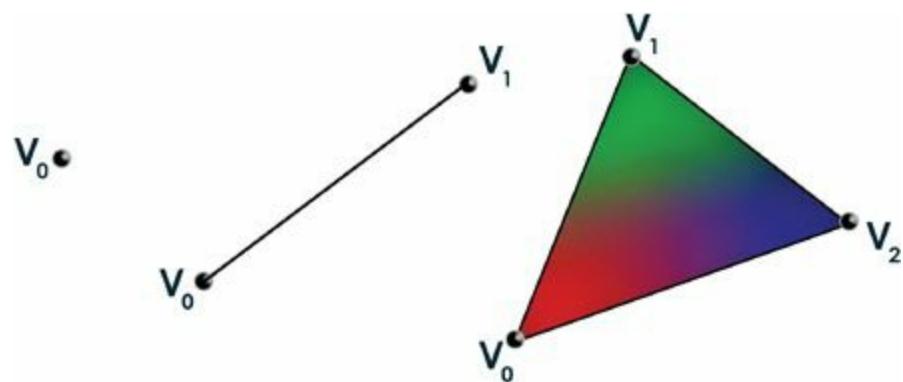
**Figure 1.1** The Direct3D 11 graphics pipeline.

## The Input-Assembler Stage (IA)

The input-assembler stage is the entry point of the graphics pipeline, where you supply vertex and index data for the objects you want to render. The IA stage “assembles” this data into primitives (such as points, lines, and triangles) and sends the corresponding output to the vertex shader stage. So what exactly is a vertex?

### Vertex Buffers

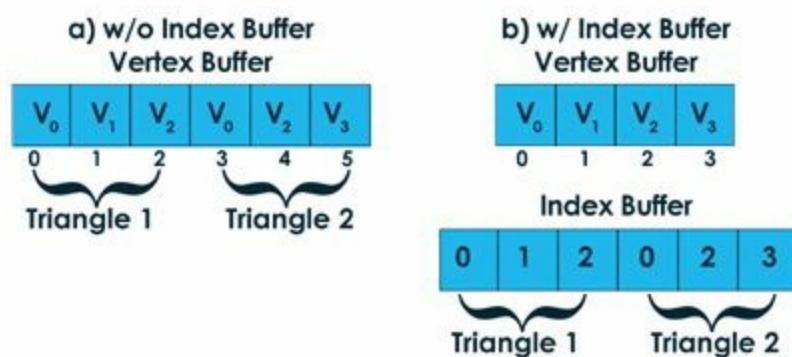
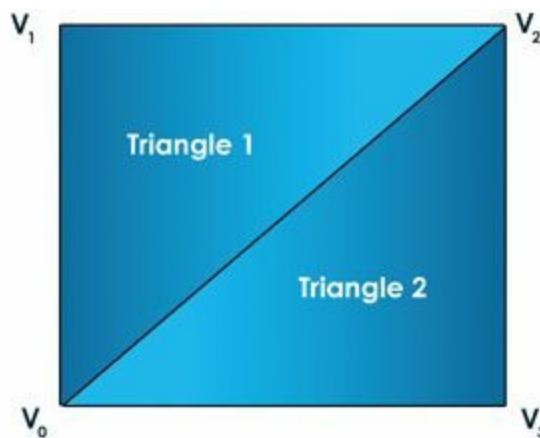
A vertex is at least a position in three-dimensional space. When considering a line, a single vertex marks one of the endpoints of the line: one of three points for a triangle (see [Figure 1.2](#)). But I say a vertex is “at least” a position because it’s often much more than that. A vertex might also contain a color, a normal (useful for lighting calculations), texture coordinates, and more. All this data is made available to the input-assembler stage through a **vertex buffer**. Aside from a position, Direct3D leaves the definition of a vertex entirely in the hands of the programmer. You define what your vertices contain and then inform Direct3D of the vertex format through an **input layout**. [Part III, “Rendering with DirectX,”](#) describes the specific calls to set up vertex buffers and input layouts; all you need to know right now is the terminology.



**Figure 1.2** 3D primitives: point (left), line (middle), triangle (right).

## Index Buffers

**Index buffers** are the (optional) second type of input into the IA stage. **Indices** identify specific vertices within your vertex buffer and are employed to reduce duplication of reused vertices. Consider the following scenario: You want to render a rectangle (more generically, a quadrilateral). A quad can be minimally defined by four vertices. However, Direct3D doesn't support quads as a primitive type (but doesn't particularly need to because all polygonal shapes can be decomposed into triangles). To render the quad, you can split it into two triangles composed of three vertices each (see [Figure 1.3](#)). So now you have six total vertices instead of four, with two of the vertices duplicates. With an index buffer, you can instead specify just the four unique vertices and six indices into the vertex buffer.



**Figure 1.3** Vertices and indices for a 3D quadrilateral.

Now, you might be thinking, “How does *adding* an index buffer *decrease* the total size of the data I’m using?” Well, if your vertex data is composed of just a 3D position (x, y, z), where each component is

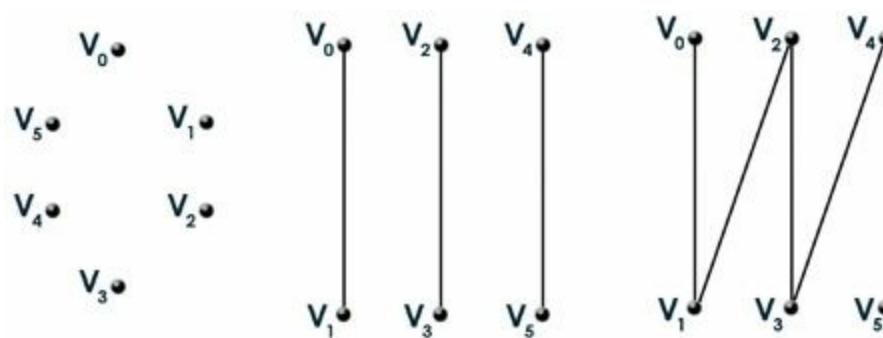
a 32-bit floating point number (4 bytes per component), then each vertex is 12 bytes. Without an index buffer, your vertex buffer will be filled with 72 bytes of data ( $6 \text{ vertices} \times 12 \text{ bytes/vertex} = 72 \text{ bytes}$ ). With an index buffer, your vertex buffer becomes 48 bytes ( $4 \text{ vertices} \times 12 \text{ bytes/vertex} = 48 \text{ bytes}$ ). If you use 16-bit integers for your indices, then your index buffer is a total of 12 bytes ( $6 \text{ indices} \times 2 \text{ bytes/index} = 12 \text{ bytes}$ ). Combined, the total size of the vertex and index buffers is 60 bytes (48-byte vertex buffer + 12-byte index buffer). You see hardly any savings at all. But consider passing additional vertex data—perhaps a 16-byte color, a 12-byte normal, and 8 bytes of texture coordinates. That's an additional 36 bytes per vertex saved for each vertex shared between two triangles. This really adds up when loading meshes with hundreds or thousands of vertices. Furthermore, you're not concerned with just memory—you also need to consider the bus between the CPU and the GPU. Every bit of data that you employ must be transferred from the application to the video card over the graphics bus (such as PCI Express). This bus is slow (compared to CPU to RAM or GPU to VRAM), so reducing the amount of data you transmit is vital.

## Primitive Types

When you supply a vertex buffer to the input-assembler stage, you must define the **topology** of those vertices—that is, how the pipeline interprets those vertices. Direct3D supports the following basic primitive types:

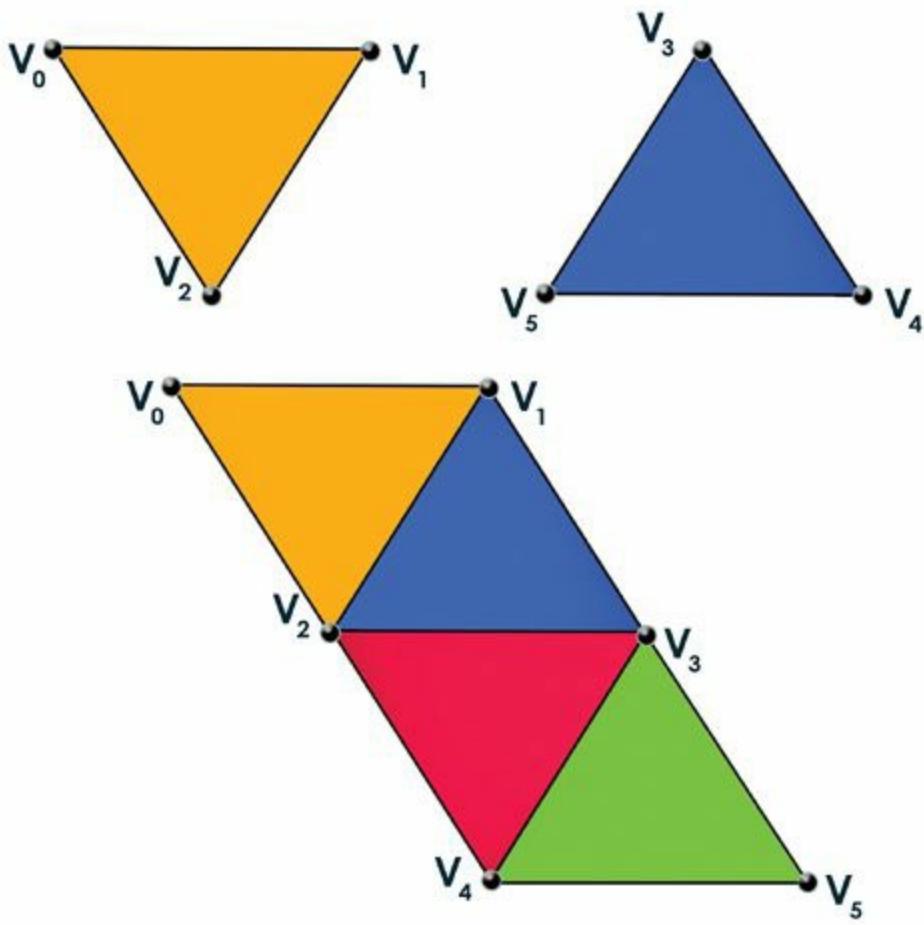
- Point list
- Line list
- Line strip
- Triangle list
- Triangle strip

A **point list** is a collection of vertices that are rendered as individual, unconnected points. A **line list**, by contrast, connects pairs of points into line segments. However, these line segments are not connected as they are in a **line strip**. Furthermore, a line strip specifies its vertices not in pairs, but as a connect-the-dots-style sequence of points. [Figure 1.4](#) illustrates these topologies.



**Figure 1.4** Point list (left), line list (middle), and line strip (right).

A **triangle list** is the most common topology we'll be working with. In a triangle list, each set of three vertices is interpreted as an isolated triangle. Any shared vertices are repeated (notwithstanding our discussion of index buffers). By contrast, a **triangle strip** interprets vertices as a series of connected triangles in which shared vertices are not repeated. [Figure 1.5](#) depicts both topologies.



**Figure 1.5** Triangle list (top) and triangle strip (bottom).

### Triangle Strip Vertex Winding Order

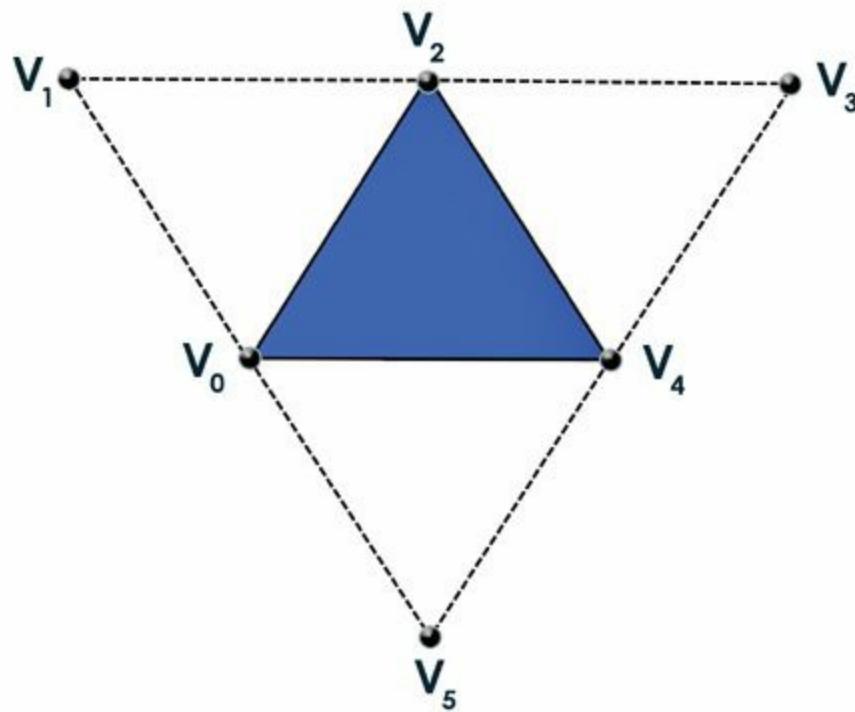
Note that, for the triangle strip illustration in [Figure 1.5](#), the numbering of the vertices does not make clear how groups of three vertices are rendered. In the figure, four triangles are formed out of the six total vertices. Direct3D draws the triangles in the following order:

- Triangle 1: V0, V1, V2
- Triangle 2: V1, V3, V2
- Triangle 3: V2, V3, V4
- Triangle 4: V3, V5, V4

Intuitively, you might group the vertices in ascending numeric order, but that's not what's happening here—the even-numbered triangles appear to be out of order. Direct3D **winds** triangles in clockwise order to aid in **backface culling**. [Part II](#), “[Shader Authoring with HLSL](#),” discusses this further.

### Primitives with Adjacency

Since version 10, Direct3D has included support for primitives with **adjacency data**. For primitives with adjacency, you specify not only vertices for the base primitive, but also adjacent vertices “surrounding” the base primitive. [Figure 1.6](#) illustrates this concept.



**Figure 1.6** Triangle list with adjacency.

## Control Point Patch Lists

Direct3D 11 added **control point patch lists** as a supported topology for use with the tessellation stages of the pipeline. We discuss patch lists in [Chapter 21, “Geometry and Tessellation Shaders.”](#)

## The Vertex Shader Stage (VS)

The vertex shader stage processes the primitives interpreted by the input-assembler stage. It does this processing on a per-vertex basis. This is the first stage in the pipeline that is programmable. In fact, a vertex shader must *always* be supplied to the VS stage. So what exactly is a shader?

A shader is a small program—a function, if you’d like—that *you* write and the GPU executes. A vertex shader operates on every vertex passing through the pipeline (the input to the shader), performing a series of instructions and passing output to the next active stage. As mentioned before, the input to the vertex shader is at least the position of the vertex. Generally, the vertex shader transforms the vertex in some way and outputs the modified or newly computed data. [Listing 1.1](#) shows perhaps the simplest vertex shader we can create.

### Listing 1.1 Your First Vertex Shader

[Click here to view code image](#)

---

```
float4 vertex_shader(float3 objectPosition : POSITION) :
SV_Position
{
    return mul(float4(objectPosition, 1), WorldViewProjection);
}
```

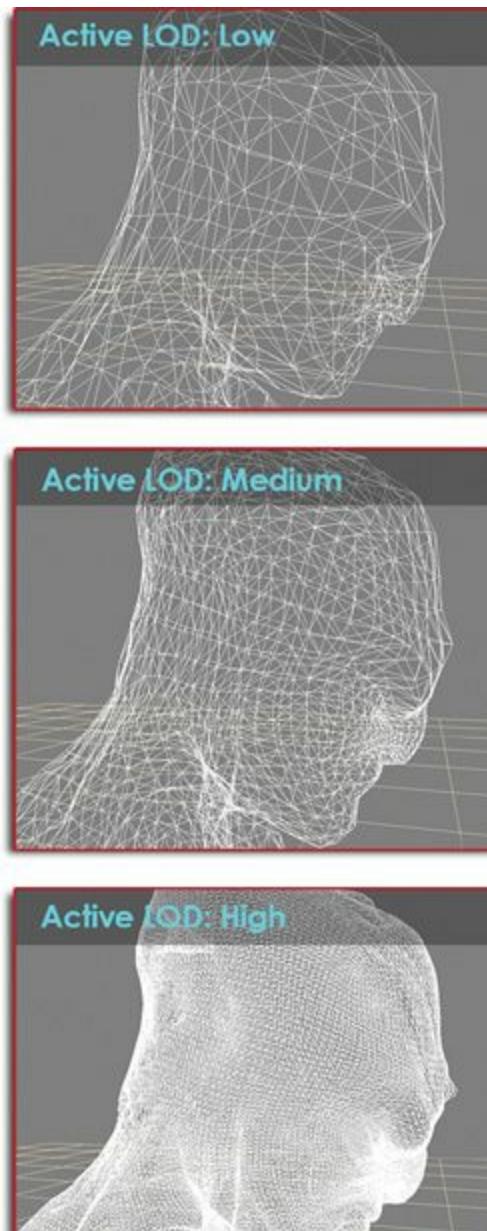
---

## Note

Did you say *simple*? Don't stress over the syntax of this vertex shader. As a C/C++ programmer, this should look vaguely like a function, but clearly there's some special sauce in the mix. We cover all of this syntax in [Part II](#).

## Tessellation Stages

New to Direct3D 11, hardware tessellation is a process that adds detail to objects directly on the GPU. Generally, more geometric detail (that is, more vertices) yields a better-looking render. Consider the images in [Figure 1.7](#).



**Figure 1.7** A 3D model with low, medium, and high levels of detail. (*3D Model by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

This image shows a 3D model with low, medium, and high levels of detail (LODs). Traditionally, LODs are authored by an artist, and a particular LOD is rendered based on the distance of the object from the camera.

## Note

Little reason exists for rendering a high-resolution object that is far away because all that added detail is lost from the perspective of the viewer. You choose a level of detail corresponding to the object's distance—the farther away, the lower the detail.

The fewer vertices your vertex shaders have to process, the faster your rendering is.

With a traditional LOD system, you have a fixed amount of detail (polygon count) in your models. Hardware tessellation enables you to subdivide an object dynamically and without the cost of additional geometry passed to the input-assembler stage. This allows for a dynamic LOD system and less utilization of the graphics bus (both good things). In Direct3D 11, these three stages correspond to tessellation:

- The hull shader stage (HS)
- The tessellator stage
- The domain shader stage (DS)

The hull and domain shader stages are programmable; the tessellator stage is not. We cover the topic of tessellation in more detail in [Chapter 21](#).

## The Geometry Shader Stage (GS)

Unlike vertex shaders, which operate on individual vertices, geometry shaders operate on complete primitives (such as points, lines, and triangles). Moreover, geometry shaders have the capability to add or remove geometry from the pipeline. This feature can provide some interesting applications. For example, you could create a particle effects system, in which a single vertex represents each particle. In the geometry shader, you can create quads around those central points, to which you can map textures. Such objects are commonly known as **point sprites**.

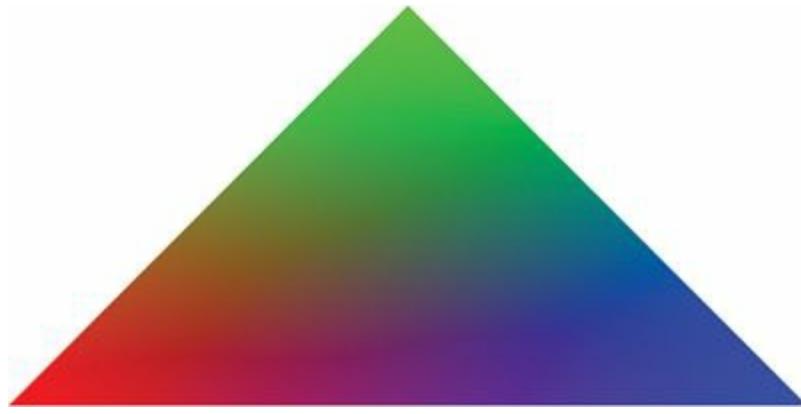
Connected to the GS stage is the **stream-output stage (SO)**. This stage streams in the vertices output from the geometry shader and stores them in memory. For multipass rendering, this data can be read back into the pipeline in a subsequent pass, or the CPU can read the data. As with the tessellation stages, the geometry shader stage is optional. We discuss geometry shaders and multipass rendering in more detail in [Part IV, “Intermediate-Level Rendering Topics.”](#)

## The Rasterizer Stage (RS)

Up to this point in the pipeline, we've mostly been discussing vertices and the interpretation of those vertices into primitives. The rasterizer stage converts those primitives into a raster image, otherwise known as a bitmap. A raster image is represented as a two-dimensional array of pixels (colors) and generally makes up some area of your computer screen.

The rasterizer stage determines what, if any, pixels should be rendered and passes those pixels to the pixel shader stage. Along with the pixels to render, the rasterizer passes per-vertex values interpolated across each primitive. For example, a triangle primitive has three vertices, each of which contains at least a position and potentially additional data such as color, normal, and texture coordinates. This vertex data is interpolated for the *in-between* pixels the rasterizer computes. [Figure 1.8](#) illustrates this concept for vertex colors. In this image, the three points of the triangle are supplied with a red, green, and blue vertex color, respectively. Notice how the pixels within the triangle change color relative to their proximity to the three vertices. The rasterizer stage produces those

interpolated colors.



**Figure 1.8** Rasterizer interpolation for a triangle with red, green, and blue colors specified for the three vertices.

## The Pixel Shader Stage (PS)

Although technically optional, you almost always provide a shader to the pixel shader stage. This stage executes your shader code against each pixel input from the rasterizer stage and (typically) outputs a color. This gives the programmer control over every pixel that's rendered to the screen. The pixel shader uses interpolated per-vertex data, global variables, and texture data to produce its output. [Listing 1.2](#) presents a shader that outputs pure red for each pixel.

### **Listing 1.2** Your First Pixel Shader

[Click here to view code image](#)

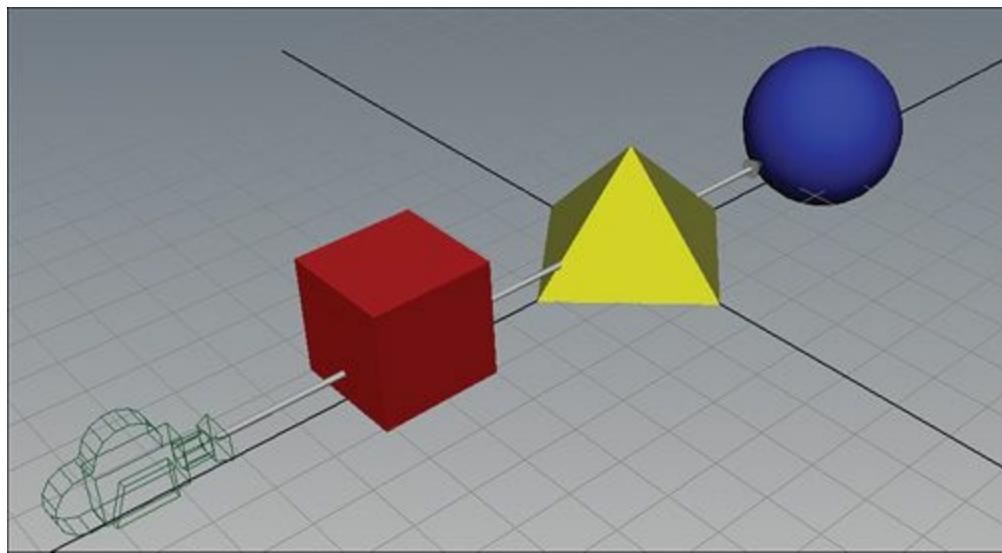
```
float4 pixel_shader() : SV_Target
{
    return float4(1, 0, 0, 1);
}
```

## The Output-Merger Stage (OM)

The output-merger stage produces the final rendered pixel. This stage isn't programmable (you don't write a shader for the OM stage), but you do control how it behaves through customizable pipeline states. The OM stage generates the final pixel through a combination of these states, the output from the pixel shader stage, and the existing contents of the render target. This allows for, among other interesting effects, the implementation of transparent objects through **color blending**. We discuss blending in [Chapter 8, “Gleaming the Cube.”](#)

The OM stage also determines which pixels are visible in the final render through processes known as **depth testing** and **stencil testing**. Depth testing uses data that has previously been written to the render target to determine whether a pixel should be drawn. Consider the scenario in [Figure 1.9](#). In this figure, a number of objects are closer to the camera than others, but they otherwise represent the same screen space. These objects **occlude** the ones behind them either entirely or in part. Depth testing makes use of the distance between the object and the camera for each corresponding pixel written to the render target. Most commonly, if the pixel already in the render target is closer to the

camera than the pixel being considered, the new pixel is discarded.



**Figure 1.9** A scene illustrating depth testing, with some objects occluding others.

Stencil testing uses a mask to determine which pixels to update. This is conceptually similar to a cardboard or plastic stencil you might use to paint or print a design on a physical surface. We discuss depth and stencil testing in more detail in [Part III, “Rendering with DirectX.”](#)

### Note

The rasterizer stage also plays a role in determining which pixels are rendered to the screen in a process known as clipping. Any pixels that the rasterizer determines to be “off-screen” aren’t sent to the pixel shader stage and are discarded from further processing in the pipeline.

## Summary

In this chapter, you discovered that DirectX is a set of APIs that covers a range of game-related topics, including 2D and 3D rendering, input, audio, and even general-purpose GPU programming. You learned that DirectX has been evolving since the mid-1990s and is now in its 11th major revision. The chapter provided an overview of the Direct3D graphics pipeline, from the input-assembler stage to the output-merger stage. You were introduced to topics including vertex and index buffers, primitive topologies, vertex and pixel shaders, and tessellation.

This is just the beginning—the coming chapters expand upon all these topics and much more.

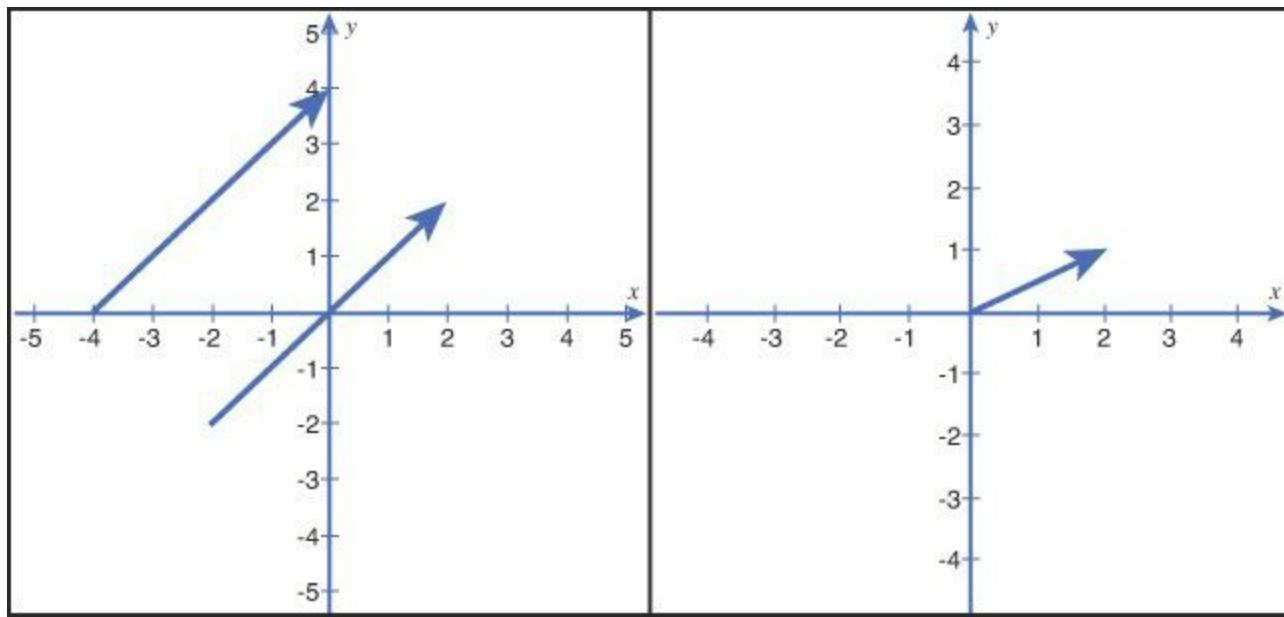
# Chapter 2. A 3D/Math Primer

This chapter is an introduction to the mathematics behind 3D graphics. And in the immortal words of Douglas Adams (of *Hitchhiker's Guide to the Galaxy*), “DON’T PANIC!” We discuss vectors, matrices, coordinate systems, transformations, and the DirectX Math library. Even if linear algebra is old hat to you, we encourage you to review the material specific to DirectX. Otherwise, take a deep breath, hold on, and don’t forget your towel.

## Vectors

You might recall the definitions of scalars and vectors from your high school physics class. Scalars are quantities with just a magnitude (a numerical value); vectors are quantities described by both magnitude and direction. Vectors wear multiple hats in 3D graphics—they describe velocities, forces, directions, or simply positions in 3D space.

Geometrically, you can designate a vector as an arrow; the vector’s magnitude is its length and the line segment points along the direction of the vector. Under this definition, the tail of a vector has no fixed location. Thus, two vectors are equal if merely their lengths and their directions are the same. However, if you fix the tail at the origin of a coordinate system, the vector can describe a position specified by the number of components matching the dimensions of the coordinate system. In 3D space, you describe a vector by x, y, and z components. A position of (2, 5, 8) describes a vector whose head is located 2 units from the origin along the x-axis, 5 units along the y-axis, and 8 units along the z-axis. [Figure 2.1](#) illustrates these concepts in 2D.

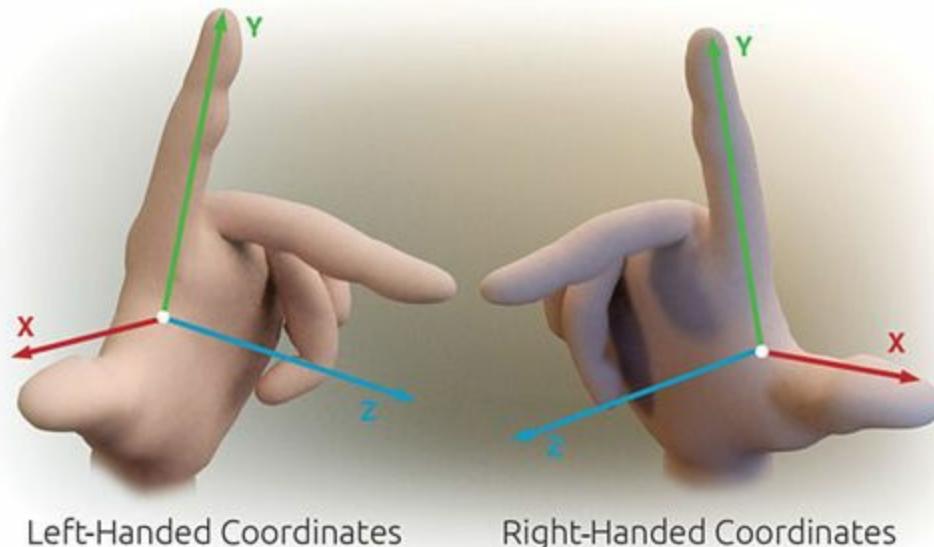


**Figure 2.1** An illustration of vectors. To the left, the two vectors are equivalent, although their tails are drawn at different locations. To the right, a vector is rooted at the origin of a 2D coordinate system and describes a position.

## Left-Handed and Right-Handed Coordinate Systems

The illustration in [Figure 2.1](#) shows a 2D Cartesian coordinate system, where the x-axis runs horizontally and the y-axis runs vertically, with positive numbers to the right and up, respectively. In 3D space, the z-axis is orthogonal (perpendicular) to the x- and y-axes, with positive numbers running

into or out of the screen, depending on the **handedness** of the coordinate system. In a left-handed coordinate system, positive z-values move into the screen, and they move out of the screen for a right-handed system. A useful mnemonic for handedness is to hold your right hand in front of you, pointing your thumb toward the positive x-axis and your index finger toward the positive y-axis. The direction of your middle finger (toward you) indicates the positive z-values headed out of the screen; conversely, the opposite is true for your left hand. [Figure 2.2](#) illustrates this technique.



**Figure 2.2** A mnemonic for 3D coordinate system handedness. (By Primalshell [CC-BY-SA-3.0 (<http://creativecommons.org/licenses/by-sa/3.0>)] via Wikimedia Commons.)

## Basic Vector Operations

Two vectors are added and subtracted **component wise**—that is, the operands are drawn from matching components of the two vectors. For example, if you define two vectors  $a = (a_x, a_y, a_z)$  and  $b = (b_x, b_y, b_z)$ , and perform addition or subtraction between them to produce vector  $c$ , then these operations are performed according to [Table 2.1](#).

Description	Operation	Result
Addition	$a + b$	$c = (a_x + b_x, a_y + b_y, a_z + b_z)$
Subtraction	$a - b$	$c = (a_x - b_x, a_y - b_y, a_z - b_z)$

**Table 2.1** Basic Vector Operations

Likewise, multiplication and division are performed component wise as are basic operations between a scalar and a vector. The following is an example of scalar multiplication:

$$4 * a = (4a_x, 4a_y, 4a_z)$$

## Vector Length

The **length** (or **magnitude**) of a vector is calculated using the Pythagorean Theorem. Specifically, you use the following equation (for a three-dimensional vector):

$$\|a\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

A vector whose length is exactly 1 is called a **unit vector**. Such vectors are useful for situations that require only the direction of the vector, not its magnitude. When you force a vector to have a length of 1, it's called **normalizing** the vector; it is accomplished by dividing the vector by its magnitude. In equation form:

$$\hat{a} = \frac{a}{\|a\|} = \left( \frac{a_x}{\|a\|}, \frac{a_y}{\|a\|}, \frac{a_z}{\|a\|} \right)$$

## Dot Product

The **dot product** between two vectors is the sum of the products of the matching components. In equation form:

$$a \cdot b = (a_x * b_x) + (a_y * b_y) + (a_z * b_z)$$

This produces a scalar value, hence the dot product is also referred to as the **scalar product** (or **inner product**). Considering the definition of vector length, you can use the square root of the vector dotted with itself to compute the vector's magnitude.

Geometrically, the dot product describes an angle between two vectors. In equation form:

$$a \cdot b = \|a\| * \|b\| * \cos(\theta)$$

Where  $\theta$  is the angle between vectors  $a$  and  $b$ . If both vectors are unit length, the dot product reduces to:

$$a \cdot b = \cos(\theta)$$

From this equation, you can observe the following:

- If  $a \cdot b > 0$ , then the angle between the vectors is less than 90 degrees.
- If  $a \cdot b < 0$ , then the angle between the vectors is greater than 90 degrees.
- If  $a \cdot b = 0$ , then the vectors are orthogonal.

As you'll see in coming chapters, the dot product has a variety of applications in computer graphics. For instance, the dot product is useful in lighting calculations to determine the angle between the surface and a light source. We discuss such applications in detail in [Part II, “Shader Authoring with HLSL.”](#)

## Cross Product

Another useful vector operation is the **cross product**. The cross product of two vectors produces a third vector that is orthogonal to the two other vectors. The equation for cross product is:

$$a \times b = (a_y * b_z - a_z * b_y, a_z * b_x - a_x * b_z, a_x * b_y - a_y * b_x)$$

You can use the cross product, for example, to compute the surface normal of a triangle (which describes the direction the triangle faces).

## Matrices

An  $m \times n$  matrix is a two-dimensional array of numbers with  $m$  rows and  $n$  columns. A  $4 \times 1$  matrix, for example, has four rows and one column. A  $2 \times 3$  matrix has two rows and three columns. Such matrices are notated as follows:

$$A = \begin{bmatrix} 5 \\ 2 \\ 1 \\ 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 4 & 8 \\ 2 & 3 & 9 \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \dots & \dots & \dots & \dots \\ C_{m1} & C_{m2} & \dots & C_{mn} \end{bmatrix}$$

Note the matrix  $C$ , which depicts the subscript notation for identifying an element within the matrix. Matrices with only a single row or column are sometimes referred to as **row vectors** or **column vectors**. Matrices with the same number of rows and columns are known as **square matrices**. In 3D graphics,  $4 \times 4$  square matrices are used extensively.

## Basic Matrix Operations

You can perform a number of basic arithmetic operations on matrices. As with vectors, addition and subtraction are performed component wise. Thus, the two matrices must have the same number of rows and columns. Scalar multiplication is also performed component wise, with the scalar multiplied with each element of the matrix. However, matrix-to-matrix multiplication is a bit different.

## Matrix Multiplication

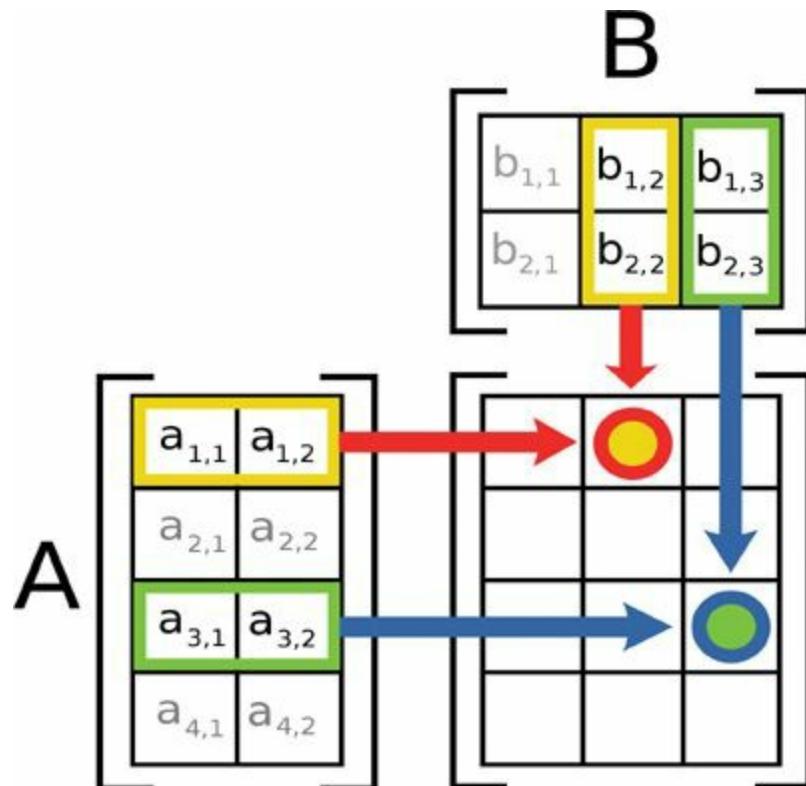
For matrix multiplication, a matrix with  $n$  columns can be multiplied only by a matrix with  $n$  rows. Each element in the resulting matrix is computed as the dot product between the rows in the first matrix and the corresponding columns in the second matrix. In equation form, if you consider each row of matrix  $A$  as a row vector and each column of matrix  $B$  as a column vector, you can define the elements of the product  $C$  as follows:

$$a_i = (A_{i1}, A_{i2}, \dots, A_{in})$$

$$b_j = \begin{pmatrix} B_{1j} \\ B_{2j} \\ \dots \\ B_{nj} \end{pmatrix}$$

$$C_{ij} = a_i \bullet b_j$$

[Figure 2.3](#) illustrates this process.



**Figure 2.3** Matrix multiplication. (By Bilou [GFDL (<http://www.gnu.org/copyleft/fdl.html>), CC-BY-SA-3.0 (<http://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons.)

By way of example, consider the following matrices:

$$A = \begin{bmatrix} 1 & 4 & 8 \\ 2 & 3 & 9 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 0 & 6 \\ -1 & 5 & 9 \end{bmatrix}$$

Matrix  $A$  has two rows and three columns (a  $2 \times 3$  matrix). Matrix  $B$  has three rows and three columns (a  $3 \times 3$  matrix). The number of rows in  $B$  is equal to the number of columns in  $A$ , so multiplication is permitted. The resulting product is a  $2 \times 3$  matrix and is computed as:

$$A * B = \begin{bmatrix} (1,4,8) \bullet (2,4,-1) & (1,4,8) \bullet (1,0,5) & (1,4,8) \bullet (3,6,9) \\ (2,3,9) \bullet (2,4,-1) & (2,3,9) \bullet (1,0,5) & (2,3,9) \bullet (3,6,9) \end{bmatrix} = \begin{bmatrix} 10 & 41 & 99 \\ 7 & 47 & 105 \end{bmatrix}$$

Note that matrix multiplication is not commutative (you can't reverse the operands and yield the same value). Indeed, it's common that multiplication defined for  $A * B$  is undefined for  $B * A$ , as is the case for a  $3 \times 3$  matrix multiplied by a  $2 \times 3$  matrix.

## Transposition

The **transpose** of a matrix is found by reflecting its elements over its main diagonal. For a square matrix, this is easy to envision, as the following example demonstrates:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

Another way to think of transposing a matrix is swapping the rows and columns. This approach might be simpler to imagine, particularly for row vectors and column vectors, or nonsquare matrices. For example:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$C^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

## Row-Major and Column-Major Order

When dealing with matrices, you must consider how they are stored in a computer's memory. Direct3D stores matrices in **row-major order**—that is, if stored in contiguous memory, the rows are

laid out one after another. This is also the format of multidimensional arrays in the C programming language. The following example shows a  $2 \times 3$  matrix stored contiguously in row-major order.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

*Row - Major : 1 2 3 4 5 6*

Conversely, the same matrix, written in **column-major order**, would appear in memory as:

*Column - Major: 1 4 2 5 3 6*

Converting a row-major matrix to a column-major matrix (or vice versa) involves transposing it. How a matrix is stored in memory can have performance implications, and various library methods depend on a particular order. We discuss this further in [Chapter 4, “Hello, Shaders!”](#)

## The Identity Matrix

The **identity matrix** is a special square matrix that has ones along the main diagonal and zeroes everywhere else. For example, a  $4 \times 4$  identity matrix is defined as:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The identity matrix is essentially the number one, with respect to matrix multiplication. In other words, multiplying a matrix by the identity matrix results in the original matrix.

## Transformations

As mentioned previously, vectors describe (among other things) directions and positions in 3D space. For instance, the positions of the vertices of your 3D triangles are stored in vectors, as are their surface normals. However, these vectors are unlikely to remain static throughout their lives inside your graphics applications. Your 3D objects will move, grow, shrink, and turn within a scene. More formally, displacement is known as **translation**, growing and shrinking is **scaling**, and turning is **rotation**. Such operations are known as **transformations**. Furthermore, to display your objects to the screen, they will be considered from multiple frames of reference—coordinate systems. You’ll use matrices to transform positions and directions from one coordinate system to another, as well as to apply translation, scaling, and rotation.

## Homogeneous Coordinates

Before diving into the specific transformation matrices, it’s important to note that we’ll be using **homogeneous coordinates**. This coordinate system enables you to incorporate translation into a  $4 \times 4$  matrix along with scaling and rotation. Matrix multiplication is associative, so a single  $4 \times 4$  matrix can encapsulate all these transformations. With homogeneous coordinates, each position has four components: x, y, z, and w. The xyz coordinates are the normal 3D coordinates, and the w component is the value 1 for positions and 0 for directions. As you’ll see, the w = 0 for direction vectors nullifies translation, which makes sense because direction vectors have no location in 3D space.

## Scaling

To transform a vector, you perform a vector-to-matrix multiplication, which is just a matrix multiplication against a row vector or column vector. By way of example, the 3D scaling matrix is defined as:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The values along the main diagonal represent the scaling factors for the x-, y-, and z-axes. If these values are the same, the vector will be scaled uniformly. Note the similarity to the identity matrix, which is merely a scaling matrix with ones along the diagonal.

Following is an example of transforming a position vector by a nonuniform scaling matrix. The position stored in  $A$  is  $(-4, 2, 6)$ . The fourth column (the w component with the value 1) is necessary because the scaling matrix has four rows.

$$A = \begin{bmatrix} -4 & 2 & 6 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & .5 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A * S = \begin{bmatrix} -8 & 1 & 18 & 1 \end{bmatrix}$$

## Translation

The act of translation moves an object along the x-, y-, and z-axes. The translation matrix is defined as:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Next is an example of translating a position vector 10 units along the x-axis, 5 units along the y-axis, and 6 units along the negative z-axis.

$$A = \begin{bmatrix} -4 & 2 & 6 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 10 & 5 & -6 & 1 \end{bmatrix}$$

$$A * T = \begin{bmatrix} 6 & 7 & 0 & 1 \end{bmatrix}$$

Notice that if the w component of the vector was 0 (as if the vector represented a direction), the transformation would have no impact.

## Rotation

Rotation is a bit more complicated because each axis has a different rotation matrix.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Following is an example of rotating the position  $(1, 0, 0)$  90 degrees ( $\pi/2$  radians) counterclockwise around the z-axis.

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\pi/2) & \sin(\pi/2) & 0 & 0 \\ -\sin(\pi/2) & \cos(\pi/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A * R = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}$$

## Matrix Concatenation

Instead of applying each of these transforms individually, they can be concatenated into a single transformation matrix. For example, a scale, rotation, and translation matrix can be combined as:

$$W = S * R * T$$

Transforming a position vector by this concatenated matrix scales, rotates, and translates the vector in a single operation. However, the order of concatenation is important. Concatenation is performed from left to right, and matrix multiplication is not commutative. For example, suppose you were rendering Earth orbiting the sun. It has both a translation away from the sun and a rotation around it, and the concatenation should be performed in that order. However, if you also wanted to simulate Earth's axial rotation, you'd want to perform that rotation before the "orbit" transformations. In such a scenario, the complete concatenated matrix would appear as:

$$W = R_{\text{axial}} * T * R_{\text{orbit}}$$

You can concatenate any number of individual transforms; just be aware of the left-to-right ordering.

## Changing Coordinate Systems

In addition to scaling, translation, and rotation, you can use transformation matrices to move a vector from one coordinate system to another.

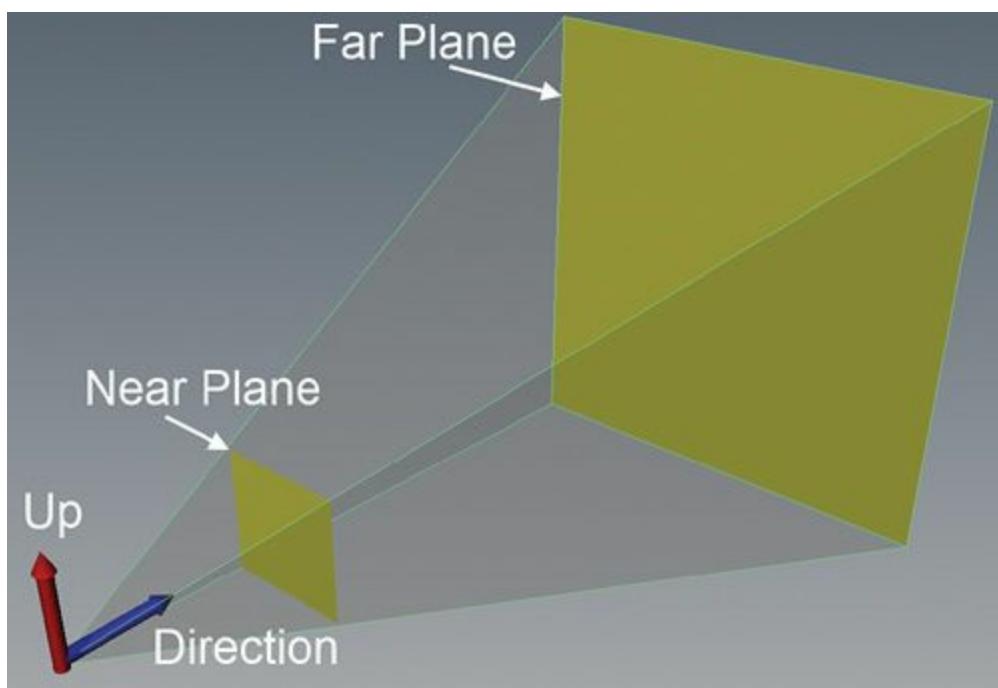
### Object Space and World Space

Typically, you will create your 3D models in a digital content creation package, such as Autodesk Maya or 3D Studio Max. You also will author a model in its own coordinate system, called **object space** (or **model space**). Your scenes might contain hundreds of individual models, but these objects are rarely authored as a whole. Instead, each object is stored in a separate file, and its vertices are relative to its own origin, with no relationship to another 3D model. For these objects to interact, they must share a common coordinate system—they must exist in the same world. Therefore, they must be transformed from object space to **world space**.

A **world transformation matrix** (or **world matrix**) is made from concatenated scale, translation, and rotation matrices. To apply the world matrix to a 3D model is to multiply each vertex in the object by the matrix.

### View Space

From world space, the object is then transformed into **view space** (or **camera space**), a coordinate system relative to a virtual camera. The properties of the camera define a volume within which objects are visible. This volume is known as the **view frustum** (see [Figure 2.4](#)). A **view matrix** is used to transform vectors from world space to view space and is commonly created through a combination of the camera's position (in world space), a vector describing where the camera is looking, and a vector describing which way is "up." You use the DirectX API to create a view matrix in [Part III, "Rendering with DirectX."](#)



**Figure 2.4** An illustration of the view frustum specified by the properties of a virtual camera.

## Projection Space

Object space, world space, and view space are all three-dimensional coordinate systems. However, computer screens are two-dimensional, and the **projection matrix** projects this 3D geometry onto a 2D display. In this stage, the illusion of depth is created—for example, objects that are closer to the screen appear larger than those farther away. This is known as **perspective projection**, as opposed to **orthographic projection**, in which all objects appear to be the same distance away from the camera, regardless of their actual positions.

The properties of the projection matrix help define the view frustum and include a **near plane** and **far plane**, a vertical **field of view**, and an **aspect ratio**. Objects between the camera’s position and the near plane are not rendered, nor are objects past the far plane. A good analogy is the tip of your nose (near plane) and the horizon (far plane). The field of view defines the extent of the frustum, and the aspect ratio is typically the width of the display (or window) divided by the height. As with the view matrix, the projection matrix is commonly constructed through DirectX API calls.

The transformations for world, view, and projection space are typically performed within the vertex shader, and you are responsible for them. You can apply each transformation individually, or you can concatenate the three matrices into what’s commonly referred to as the **world-view-projection matrix**, or **WVP**. After they’re multiplied by the projection matrix, the coordinates are in **projection space** (or **homogeneous clip space**).

## DirectXMath

**DirectXMath** is a SIMD-friendly C++ API from Microsoft that supports the mathematics common to graphics applications. **SIMD** stands for **Single Instruction Multiple Data** and provides for multiple data points to be operated on through a single CPU instruction. For example, on the Windows platform, SIMD instructions can operate on four 32-bit floats simultaneously. This offers significant performance gains. For example, if you added two vectors together, each with four components, you would ordinarily perform four add instructions. But with SIMD, the same addition can be performed

with a single instruction.

DirectXMath is distributed as part of the Windows SDK (we discuss this further in [Chapter 3, “Tools of the Trade”](#)). To use the library, simply include the `DirectXMath.h` file in your applications. All the implementation is included in the header file; it has no libraries to link against.

## Vectors

Vectors are supported through `XMVECTOR`, a 16-byte aligned data type mapped to SIMD registers. For best performance, all vector operations (for example, addition or subtraction) should be performed through `XMVECTOR` instances. However, this is an opaque data type, so individual vector components are inaccessible as structure members. Additionally, because this data type is 16-byte aligned, it’s not a good candidate for class member storage. For class storage, the types `XMFLOAT2`, `XMFLOAT3`, and `XMFLOAT4` are recommended instead. These types have accessible members for two, three, and four components (`x`, `y`, `z`, and `w`), respectively.

### Loading and Storing

To take advantage of SIMD instructions, you need to copy data out of `XMFLOAT*` instances and into `XMVECTOR` objects. You accomplish this with the following *load* methods:

[Click here to view code image](#)

```
XMVECTOR XMLoadFloat2 (const XMFLOAT2* pSource);  
XMVECTOR XMLoadFloat3 (const XMFLOAT3* pSource);  
XMVECTOR XMLoadFloat4 (const XMFLOAT4* pSource);
```

Conversely, you copy data out of `XMVECTOR` instances and into `XMFLOAT*` objects through the following *store* methods:

[Click here to view code image](#)

```
void XMStoreFloat2 (XMFLOAT2* pDestination, FXMVECTOR V);  
void XMStoreFloat3 (XMFLOAT3* pDestination, FXMVECTOR V);  
void XMStoreFloat4 (XMFLOAT4* pDestination, FXMVECTOR V);
```

Thus, the pattern for using vectors in DirectXMath is to store `XMFLOAT*` instances as class members and *load* them into local `XMVECTOR` objects for SIMD vector operations. If you need to save the output of a vector operation, you *store* the resulting `XMVECTOR` object back into an `XMFLOAT*` class member.

### Note

DirectXMath supports vectors with many other underlying data types, including byte, int, short, and half-float (16-bit floats). They all follow the same naming and calling conventions. For example, `XMBYTE4` stores four 8-bit signed values and has matching *load* and *store* methods: `XMLoadByte4()` and `XMStoreByte4()`.

## Calling Conventions

Note the `FXMVECTOR` type in the parameter lists of the *store* functions. This is one of four aliases used for `XMVECTOR`—the others are `GXMVECTOR`, `HXMVECTOR`, and `CXMVECTOR`. These aliases are used for optimal data layout and portability between platforms. On the x86 and Xbox 360

platforms, these are simply synonyms for XMVECTOR. For the x64 platform, these types are XMVECTOR references. The rules for passing XMVECTOR arguments follow:

- Use FXMVECTOR for the first three XMVECTOR input parameters.
- Use GXMVECTOR for the fourth XMVECTOR input parameter.
- Use HXMVECTOR for the fifth and sixth XMVECTOR input parameters.
- Use CXMVECTOR for any additional XMVECTOR input parameters.
- Use XMVECTOR\* or XMVECTOR& for all output parameters.

You can find more details concerning DirectXMath calling conventions in the online documentation at [http://msdn.microsoft.com/en-us/library/windows/desktop/hh437833\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/hh437833(v=vs.85).aspx).

## Accessors and Mutators

Although XMVECTOR objects are opaque, you need not *store* or *load* the object if you need to read or write only a single component. Instead, you can use the following accessor and mutator methods:

[Click here to view code image](#)

```
float XMVectorGetX(XMVECTOR V);  
float XMVectorGetY(XMVECTOR V);  
float XMVectorGetZ(XMVECTOR V);  
float XMVectorGetW(XMVECTOR V);  
  
void XMVectorSetX(XMVECTOR V, float X);  
void XMVectorSetY(XMVECTOR V, float Y);  
void XMVectorSetZ(XMVECTOR V, float Z);  
void XMVectorSetW(XMVECTOR V, float W);
```

## Initialization Functions

You can also initialize XMVECTOR objects in a variety of ways. [Table 2.2](#) lists some common initialization functions.

Function Prototype	Description
XMVECTOR XMVectorSet(float x, float y, float z, float w);	Returns (x, y, z, w).
XMVECTOR XMVectorReplicate(float value);	Returns (value, value, value, value).
XMVECTOR XMVectorSplatOne();	Returns (1, 1, 1, 1).
XMVECTOR XMVectorZero();	Returns (0, 0, 0, 0).

**Table 2.2** Common DirectXMath Vector Initialization Functions

## Operators

XMVECTOR has a series of overloaded operators for vector-to-vector addition, subtraction, multiplication, and division, as well as scalar-to-vector multiplication. These operators are listed here:

[Click here to view code image](#)

```

XMVECTOR operator+ (FXMVECTOR V);
XMVECTOR operator- (FXMVECTOR V);

XMVECTOR& operator+= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator-= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator*= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator/= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator*= (XMVECTOR& V, float S);
XMVECTOR& operator/= (XMVECTOR& V, float S);

XMVECTOR operator+ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator- (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator* (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator/ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator* (FXMVECTOR V, float S);
XMVECTOR operator* (float S, FXMVECTOR V);
XMVECTOR operator/ (FXMVECTOR V, float S);

```

## Vector Functions

DirectXMath has a variety of vector functions that support the operations discussed in this chapter. [Table 2.3](#) lists commonly used functions for 3D vectors. Note that most of these functions also exist in 2D and 4D form.

Function Prototype	Description
XMVECTOR XMVector3Length(FXMVECTOR V);	Returns the vector's length
XMVECTOR XMVector3LengthSq(FXMVECTOR V);	Returns the length squared
XMVECTOR XMVector3Normalize(FXMVECTOR V);	Normalizes the vector
XMVECTOR XMVector3Dot(FXMVECTOR V1, FXMVECTOR V2);	Computes the dot product
XMVECTOR XMVector3Cross(FXMVECTOR V1, FXMVECTOR V2);	Computes the cross product

**Table 2.3** Common DirectXMath 3D Vector Functions

### Note

For the vector functions in [Table 2.3](#), the operations that return a scalar replicate the value across each of the XMVECTOR components.

## Matrices

Matrices are supported through the XMMATRIX type—an array of four XMVECTOR objects. Just as with XMVECTOR, XMMATRIX objects aren't good candidates for class storage. Types such as XMFLOAT3X3 and XMFLOAT4X4 should be used instead. XMFLOAT4X4 has the following structure:

[Click here to view code image](#)

```
struct XMFLOAT4X4
{
    union
    {
        struct
        {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
};
```

Thus, you can access elements in traditional matrix subscript form or through zero-based [row, column] array access.

## Loading and Storing

XMMATRIX objects have *load* and *store* functions similar to XMVECTOR objects. For example:

[Click here to view code image](#)

```
XMVECTOR XMLoadFloat4×4 (const XMFLOAT4X4* pSource);

void XMStoreFloat4×4 (XMFLOAT4X4* pDestination, FXMMATRIX M);
```

Notice the type FXMMATRIX in the *store* method. Similar to the vector object, this is one of two aliases for XMMATRIX; the other is CXMMATRIX. Essentially, the calling conventions for XMMATRIX are to use FXMMATRIX for the first argument and use CXMMATRIX for everything else. A few exceptions to this rule apply; see the online documentation for details.

## Accessors, Mutators, and Initialization

XMMATRIX has an accessible member *r* for the array of XMVECTOR objects it contains. You can use it in conjunction with the XMVECTOR accessor, mutator, and initialization functions. An XMMATRIX can also be initialized through two constructors that accept either 4 XMVECTOR objects or 16 individual floats. In addition, you can initialize an identity matrix through this function:

[Click here to view code image](#)

```
XMMATRIX XMMatrixIdentity();
```

## Operators

XMMATRIX has a set of overloaded operators for matrix-to-matrix addition, subtraction, and multiplication, as well as scalar-to-matrix multiplication and division. These operators are listed here:

[Click here to view code image](#)

```
XMMATRIX& operator+= (FXMMATRIX M);
```

```

XMMATRIX& operator-= (FXMMATRIX M);
XMMATRIX& operator*= (FXMMATRIX M);
XMMATRIX& operator*= (float S);
XMMATRIX& operator/= (float S);

XMMATRIX operator+ (FXMMATRIX M) const;
XMMATRIX operator- (FXMMATRIX M) const;
XMMATRIX operator* (FXMMATRIX M) const;
XMMATRIX operator* (float S) const;
XMMATRIX operator/ (float S) const;

```

## Matrix Functions

DirectXMath includes a number of functions for performing matrix operations (such as the ones discussed in this chapter). [Table 2.4](#) lists some of the more commonly used functions.

Function Prototype	Description
XMMATRIX XMMatrixMultiply(FXMMATRIX M1, CXMMATRIX M2);	Returns the product of M1 and M2
XMMATRIX XMMatrixTranspose(FXMMATRIX M);	Returns the transpose of the matrix
XMMATRIX XMMatrixScaling(float sX, float sY, float sZ);	Returns a scaling matrix
XMMATRIX XMMatrixTranslation(float x, float y, float z);	Returns a translation matrix
XMMATRIX XMMatrixTranslationFromVector(FXMVECTOR offset);	Returns a translation matrix
XMMATRIX XMMatrixRotationX(float angle);	Returns a rotation matrix around the x-axis
XMMATRIX XMMatrixRotationY(float angle);	Returns a rotation matrix around the y-axis
XMMATRIX XMMatrixRotationZ(float angle);	Returns a rotation matrix around the z-axis
XMMATRIX XMMatrixRotationAxis(FXMVECTOR, float angle);	Returns a rotation matrix around an arbitrary axis
XMVECTOR XMVector3Transform(FXMVECTOR V, FXMMATRIX M);	Transforms the vector V by matrix M
XMVECTOR XMVector3TransformCoord(FXMVECTOR V, FXMMATRIX M);	Transforms the vector V by matrix M, where V.w = 1.
XMVECTOR XMVector3TransformNormal(FXMVECTOR V, FXMMATRIX M);	Transforms the vector V by matrix M, where V.w = 0.

**Table 2.4** Common DirectXMath Matrix Functions

The last three functions in [Table 2.4](#) perform vector-to-matrix multiplication using three-dimensional vectors. Note that you need not explicitly set the w component of the input vector for any of these functions. In `XMVector3Transform()` and `XMVector3TransformCoord()`, the value 1.0 is used for the w component. For `XMVector3TransformNormal()`, the fourth row (the translation row) is ignored. For `XMVector3Transform()`, the w component of the returned vector may be nonhomogeneous (not 1.0); for `XMVector3TransformCoord()`, the w component of the returned vector is guaranteed to be 1.0.

## Summary

This chapter provided an introduction to the mathematics behind 3D graphics. You've learned about vectors, matrices, coordinate systems, and transformations, and you've seen some of their use in graphics programming. You've also discovered the DirectXMath library, a cross-platform, SIMD-friendly C++ API for graphics applications.

This introduction focused on concepts central to graphics programming, but it just scratched the surface of a much larger topic. Indeed, entire books are dedicated to graphics-related mathematics, to which we've devoted only this single chapter. Likewise, the DirectXMath library has much more functionality than we've covered here. If you are interested in exploring these subjects further, many printed and online resources are available.

# Chapter 3. Tools of the Trade

Graphics programmers have access to a variety of excellent tools to assist in shader authoring, application development, debugging, and asset creation. Happily, many of these are free to use, so you can explore the subject without having to make a large investment. This chapter introduces some of the best tools currently on the market.

## Microsoft Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) for writing software on Microsoft platforms. It supports a large number of programming languages and hardware platforms, including Windows Phone, Xbox 360, and Xbox One. Of particular interest to us is Visual Studio's support for C++ and HLSL, along with the integrated Visual Studio Graphics Debugger.

Visual Studio has been evolving since the mid-1990s, and ten major releases have been produced since its inception. The latest version, and what we use in this book, is Visual Studio 2013 (see [Figure 3.1](#)). Visual Studio comes in several for-fee editions, with different feature sets. However, Visual Studio Express 2013 for Windows Desktop is a free alternative that provides much of the functionality we'll be using.

The screenshot shows the Microsoft Visual Studio 2013 interface. The title bar reads "Rendering - Microsoft Visual Studio". The menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, ANALYZE, WINDOW, and HELP. The toolbar includes icons for opening files, saving, and building. The status bar at the bottom shows "100 %", "Error List", "Output", "Find Symbol Results", "Find Results 1", "Ln 1", "Col 13", "Ch 13", and "INS".

The left side features the "Toolbox" and "Solution Explorer". The "Solution Explorer" pane shows a solution named "Rendering" containing two projects: "Game" and "Library". The "Library" project contains a "Content" folder with various effect files like BasicEffect.fx, Bloom.fx, DepthMap.fx, DistortionMask.fx, GaussianBlur.fx, ProjectiveTextureMapping.fx, ShadowMapping.fx, SkinnedModel.fx, and Skybox.fx. It also contains "Header Files", "Cameras", "Effects", "Input", "Lights", and "Materials" folders. The "Game" project is partially visible.

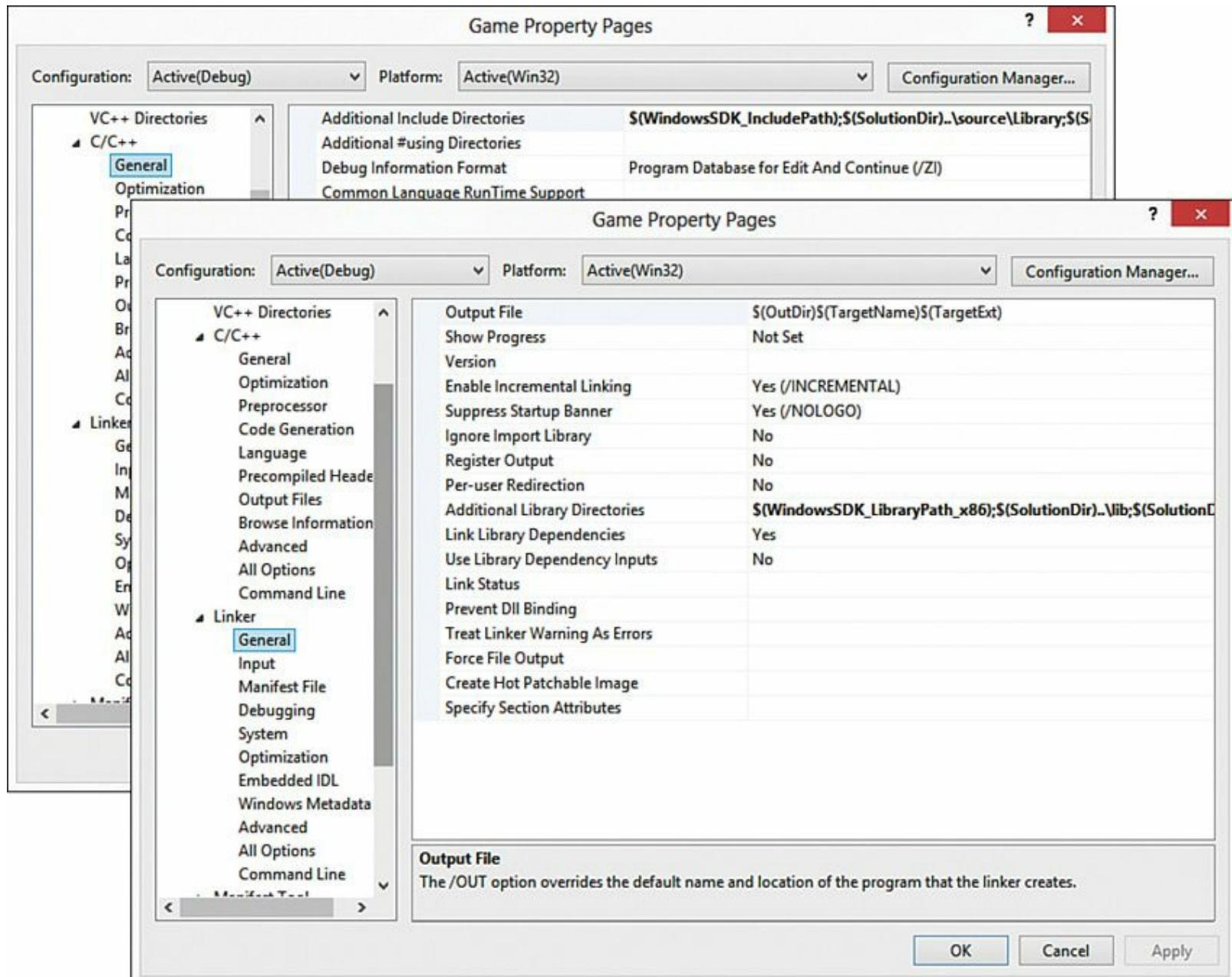
The main workspace is a code editor displaying the file "Skybox.h" (Global Scope). The code defines a class "Skybox" that inherits from "DrawableGameComponent". It includes RTTI declarations, constructor/destructor definitions, and overridden methods for Initialize, Update, and Draw. The class also contains private members for cube map file names, effects, materials, and buffers.

**Figure 3.1** Visual Studio 2013.

## Windows SDK

A full explanation of Visual Studio is beyond the scope of this book, but this section walks you through setting up Visual Studio for work against DirectX 11.1. The first step is to install the Windows SDK. With the release of Windows 8, DirectX is no longer a stand-alone installation, but it has been integrated with the Windows SDK. You can find links to the Windows SDK on this book's companion website.

The Windows SDK installation should add location macros for use in your C++ include and library paths—specifically, \$(WindowsSDK\_IncludePath) and \$(WindowsSDK\_LibraryPath\_x86) for the header and library files, respectively. Be sure these paths are included within your project's Additional Include/Library Directories settings (see [Figure 3.2](#)).



**Figure 3.2** Visual Studio 2013 property pages for include and library paths.

We'll be using Win32 projects, but the Windows SDK includes x64 versions of the libraries as well. Use the \$(WindowsSDK\_LibraryPath\_x64) macro for x64 builds. Also note that we explore the topic of C++ project setup more thoroughly in [Part III](#), “[Rendering with DirectX](#).”

## Effects 11 Library

If you have experience with a previous version of DirectX, you might be familiar with the Direct3D Extension (D3DX) library. This was a utility library that supplemented older versions of DirectX, but it has been deprecated in the Windows SDK. Plenty of useful features of D3DX existed, including a set of interfaces for the effect file format (the format we're using for our shaders). Thankfully, Microsoft has released replacement libraries for many of the D3DX features, including the Effects 11 library. However, the Effects 11 library isn't included within the Windows SDK either. As of this writing, the most recent version of the Effects 11 library is version 11.09 (released in January 2014). A link to this revision is available on this book's companion website.

The Effects 11 library is distributed in source code form, enabling users to customize its behavior. This also implies that you'll have to build the library before you'll be able to use it in your projects. A Visual Studio 2013 solution does exist for the library, with configurations for debug and release builds for Win32 and x64 platforms. The output of any of the stock configurations is a static library (Effect11.lib). [Part III](#) provides a full treatment of the library.

## DirectX Tool Kit

The DirectX Tool Kit (DirectXTK) is another D3DX replacement library that provides a number of useful C++ classes for working with Direct3D 11. [Table 3.1](#) lists a few highlights of the library.

Module	Description
SpriteBatch	Efficient 2D sprite rendering
SpriteFont	Text rendering with bitmap fonts
DDSTextureLoader	File loader for DDS textures
WICTextureLoader	File loader for Windows Imaging Components (WIC)
ScreenGrab	Screen shot grabber
GeometricPrimitives	Capability to generate and draw basic shapes such as cubes and spheres

**Table 3.1** Useful DirectXTK Components

As with the Effects 11 library, source code is provided for DirectXTK with a Visual Studio 2013 solution file. A link to the library is available on this book's companion website. We cover using several DirectXTK modules in [Part III](#).

## Texture File Formats

Textures are (typically) 2D bitmaps used to cover the surfaces of your objects. A variety of file formats exist for storing such images. Borrowing from the DirectXTK library, we organize these formats into two groups:

- DDS, DirectDraw Surface (.dds)

- WIC, Windows Imaging Components (native: .bmp, .gif, .jpg, .png, .tiff, HD photo)

DDS is a Microsoft file format that stores compressed image data that the GPU can decompress. The compression ratio varies with the specified compression, but the capability to decompress this format directly on the GPU makes DDS quite useful. DDS takes its name (DirectDraw Surface) from a prior incarnation of DirectX.

The second texture grouping, WIC, is essentially any image format that isn't DDS. WIC is an extensible framework for working with images. This framework allows third parties to add support for non-native formats with a consistent interface for developers. WIC gets automatic support for a non-native format as soon as the user installs the associated codec.

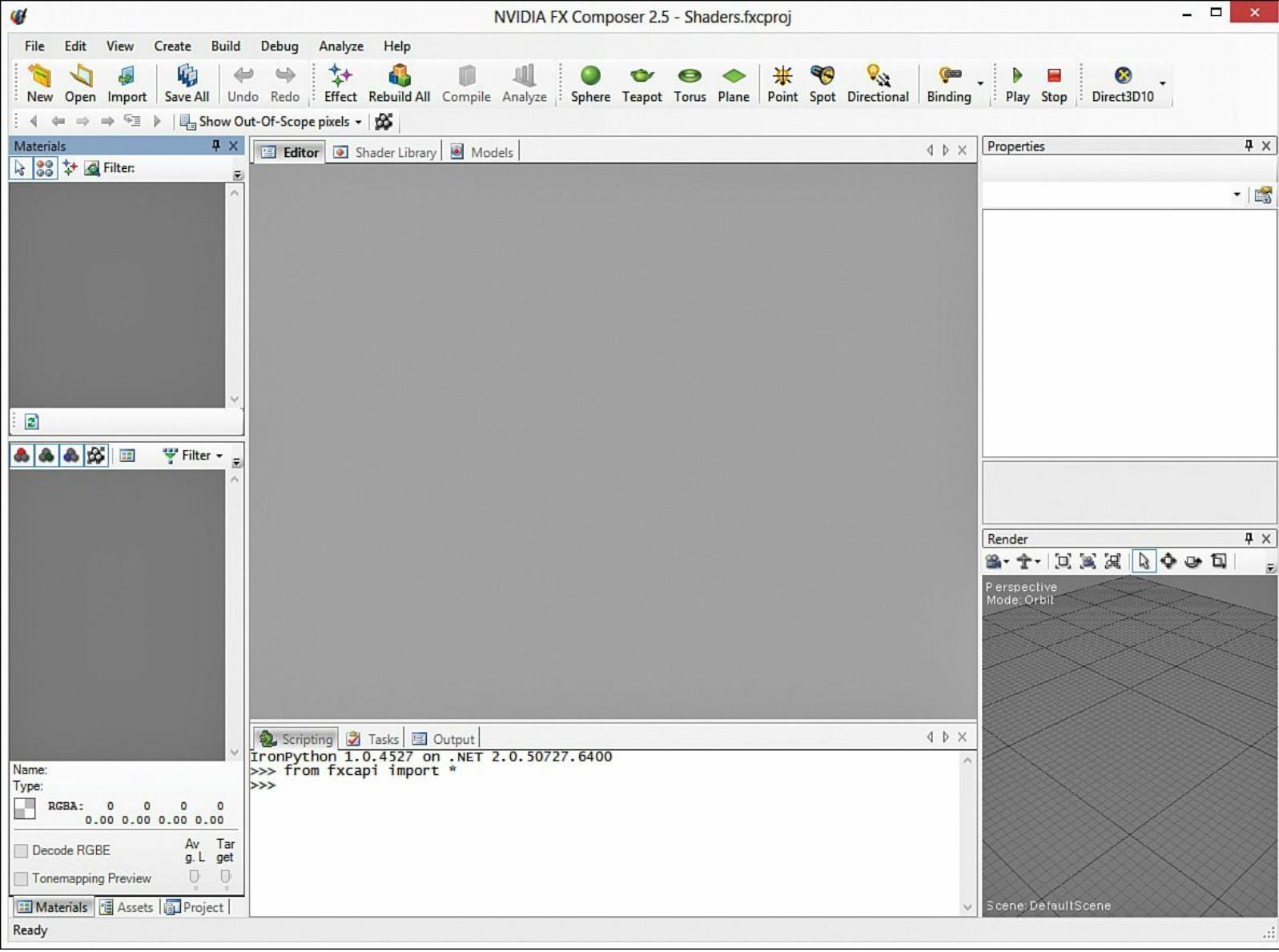
One such format you might consider is Targa (TGA), a common texture format that WIC doesn't natively support. You can find a link to a WIC TGA codec on this book's companion website. However, the current most popular TGA codec isn't free. An alternative to WIC for supporting TGA files is the DirectXTex texture processing library. This open-source library is intended for integration into a content pipeline for converting texture files to DDS format. It supports WIC but also has a TGA reader/writer.

## NVIDIA FX Composer

For gamers and graphics developers, NVIDIA has become a household name. It is credited with coining the term *GPU* and is a world leader in graphics hardware and software. Among NVIDIA's contributions to the graphics community is FX Composer, an integrated development environment for authoring shaders. This section introduces this tool—we make extensive use of FX Composer throughout this book.

First, note that writing shaders in HLSL is procedurally similar to writing any other software. Your shader code is stored in simple text files, and any text editor can modify it. Indeed, you might prefer to work outside of FX Composer entirely and write your shaders, for example, within Visual Studio. But using a purpose-built shader-authoring tool such as FX Composer offers a number of advantages, particularly with respect to the immediate visual feedback you receive as you modify your shaders. Furthermore, using a tool such as FX Composer enables you to concentrate on basic shader authoring before diving into the underlying graphics API (such as DirectX). But lest you be concerned about nonportable, tool-specific code, know that the shaders you'll write using FX Composer will be directly usable by the rendering engine authored in [Part III](#).

FX Composer is a free tool, and you can download it from NVIDIA's Developer Zone (for the link, see the book's website). As of this writing, the latest version of FX Composer is 2.5 (specifically, the filename is labeled as 2.53.0524.1905). [Figure 3.3](#) shows the default FX Composer window layout after the creation of a new project.



**Figure 3.3** NVIDIA FX Composer's default window layout.

## Aging Shader Authoring Tools

NVIDIA FX Composer is no longer in active development, and it's getting a little long in the tooth. It supports DirectX 9 and 10, but not DirectX 11. Similarly, AMD's shader-authoring tool, RenderMonkey, doesn't support DirectX 11, and its development has ceased as well. Furthermore, as of this writing and to the best of my knowledge, no other commercially available alternatives exist for the rapid development and visualization of shaders. This poses both a problem and an opportunity.

It's a problem because such tools are invaluable, particularly for newcomers to the graphics field. Without a shader-authoring tool, you must wade into the details of the underlying graphics API before you can visualize even the most simplistic rendering techniques. You experience what's required firsthand in [Part III](#). The first several chapters in the section are devoted to establishing a rendering framework in C++ without accomplishing any actual rendering. Such a hurdle can be discouraging, whereas a shader-authoring tool enables you to begin immediately visualizing the fruits of your labor.

The lack of up-to-date shader-authoring tools also presents an opportunity for ambitious

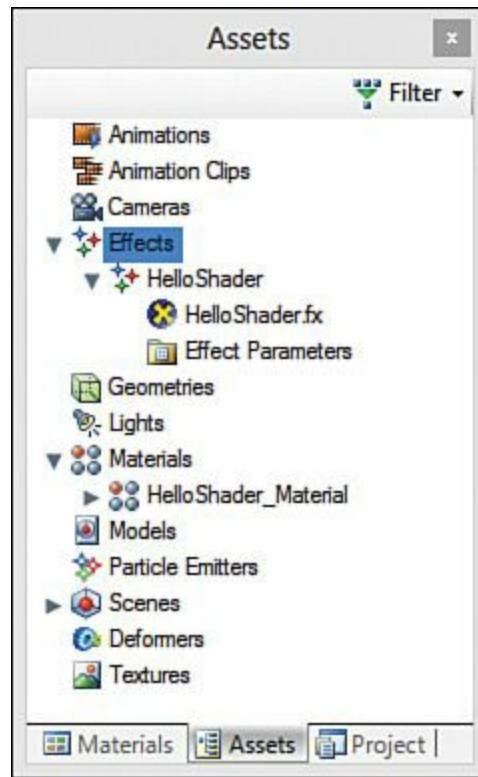
software developers because there is a gap in this space.

Ultimately, I chose to use NVIDIA FX Composer for [Part II](#) of the book because it provides immediate visual feedback and because its lack of DirectX 11 support isn't a show stopper for introductory rendering techniques. None of this early work requires features specific to DirectX 11, and none of this material is in any way invalidated by the latest version of the API.

As you can see, the layout is similar to Visual Studio, with resizable, dockable panels. Many of these panels (including the Editor, Properties, and Output panels) are self-explanatory. We need to look at several others.

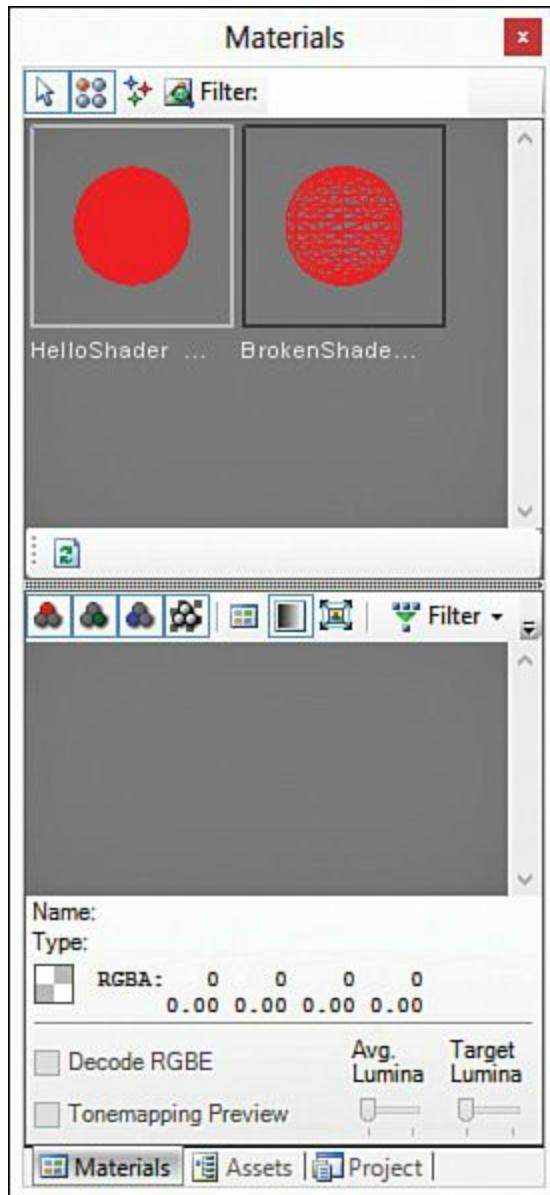
## Effects and Materials

Within the Assets panel (see [Figure 3.4](#)), you find headings marked **Effects** and **Materials**. An effect represents a file containing your shader code. A material is essentially an *instance* of an effect with values specified for any variables the shaders expose. Multiple materials can be associated with the same effect.



**Figure 3.4** NVIDIA FX Composer's Assets panel.

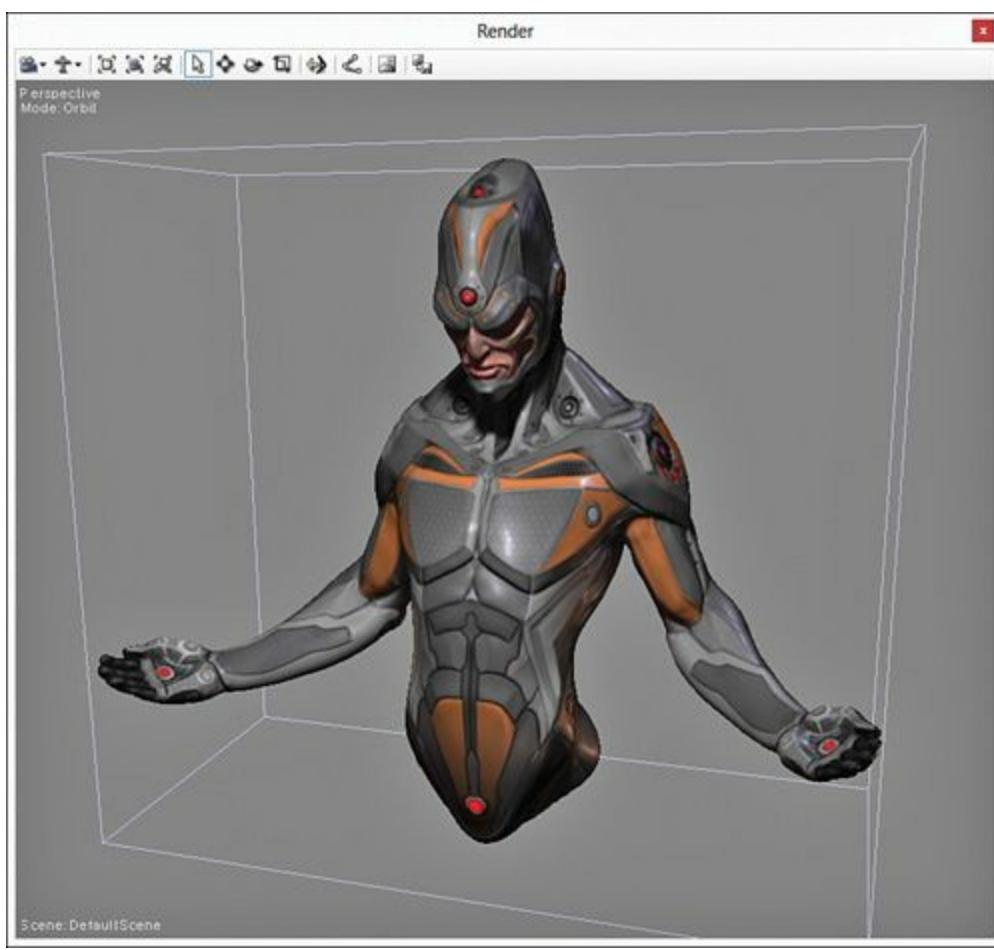
An error-free material (one whose underlying effect compiles correctly) is displayed in the Materials panel as a sphere with a representation of the shader's rendered output. If the associated effect fails to compile, the sphere appears as a red wireframe (see [Figure 3.5](#)). The lower portion of the Materials panel displays any textures assigned to the material.



**Figure 3.5** NVIDIA FX Composer's Materials panel with two materials (broken material on right).

## The Render Panel

By default, the Render panel resides in the lower-right corner. This panel presents the output of your shaders applied to geometry in a scene (see [Figure 3.6](#)). For such objects, you can use any of the prebuilt model buttons in the main toolbar (Sphere, Teapot, Torus, and Plane), or you can import your own custom geometry. To import a model, choose **File, Import** from the menu or click the **Import** button from the toolbar. You can include as many models in a scene as you'd like, and you can assign each a different material. To apply a material to an object in the scene, simply drag and drop the material from the Materials panel to the object in the Render panel.



**Figure 3.6** NVIDIA FX Composer’s Render panel. (*3D Model by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

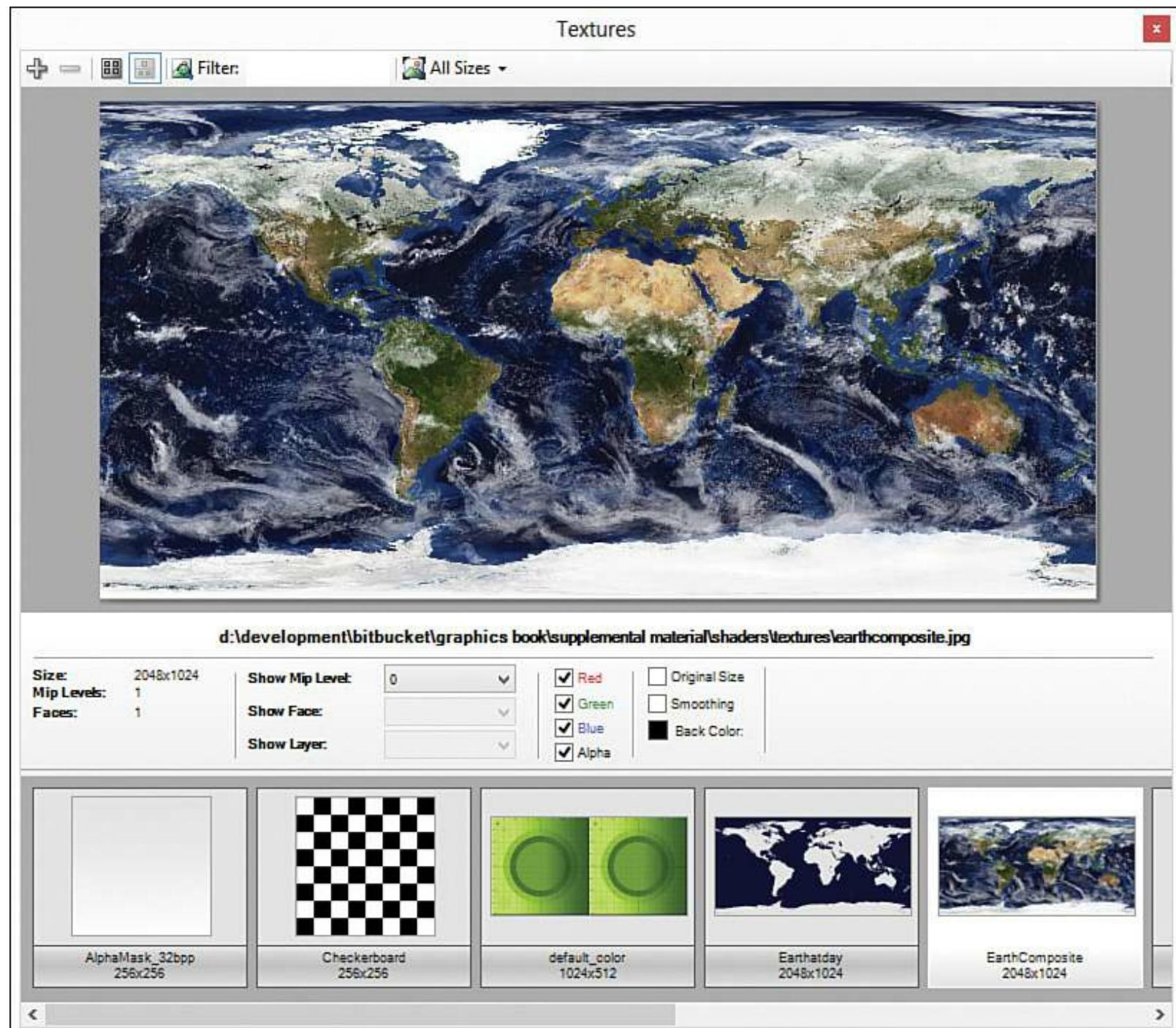
You control the camera presenting the 3D scene within the Render panel using the mouse and keyboard. If you’re familiar with Autodesk Maya (a popular 3D modeling package), the controls are similar. You can select objects in the scene with a left mouse click. With the default **Orbit** camera, pressing the **F** key selects a model as the camera’s focus point. Holding the **Alt** key while left-clicking and dragging the mouse rotates the camera around the object. You hold the **Shift** key while left-clicking and vertically dragging the mouse to zoom the camera in or out. Alternately, you can zoom the camera with the mouse wheel. To pan the camera, hold the **Ctrl** key while left-clicking and dragging the mouse.

Along with the camera, you can manipulate objects within the scene. This is accomplished through the *mode* keys **Q**, **W**, **E**, and **R**, or their associated toolbar buttons along the top of the Render panel. The **Select Object** mode (**Q**) selects an object with a left mouse-click but does not otherwise modify it. **Translate Object** mode (**W**) enables you to move the object within the scene. With an object selected in translate mode, axis-aligned control arrows appear at the center of the object. You can translate along a specific axis by dragging the corresponding arrow, or you can translate in free-move mode by dragging the plus icon at the intersection of the three arrows. **Rotate Object** (**E**) and **Scale Object** (**R**) modes behave in the same way as Translate Object mode.

The Render panel has additional features, including background color customization, a reference grid, trackball and flythrough cameras, and screen capture. I encourage you to experiment with the Render panel to familiarize yourself with these features.

## The Textures Panel

The Textures panel (see [Figure 3.7](#)) enables you to add and remove textures from the project, assign them to materials, and inspect the details of individual textures. To add textures, click the plus icon in the upper-left corner of the panel, or just drag and drop the textures from Windows Explorer. To assign a texture to a material, you can drag and drop from the Textures panel to an object within the Render panel, or you can specify the variable from within the Properties panel. Double-click an item in the Textures panel to view its details and to preview specific color channels.



**Figure 3.7** NVIDIA FX Composer's Textures panel. (*Original Earth textures from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

### Note

Assigning textures to a material requires that the associated effect support one or more texture variables. If an effect supports multiple textures, dropping a texture on an object

presents a dialog box for selecting the variable to assign.

We discuss this topic further in [Chapter 5](#), “[Texture Mapping](#).”

## Visual Studio Graphics Debugger

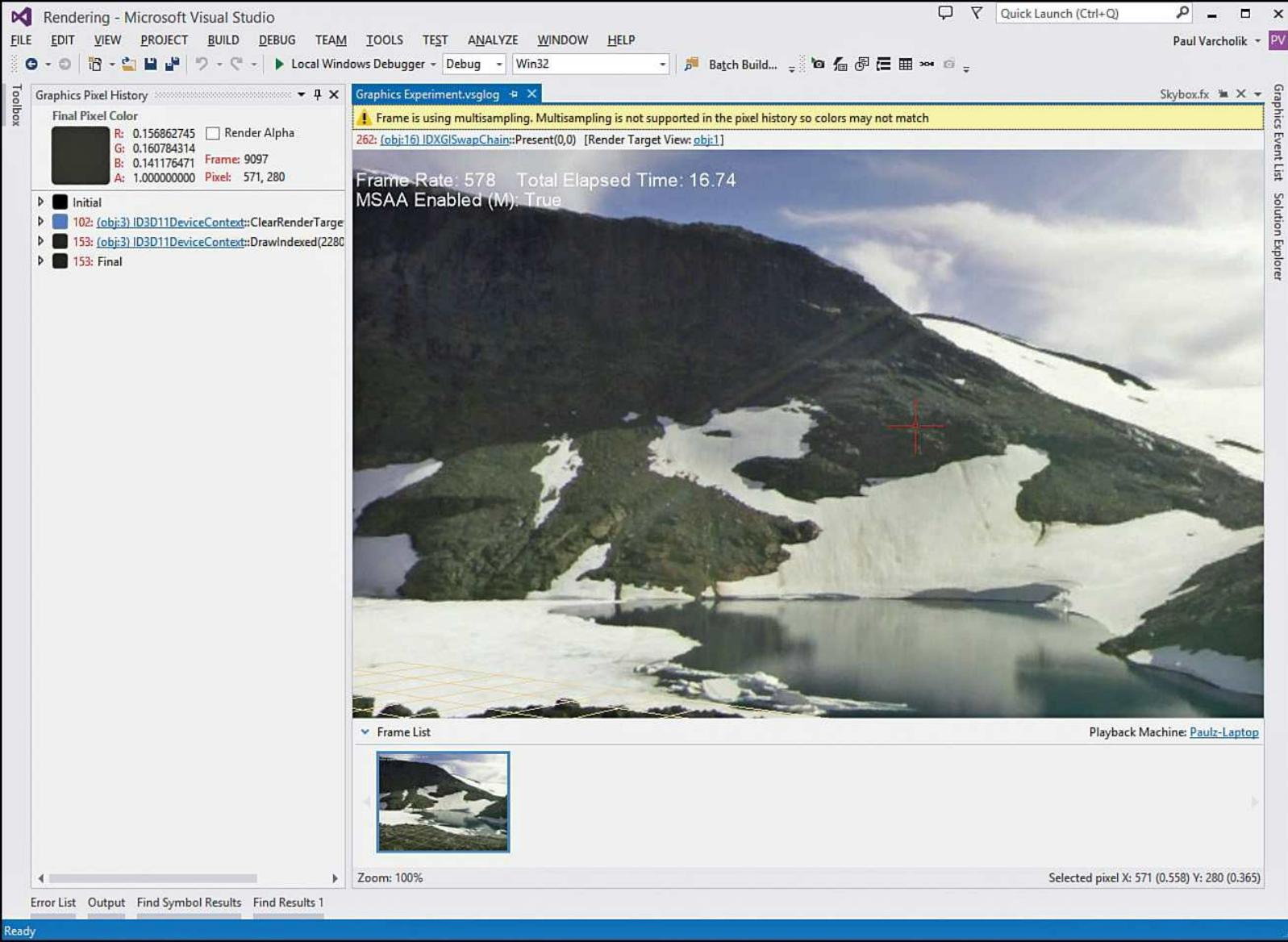
The Visual Studio Graphics Debugger allows interactive debugging of your custom shaders directly within the Visual Studio 2013 IDE. If you are familiar with older versions of DirectX, you might have worked with the graphics debugger PIX. The Visual Studio Graphics Debugger has replaced PIX but retained the same basic workflow.

### Warning

The Visual Studio Graphics Debugger is not available within Visual Studio Express 2013. You must acquire one of the for-fee versions of Visual Studio to work with this tool. No worries, though—you can complete all the work in this book without the aid of the Visual Studio Graphics Debugger.

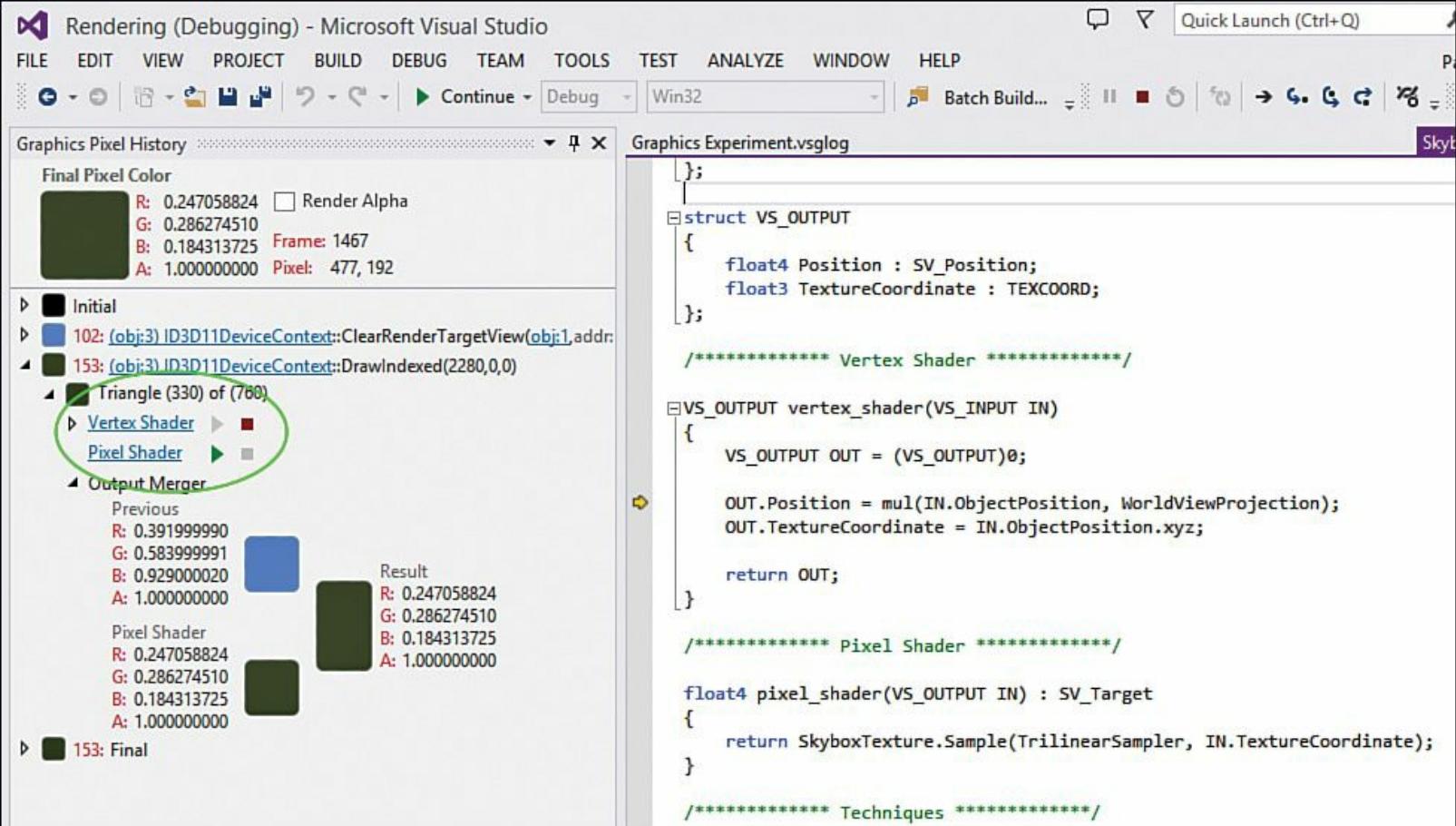
To launch the debugger, choose **Debug, Graphics, Start Diagnostics** from the Visual Studio main menu, or press **Alt+F5**. This launches your application and also opens a Visual Studio tab titled `Graphics Experiment.vsglog`. Press the **Print Screen** key to capture diagnostic information for a single frame from your running application. This adds the captured image to the log tab’s frame list, enabling you to capture multiple frames at various points in your application.

Debugging shaders isn’t quite the same as debugging a traditional application. Recall that your shader code executes on, for example, individual vertices and pixels. So you need to specify which pixel you want to debug (which also identifies the pixel’s corresponding primitive). To select a pixel, move your mouse over a captured frame. Notice that the cursor becomes a large red crosshair (see [Figure 3.8](#)). Choosing a pixel with the crosshair displays its history in the **Graphics Pixel History** panel. If you don’t see that panel in your display, choose **Debug, Graphics, Pixel History** from Visual Studio’s main menu.



**Figure 3.8** The Visual Studio Graphics Debugger log with a pixel selected from a captured frame.  
*(Texture by Emil Persson.)*

With a pixel selected, you can view any draw calls associated with that pixel in the Graphics Pixel History panel. You can then expand one of those calls for a list of drawn primitives. After you expand a primitive, you see various stages of the graphics pipeline—ones that you've provided shaders to—and the output-merger stage. Notice the Play and Stop icons next to the vertex and pixel shader stages in [Figure 3.9](#). Click the Play button next to a shader to begin debugging your shader code. The normal Visual Studio debugging panels (for example, Call Stack, Locals, and Watch windows) are enabled when debugging shaders. The specifics of shader debugging will become clear as we proceed with future chapters.



**Figure 3.9** The start/stop shader debugging buttons within the Visual Studio Graphics Debugger.

### Note

Don't close the vsglog tab while the graphics debugger is running if you still intend to capture frames. Closing the tab ends the graphics debugger session; it doesn't stop the application. The normal Visual Studio debugger takes over, but graphics diagnostics ends until you restart the application by pressing Alt+F5.

## Graphics Debugging Alternatives

You might want to explore two alternatives to the Visual Studio Graphics Debugger: GPU PerfStudio 2 by AMD and NVIDIA Nsight Visual Studio Edition. Both products are free and offer compelling feature sets. In particular, GPU PerfStudio 2 is a nice alternative for developers using an Express Edition of Visual Studio because it is a stand-alone tool and is not limited by your choice of IDE. You can find links to these products on the book's companion website.

## Summary

This chapter has provided an overview of the graphics-related tools we're using in this book. We discussed Visual Studio 2013 and the chief libraries we're using for DirectX application-side development. We also introduced NVIDIA's FX Composer for shader authoring. Finally, we looked at the Visual Studio Graphics Debugger for capturing diagnostic information and interactive shader debugging. This chapter marks the end of [Part I](#), “[An Introduction to 3D Rendering](#).” In the next chapter, we dive into the process of authoring shaders.

## Exercises

1. Set up your Visual Studio 2013 installation and the libraries discussed in this chapter. Visit the book's website for links to these resources.
2. Install NVIDIA FX Composer and experiment with the interface. In particular, explore the Render panel to become familiar with the camera controls. Add a sphere, teapot, torus, and plane to the Render panel, and modify them using the mode keys Q, W, E, and R (translate, rotate, and scale).

# Part II: Shader Authoring with HLSL

[4 Hello, Shaders!](#)

[5 Texture Mapping](#)

[6 Lighting Models](#)

[7 Additional Lighting Models](#)

[8 Gleaming the Cube](#)

[9 Normal Mapping and Displacement Mapping](#)

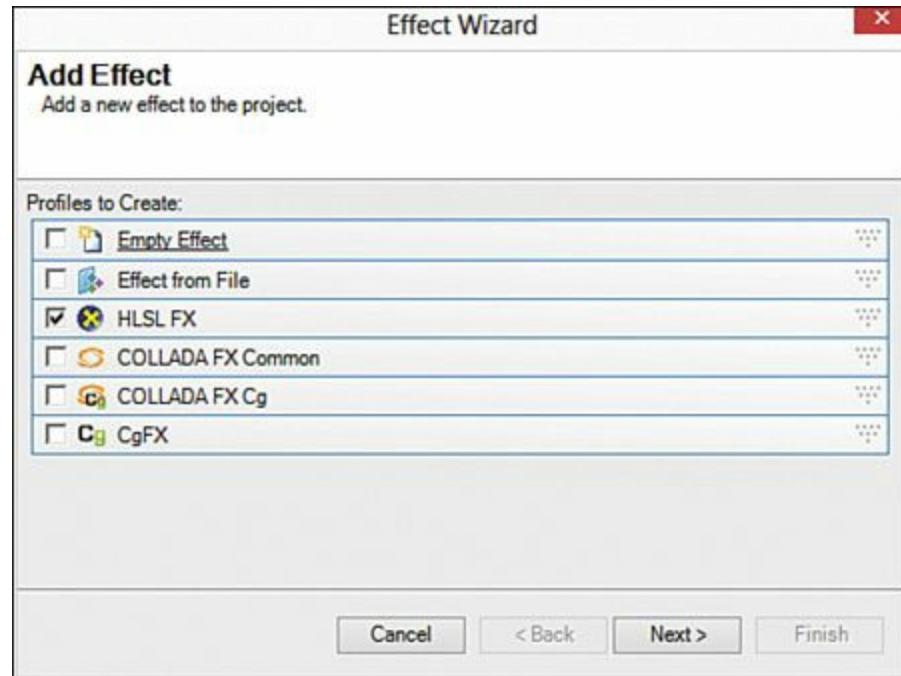
# Chapter 4. Hello, Shaders!

In this chapter, you write your first shaders. We introduce HLSL syntax, the FX file format, data structures, and more. By the end of this chapter, you'll have a base from which to launch the rest of your exploration into graphics programming.

## Your First Shader

You might recognize the canonical programming example “Hello, World!” as a first program written in a new language and whose output is this simple line of text. We follow this time-honored tradition with the shader equivalent “Hello, Shaders!”—this time, your output is an object rendered in a solid color.

To begin, launch NVIDIA FX Composer and create a new project. Open the Assets panel, right-click on the Materials icon, and choose Add Material from New Effect. Then choose HLSL FX from the Add Effect dialog box (see [Figure 4.1](#)).



**Figure 4.1** NVIDIA FX Composer Add Effect dialog box.

In the next dialog box, select the Empty template and name your effect HelloShaders.fx (see [Figure 4.2](#)).



**Figure 4.2** NVIDIA FX Composer Select HLSL FX Template dialog box.

Click Finish in the final dialog box of the Effect Wizard to complete the process. If all went well, you should see your `HelloShaders.fx` file displayed in the Editor panel and associated `HelloShaders` and `HelloShaders_Material` objects listed in the Assets panel. Notice that the Empty effect template isn't empty after all—NVIDIA FX Composer has stubbed out a bit of code for you. This code is actually close to what you want in your first shader, but it's written for DirectX 9, so delete this code and replace it with the contents of [Listing 4.1](#). Then we walk through this code step by step.

### **Listing 4.1** `HelloShaders.fx`

[Click here to view code image](#)

```
cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

RasterizerState DisableCulling
{
    CullMode = NONE;
};

float4 vertex_shader(float3 objectPosition : POSITION) :
SV_Position
{
    return mul(float4(objectPosition, 1), WorldViewProjection);
}

float4 pixel_shader() : SV_Target
```

```

    {
        return float4(1, 0, 0, 1);
    }

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

---

## Effect Files

Direct3D pipeline stages can be programmed through separately compiled shaders. For instance, you can house a vertex shader in one file (commonly with the extension .hlsl) and a pixel shader in a different file. Under this configuration, each file must contain exactly one shader. By contrast, HLSL Effect files enable you to combine multiple shaders, support functions, and render states into a single file. This is the file format we use throughout this text, and [Listing 4.1](#) uses it.

## Constant Buffers

At the top of your HelloShaders.fx file, you find a block of code starting with cbuffer. This denotes a **constant buffer**, whose purpose is to organize one or more shader constants. A shader constant is input the CPU sends to a shader, which remains constant for all the primitives processed by a single draw call. Put another way, cbuffers hold variables, “constant variables.” They’re constant from the perspective of the GPU while processing the primitives of a draw call yet variable from the perspective of the CPU from one draw call to the next.

In your HelloShaders.fx file, you have just one cbuffer containing only one shader constant, WorldViewProjection, of type `float4×4`. This is a C-style variable declaration in which the data type is a  $4\times 4$  matrix of single-precision floating-point values. This particular variable (WorldViewProjection) represents the concatenated World-View-Projection matrix specific to each object. Recall from [Chapter 2, “A 3D/Math Primer,”](#) that this matrix transforms your vertices from object space, to world space, to view space, to homogeneous space, in a single transformation. You could pass the World, View, and Projection matrices into the effect separately and then perform three different transforms to produce the same result. But unless you have a specific reason to do so, sending less data as input and performing fewer shader instructions is the better option.

Note the text WORLDVIEWPROJECTION following the colon in the variable declaration. This is known as a **semantic** and is a hint to the CPU-side application about the intended use of the variable. Semantics relieve the application developer from *a priori* knowledge of the names of shader constants. In this example, you could have named your `float4×4` variable WVP or WorldViewProj without any impact to the CPU side because it can access the variable through the

WORLDVIEWPROJECTION semantic instead of through its name. A variety of common semantics exist, all of which are optional for shader constants. However, in the context of NVIDIA FX Composer, the WORLDVIEWPROJECTION semantic is not optional; it must be associated with a shader constant for your effect to receive updates to the concatenated WVP matrix each frame.

## What's in a Name?

In your HelloShaders effect, you named your constant buffer `CBufferPerObject`. Although the name itself isn't magical, it does hint at the intended update frequency for the shader constants contained within the `cbuffer`. A `PerObject` buffer indicates that the CPU should update the data within that buffer for each object associated with the effect.

In contrast, a `cbuffer` named `CBufferPerFrame` implies that the data within the buffer can be updated just once per frame, allowing multiple objects to be rendered with the same *shared* shader constants.

You organize `cbuffers` in this way for more efficient updates. When the CPU modifies any of the shader constants in a `cbuffer`, it has to update the entire `cbuffer`. Therefore, it's best to group shader constants according to their update frequency.

## Render States

Shaders can't define the behaviors of the nonprogrammable stages of the Direct3D pipeline, but you can customize them through render state objects. For example, the rasterizer stage is customized through a `RasterizerState` object. A variety of rasterizer state options exist, although I defer them to future chapters. For now, note the `RasterizerState` object `DisableCulling` (see [Listing 4.2](#)).

### Listing 4.2 RasterizerState declaration from `HelloShaders.fx`

[Click here to view code image](#)

```
RasterizerState DisableCulling
{
    CullMode = NONE;
}
```

We briefly discussed vertex winding order and backface culling in [Chapter 3, “Tools of the Trade.”](#) By default, DirectX considers vertices presented counter-clockwise (with respect to the camera) to be back-facing and does not draw them. However, the default models included with NVIDIA FX Composer (the Sphere, Teapot, Torus, and Plane) are wound in the opposite direction. Without modifying or disabling the culling mode, Direct3D would cull what we would consider front-facing triangles. Therefore, for your work within NVIDIA FX Composer, just disable culling by specifying `CullMode = NONE`.

### Note

The culling issue is present within NVIDIA FX Composer because it supports both

DirectX and OpenGL rendering APIs. These libraries disagree on the default winding order for front-facing triangles, and NVIDIA FX Composer opted for the OpenGL default.

## The Vertex Shader

The next HelloShaders code to analyze is the vertex shader, reproduced in [Listing 4.3](#).

### Listing 4.3 The vertex shader from HelloShaders.fx

[Click here to view code image](#)

```
float4 vertex_shader(float3 objectPosition : POSITION) :  
SV_Position  
{  
    return mul(float4(objectPosition, 1), WorldViewProjection);  
}
```

This code resembles a C-style function, but with some key differences. First, note the work the vertex shader is accomplishing. Each vertex comes into the shader in object space, and the `WorldViewProjection` matrix transforms it into homogeneous clip space. In general, this is the least amount of work a vertex shader performs.

The input into your vertex shader is a `float3`, an HLSL data type for storing three single-precision floating-point values—it's named `objectPosition` to denote its coordinate space. Notice the `POSITION` semantic associated with the `objectPosition` parameter. It indicates that the variable is holding a vertex position. This is conceptually similar to the semantics used for shader constants, to convey the intended use of the parameter. However, semantics are also used to link shader inputs and outputs between shader stages (for example, between the input-assembler stage and the vertex shader stage) and are therefore required for such variables. At a minimum, the vertex shader must accept a variable with the `POSITION` semantic and must return a variable with the `SV_Position` semantic.

#### Note

Semantics with the prefix `SV_` are system-value semantics and were introduced in Direct3D 10. These semantics designate a specific meaning to the pipeline. For example, `SV_Position` indicates that the associated output will contain a transformed vertex position for use in the rasterizer stage.

While other, non-system-value semantics exist, including a set of standard semantics, these are generic and are not explicitly interpreted by the pipeline.

Within the body of your vertex shader, you're calling the HLSL intrinsic function `mul`. This performs a matrix multiplication between the two arguments. If the first argument is a vector, it's treated as a row vector (with a row-major matrix as the second argument). Conversely, if the first argument is a matrix, it's treated as a column major matrix, with a column-vector as the second argument. We use

row-major matrices for most of our transformations, so we use the form `mul(vector, matrix)`. Notice that, for the first argument of the `mul` function, you are constructing a `float4` out of the `objectPosition` (a `float3`) and the number 1. This is required because the number of columns in the vector must match the number of rows in the matrix. Because the vector you're transforming is a position, you hard-code the fourth float (the `w` member) to 1. Had the vector represented a direction, the `w` component would be set to 0.

## The Pixel Shader

As with the vertex shader, the `HelloShader` pixel shader is just one line of code (see [Listing 4.4](#)).

### Listing 4.4 The pixel shader from `HelloShaders.fx`

[Click here to view code image](#)

```
float4 pixel_shader() : SV_Target
{
    return float4(1, 0, 0, 1);
}
```

The return value of this shader is a `float4` and is assigned the `SV_Target` semantic. This indicates that the output will be stored in the render target bound to the output-merger stage. Typically, that render target is a texture that is mapped to the screen and is known as the **back buffer**. This name comes from a technique called **double buffering**, in which two buffers are employed to reduce tearing, and other artifacts, produced when pixels from two (or more) frames are displayed simultaneously. Instead, all output is rendered to a back buffer while the actual video device displays a **front buffer**. When rendering is complete, the two buffers are swapped so that the newly rendered frame displays. Swapping is commonly done to coincide with the refresh cycle of the monitor—again, to avoid artifacts.

The output of your pixel shader is a 32-bit color, with 8-bit channels for Red, Green, Blue, and Alpha (RGBA). All values are supplied in floating-point format, where the range [0.0, 1.0] maps to integer range [0, 255]. In this example, you're supplying the value 1 to the red channel, meaning that every pixel rendered will be solid red. You are not employing color blending, so the alpha channel has no impact. If you were using color blending, an alpha value of 1 would indicate a fully opaque pixel. We discuss color blending in more detail in [Chapter 8, “Gleaming the Cube.”](#)

#### Note

Your `HelloShaders` pixel shader accepts no apparent input parameters, but don't let this confuse you. The homogeneous clip space position of the pixel *is* being passed to the pixel shader from the rasterizer stage. However, this happens behind the scenes and is not explicitly declared as input into the pixel shader.

In the next chapter, you see how additional parameters are passed into the pixel shader.

## Techniques

The last section of the HelloShaders effect is the technique that brings the pieces together (see [Listing 4.5](#)).

## **Listing 4.5** The technique from HelloShaders.fx

[Click here to view code image](#)

```
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

A technique implements a specific rendering sequence through a set of effect passes. Each pass sets render states and associates your shaders with their corresponding pipeline stages. In the HelloShaders example, you have just one technique (named main10) with just one pass (named p0). However, effects can contain any number of techniques, and each technique can contain any number of passes. For now, all your techniques contain a single pass. We discuss techniques with multiple passes in [Part IV, “Intermediate-Level Rendering Topics.”](#)

Note the keyword `technique10` in this example. This keyword denotes a Direct3D 10 technique, versus DirectX 9 techniques, which have no version suffix. Direct3D 11 techniques use the keyword `technique11`. Unfortunately, the current version of NVIDIA FX Composer does not support Direct3D 11. But you won’t be using any Direct3D 11–specific features at the beginning of your exploration of shader authoring, so this isn’t a show stopper. We start using Direct3D 11 techniques in [Part III, “Rendering with DirectX.”](#)

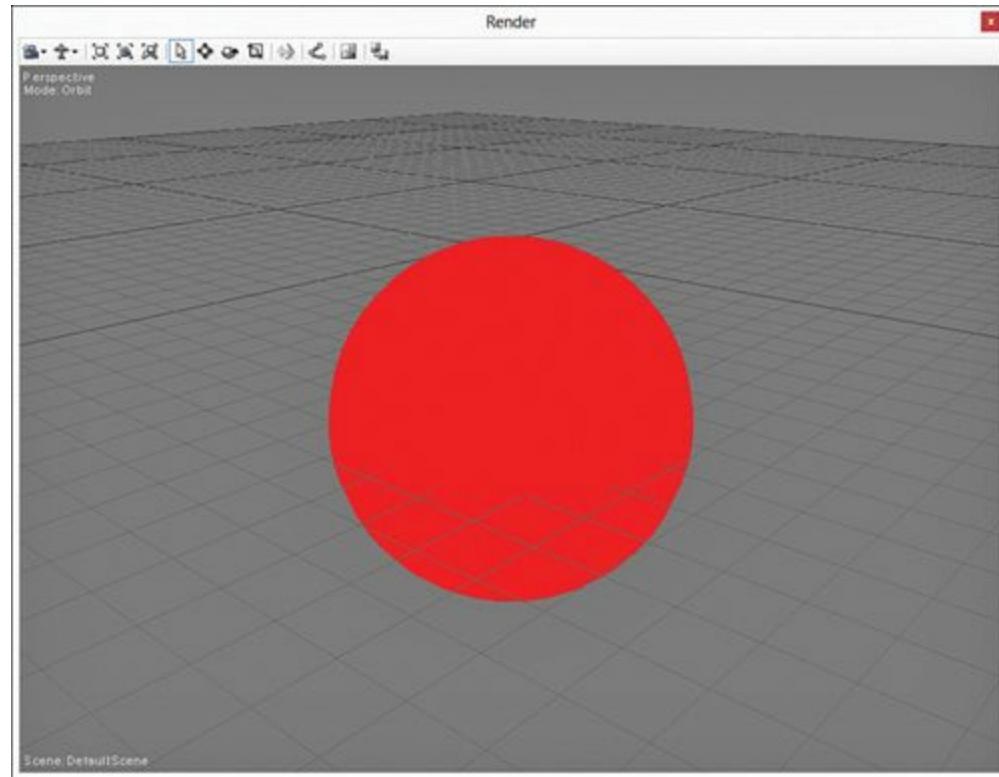
Also notice the arguments `vs_4_0` and `ps_4_0` within the `SetVertexShader` and `SetPixelShader` statements. These values identify the shader profiles to use when compiling the shaders specified in the second arguments of the `CompileShader` calls. Shader profiles are analogous to shader models, which define the capabilities of the graphics system that are required to support the corresponding shaders. As of this writing, there have been five major (and several minor) shader model revisions; the latest is shader model 5. Each shader model has extended the functionality of the previous revision in a variety of ways. Generally, however, the potential sophistication of shaders has increased with each new shader model. Direct3D 10 introduced shader model 4, which we use for all Direct3D 10 techniques. Shader model 5 was introduced with Direct3D 11, and we use that shader model for all Direct3D 11 techniques.

## Hello, Shaders! Output

You’re now ready to visualize the output of the HelloShaders effect. To do so, you first need to build your effect through the **Build, Rebuild All** or **Build, Compile HelloShaders.fx** menu commands.

Alternately, you can use the shortcut keys **F6** (Rebuild All) or **Ctrl+F7** (Compile Selected Effect). Be sure you do this after any changes you make to your code.

Next, ensure that you are using the Direct3D 10 rendering API by choosing it from the drop-down menu in the main toolbar (it's the right-most toolbar item, and it likely defaults to Direct3D 9). Now open the Render panel within NVIDIA FX Composer. Its default placement is in the lower-right corner. Create a sphere in the Render panel by choosing **Create, Sphere** from the main menu or by clicking the Sphere icon in the toolbar. Finally, drag and drop your `HelloShaders` `_Material` from either the Materials panel or the Assets panel onto the sphere in the Render panel. You should see an image similar to [Figure 4.3](#).



**Figure 4.3** `HelloShaders.fx` applied to a sphere in the NVIDIA FX Composer Render panel.

This might be a bit anti-climactic, given the effort to get here, but you've actually accomplished quite a lot! Take a few minutes to experiment with the output of this shader. Modify the RGB channels within the pixel shader to get a feel for what's happening.

## Hello, Structs!

In this section, you rewrite your `HelloShaders` effect to use C-style structs. Data structures provide a way to supply multiple shader inputs and outputs with a bit more organization than as individual parameters.

To start, create a new effect and material in NVIDIA FX Composer. You can do this through the Add Effect Wizard, as you did at the beginning of this chapter, or you can copy `HelloShaders.fx` to a new file, `HelloStructs.fx`. I like the second option because you'll often reuse your shader code, building upon the previous material. With a copied `HelloStructs.fx` file, you add it to NVIDIA FX Composer by right-clicking the Materials section of the Assets panel and choosing Add Material from File. Find and select your `HelloStructs.fx` file, and you'll see newly created `HelloStructs` and `HelloStructs_Material` objects in the Assets panel.

[Listing 4.6](#) contains a full listing of the `HelloStructs.fx` effect.

## Listing 4.6 HelloStructs.fx

[Click here to view code image](#)

---

```
cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

RasterizerState DisableCulling
{
    CullMode = NONE;
};

struct VS_INPUT
{
    float4 ObjectPosition: POSITION;
};

struct VS_OUTPUT
{
    float4 Position: SV_Position;
};

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);

    return OUT;
}

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return float4(1, 0, 0, 1);
}

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

```
}
```

```
}
```

---

The differences between `HelloShaders.fx` and `HelloStructs.fx` are minor but significant because they establish the conventions we use throughout this text. First, note what has not changed. The `CBufferPerObject` and `DisableCulling` objects are the same, as are the `main10` technique and its pass. The body of the pixel shader hasn't changed, either. What's new are the two structs named `VS_INPUT` and `VS_OUTPUT`. These names identify the structures as vertex shader inputs and outputs, respectively. Notice that the `VS_INPUT` struct has the same `ObjectPosition` input variable as the `HelloShaders` vertex shader. The only difference is that the variable is declared as a `float4` instead of a `float3`. This removes the need to append the value 1 to the `w` component of the vector. Additionally, the vertex shader now returns a `VS_OUTPUT` instance instead of a `float4`, and the `SV_Position` semantic is no longer associated directly to the return value because it's attached instead to the `Position` member of the `VS_OUTPUT` struct. That `Position` member replaces the previously unnamed return value of the vertex shader from the `HelloShaders` effect.

Next, examine the body of your updated vertex shader. Notice that you're declaring and returning a `VS_OUTPUT` instance, and in C-programming fashion, you access the `Position` member of the instance through the dot operator. Also notice that the `ObjectPosition` member of the `VS_INPUT` parameter `IN` is used for the `mul` invocation. In addition, you're using a C-style cast to initialize the members of the `OUT` variable to zero. Although this is not strictly necessary, it is a good programming practice.

Finally, observe that the input parameter for the pixel shader is the output data type from the vertex shader. You're not using any members of the input in this example, but you will do so in future shaders. The point of this reorganization is that now you can add shader inputs and outputs without modifying the signature of your vertex and pixel shaders. The output of `HelloStructs` should be identical to that of `HelloShaders`, in [Figure 4.3](#).

## Summary

In this chapter, you wrote your first HLSL shaders! You learned a bit about the FX file format, constant buffers, and render states. You also began to explore HLSL syntax, including vector and matrix data types (such as `float3`, `float4`, and `float4×4`) and user-defined structs. And you put all this together within NVIDIA FX Composer to produce your first rendered output. The work you've accomplished in this chapter serves as a foundation for the rest of the shaders in [Part II](#), “[Shader Authoring with HLSL](#).”

## Exercises

1. Change the values of the RGB channels in the `HelloShaders` or `HelloStructs` pixel shader, and observe the results.
2. Modify the `DisableCulling` rasterizer state object by setting `CullMode = FRONT` and then `BACK`, and observe the results.
3. Now that you have a couple effects, get comfortable working within NVIDIA FX Composer. Create Teapot, Torus, and Plane objects, and assign them either the `HelloShaders` or `HelloStructs` materials. Notice how all objects that are assigned the same material are

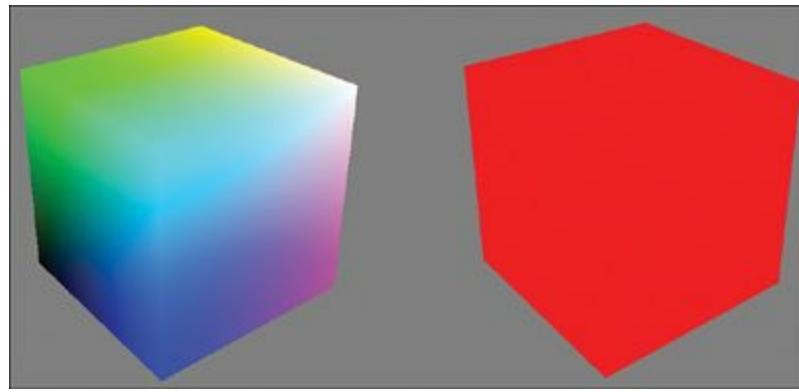
impacted when you change and recompile the associated effect.

# Chapter 5. Texture Mapping

Texture mapping is the process of adding detail to the surface of a 3D object. This can be likened to gift wrapping, where your wrapping paper is a 2D texture. Texture mapping is fundamental to modern rendering and is used for a multitude of interesting graphics techniques.

## An Introduction to Texture Mapping

Generally, you'll want more detail in your rendered 3D objects than the solid color produced by the HelloShader and HelloStructs effects from the last chapter. As you've learned, 3D models consist of vertices, typically organized into triangles. Those vertices are described, at a minimum, by a position. But they can contain additional information. A step toward additional surface detail is to provide a color for each vertex. Consider the 3D cubes in [Figure 5.1](#).

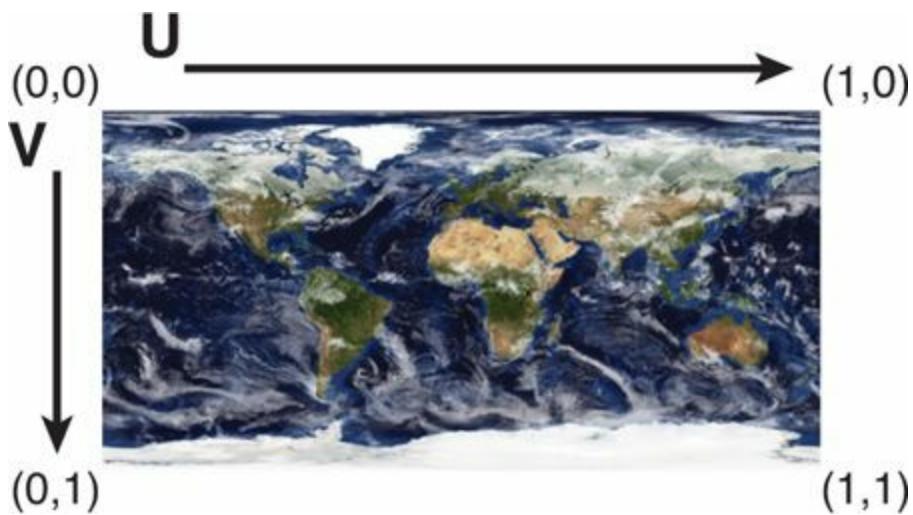


**Figure 5.1** 3D cubes with different colors for each vertex (left) and solid red (right).

On the cube to the left, each vertex is supplied a different color; on the right, each vertex has the same color. Clearly, the cube on the left has more detail. The colors add interest to the object and help define the faces of the cube. Furthermore, the colors of the individual pixels vary with respect to their positions relative to the vertices (recall the discussion of interpolation during the rasterizer stage from [Chapter 1](#), “[Introducing DirectX](#)”). You gain more control over the color of a surface by increasing the number of colored vertices that make up your 3D objects. However, when viewed close up, you'll never have enough vertices to produce a high-quality rendering, and any such attempt would quickly yield an overwhelming number of vertices. The solution is texture mapping.

To map a texture to a triangle, you include two-dimensional coordinates with each vertex. You use those coordinates to look up a color stored in a 2D texture. And you do this lookup in the pixel shader to find the color for each pixel of the triangle. The texture coordinates, then, are interpolated from the triangle vertices instead of the vertex color.

DirectX texture coordinates range from 0 to 1 (inclusive) on both the horizontal and vertical axes, where the origin is the top-left corner. These axes are commonly given the names u (horizontal) and v (vertical). [Figure 5.2](#) shows a texture mapped to a quad (two triangles) and highlights the texture coordinates for each vertex.



**Figure 5.2** DirectX 2D texture coordinates. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

### Note

Direct3D supports 1D, 2D, and 3D textures along with texture arrays and texture cubes (discussed further in [Chapter 8, “Gleaming the Cube”](#)). The number of coordinates required for lookup matches the dimensionality of the texture.

## A Texture Mapping Effect

[Listing 5.1](#) presents the code for a texture mapping effect. As before, create a new effect/material in NVIDIA FX Composer and transpose the code from the listing. Then you can examine this code step by step.

### **Listing 5.1** TextureMapping.fx

[Click here to view code image](#)

```
/****** Resources *****/
#define FLIP_TEXTURE_Y 1

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
}

RasterizerState DisableCulling
{
    CullMode = NONE;
};
```

```
Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/************************************************************************************************ Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/************************************************************************************************ Utility Functions *****/
float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

/************************************************************************************************ Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
```

```

    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    return OUT;
}

/**************************************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
}

/**************************************** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));
        SetRasterizerState(DisableCulling);
    }
}

```

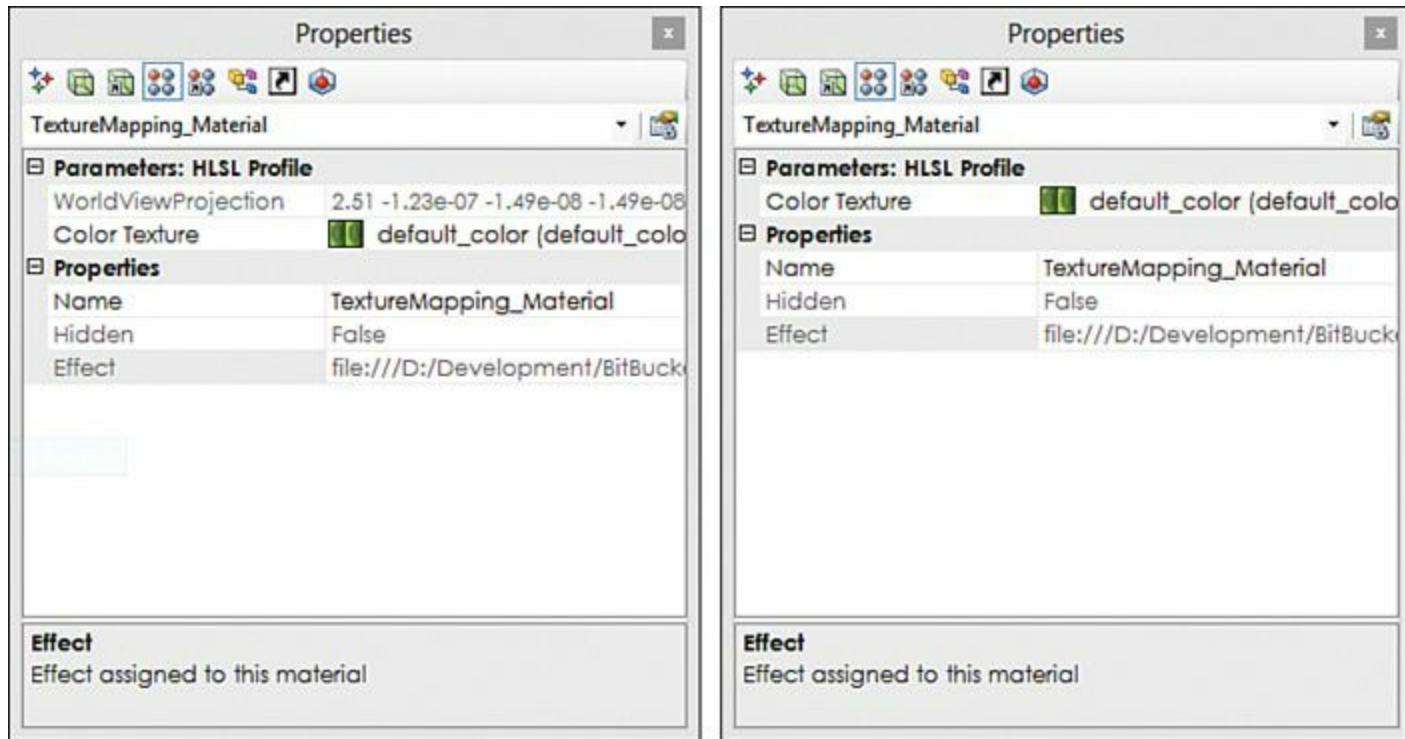
---

## Comments, the Preprocessor, and Annotations

A number of language constructs come into play from this effect. First, notice the comments organizing the effect. HLSL supports C++-style single-line comments (`// comments`) and multiline comments (`/* comments */`).

Next, recognize the `#define FLIP_TEXTURE_Y 1` macro at the top of the effect. This has behavior identical to the C/C++ `#define` directive. Indeed, HLSL has several familiar preprocessor commands, including `#if`, `#else`, `#endif`, and `#include`.

Now examine the `WorldViewProjection` shader constant, declared within the `cbuffer`. This constant has the same behavior as in `HelloShaders` and `HelloStructs`, but you've added an annotation to the end of the declaration. Annotations are *notes* to the CPU-side application and are enclosed in angled brackets. These notes do not affect shader execution, but the application can use them. For example, the `UIWidget` annotation, attached to `WorldViewProjection`, controls how NVIDIA FX Composer treats that shader constant. With a value of `None`, NVIDIA FX Composer hides the shader constant from the list of visible material properties. [Figure 5.3](#) shows the NVIDIA FX Composer Properties panel both without the annotation (left) and with the annotation to hide the constant (right). Note that this only hides the value from the Properties panel; the CPU still updates the constant. Hiding the constant makes sense for the `WorldViewProjection` matrix because you won't be hand-editing that matrix.



**Figure 5.3** NVIDIA FX Composer Properties panel showing the properties for `TextureMapping.fx` without the `UIWidget="None"` annotation on `WorldViewProjection` (left) and with the annotation (right).

## Texture Objects and Samplers

Three steps are involved in using a texture within an HLSL effect. First, you must declare the texture object (see [Listing 5.2](#)). An HLSL texture declaration can use the explicit subtype (such as `Texture2D`) or the more generic `texture` data type. Texture objects cannot be declared within `cbuffers`.

### Listing 5.2 The Texture Object Declaration from `TextureMapping.fx`

[Click here to view code image](#)

---

```
Texture2D ColorTexture <
    string resourceName = "default_color.dds";
    string UIName = "Color Texture";
    string resourceType = "2D";
>;
```

---

#### Note

Three annotations are associated with the `ColorTexture` variable in [Listing 5.2](#). As with all annotations, these are optional, but in the context of NVIDIA FX Composer, these annotations offer enhanced usability.

The `UIName` annotation enables you to customize the displayed variable name within the Properties panel. `ResourceType` identifies the type of texture that can be assigned, and `ResourceName` allows a default texture to be used when the user has not supplied

one.

Next, you must declare and initialize a texture sampler (see [Listing 5.3](#)). Samplers control how a color is retrieved from a texture. Direct3D 10 introduced the SamplerState data type, and it directly maps to a corresponding Direct3D C struct with members for **filtering** and **texture address modes**. We discuss these topics shortly.

### **Listing 5.3** The Sampler Object Declaration from `TextureMapping.fx`

[Click here to view code image](#)

```
SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

The final step is to sample the texture using the declared sampler object. This step is performed in the pixel shader (see [Listing 5.4](#)).

### **Listing 5.4** The Pixel Shader from `TextureMapping.fx`

[Click here to view code image](#)

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
}
```

Note the C++-style method invocation of the `Sample()` method against the `ColorTexture` object. The first argument to the `Sample()` method is the sampler object, followed by the coordinates to look up in the texture.

## Texture Coordinates

The vertex stream supplies the coordinates for sampling the texture, with corresponding members in the `VS_INPUT` and `VS_OUTPUT` data structures (see [Listing 5.5](#)). Observe the `TEXCOORD` semantics attached to the 2D `TextureCoordinate` members.

### **Listing 5.5** The Vertex Shader Input and Output Data Structures from `TextureMapping.fx`

[Click here to view code image](#)

```
struct VS_INPUT
```

```

{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

```

---

The vertex shader (see [Listing 5.6](#)) passes the input texture coordinates to the output of the stage, but it does so after invoking `get_corrected_texture_coordinate()`. HLSL supports user-defined, C-style helper functions, and this one simply inverts the vertical texture coordinate if `FLIP_TEXTURE_Y` is nonzero. This is necessary within NVIDIA FX Composer because it uses OpenGL-style texture coordinates for the built-in 3D models (the Sphere, Teapot, Torus, and Plane). The origin for OpenGL texture coordinates is the bottom-left corner instead of the top-left corner for DirectX. Therefore, when you're visualizing your shaders in NVIDIA FX Composer against one of the built-in models, you need to flip the vertical texture coordinate. If you import a custom model into NVIDIA FX Composer and that model has DirectX-style texture coordinates, you can disable the `FLIP_TEXTURE_Y` macro.

### **Listing 5.6** The Vertex Shader and a Utility Function from `TextureMapping.fx`

[Click here to view code image](#)

---

```

float2 get_corrected_texture_coordinate(float2
textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 -
textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

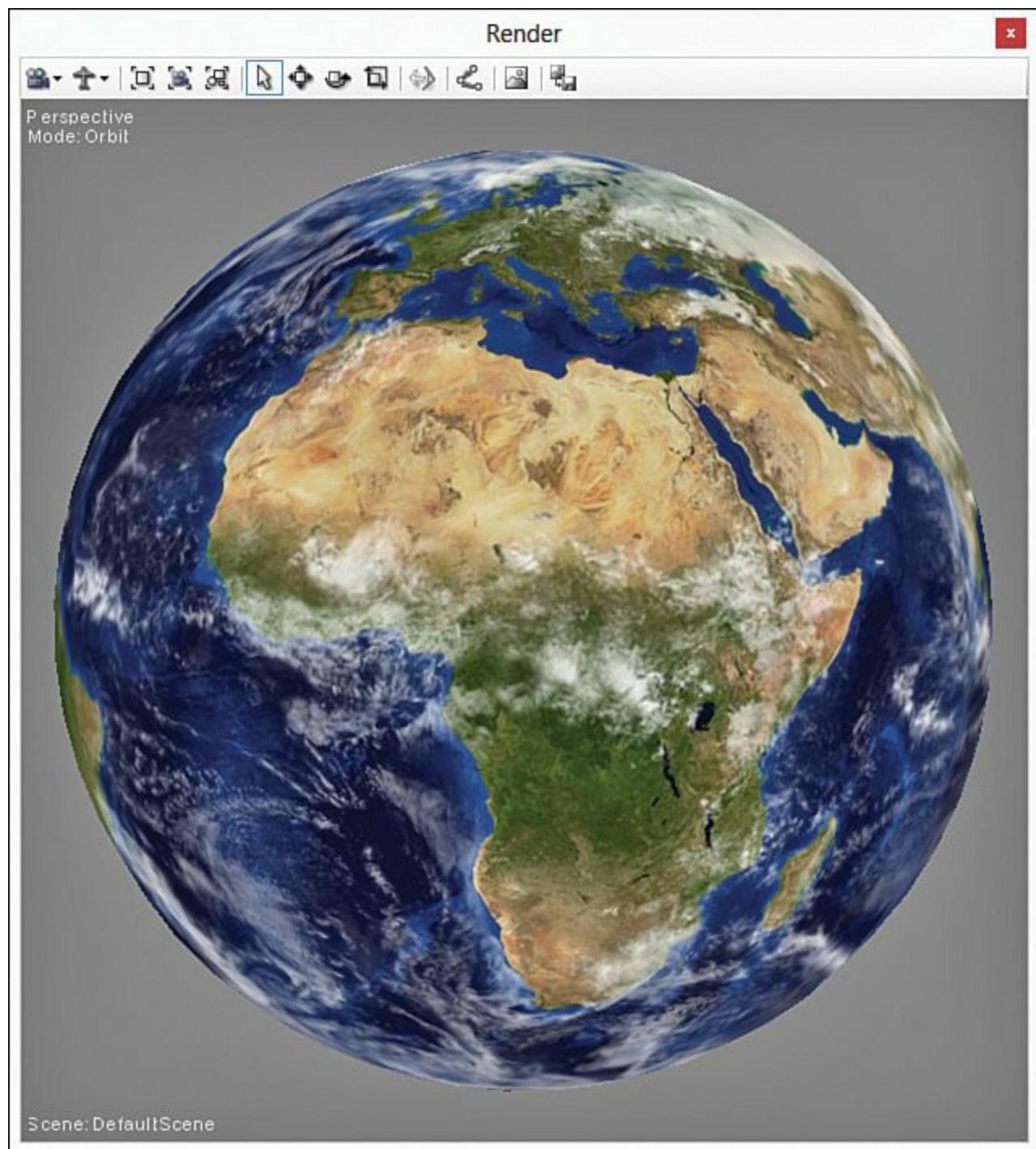
    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    return OUT;
}

```

## Texture Mapping Output

Figure 5.4 shows the output of the texture mapping effect applied to a sphere with a texture of Earth.



**Figure 5.4** TextureMapping.fx applied to a sphere with a texture of Earth. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

## Texture Filtering

In our discussion of texture coordinates and sampling, you might have realized that you will rarely view a texture-mapped object at a one-to-one correspondence between the elements in the texture (texels) and the pixels rendered to the screen. Your camera will be at an arbitrary distance from the

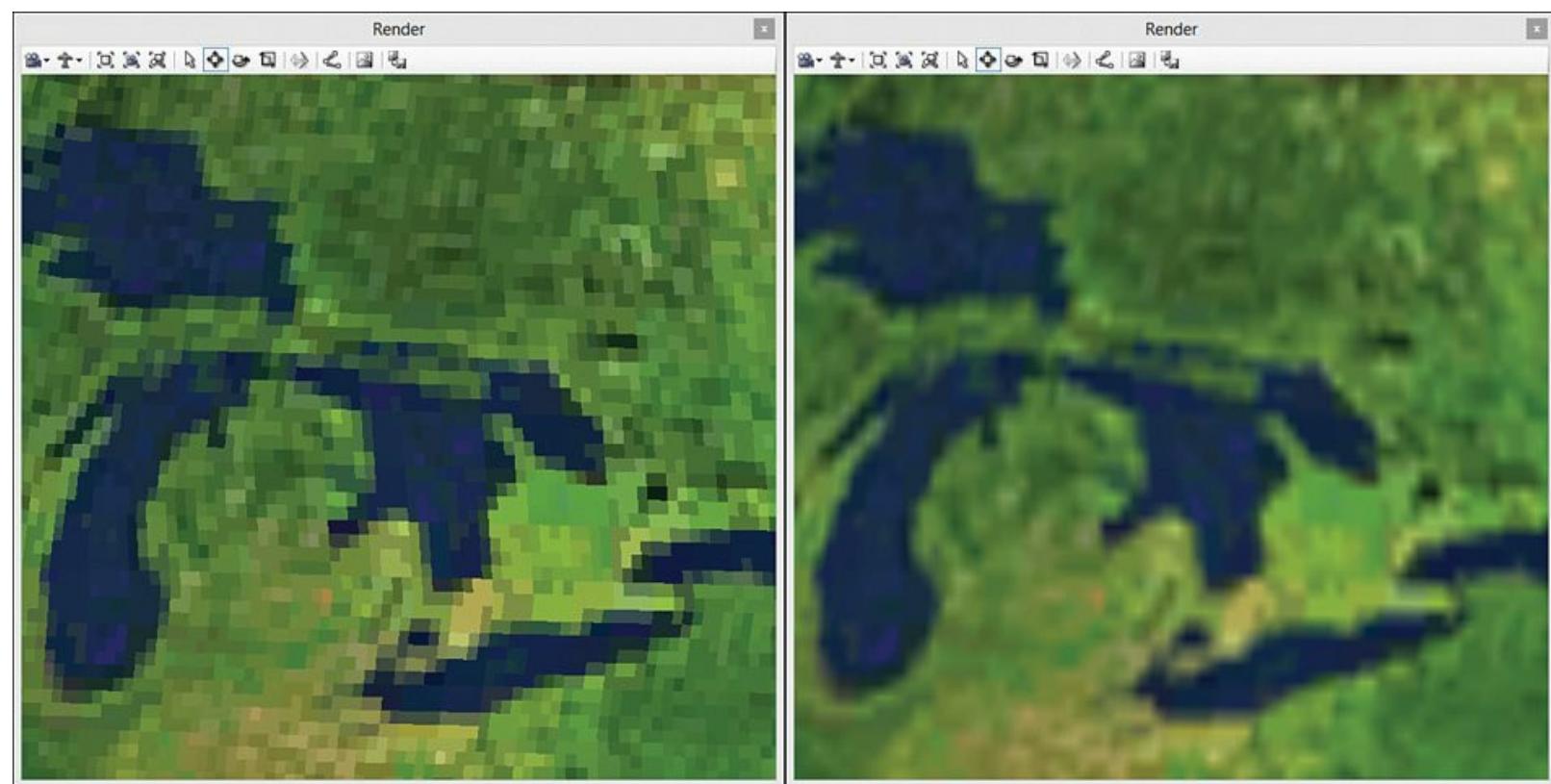
textured object and will view it at an arbitrary angle. Consider a scenario in which the camera is “zoomed in” to a textured object. The rasterizer stage determines which pixels to send to the pixel shader and interpolates their texture coordinates. But the coordinates represent locations that are in between authored texels in the texture. In other words, you’re trying to render the texture at a higher resolution than it was authored. The topic of **texture filtering** determines what color should be chosen for these pixels because they have no direct lookup in the texture.

## Magnification

The scenario just described is known as **magnification**—in this situation, there are more pixels to render than there are texels. Direct3D supports three types of filtering to determine the colors of these “in between” pixels: **point filtering**, **linear interpolation**, and **anisotropic filtering**.

### Point Filtering

Point filtering is the fastest of the filtering options, but it generally yields the lowest quality results. Also known as nearest-neighbor filtering, point filtering simply uses the color of the texel closest to the pixel center. Shown on the left side of [Figure 5.5](#), you can see the blocky-looking image produced with point filtering.



**Figure 5.5** The results of point filtering (left) and bilinear interpolation (right) on a magnified texture. (*Texture by Reto Stöckli, NASA Earth Observatory.*)

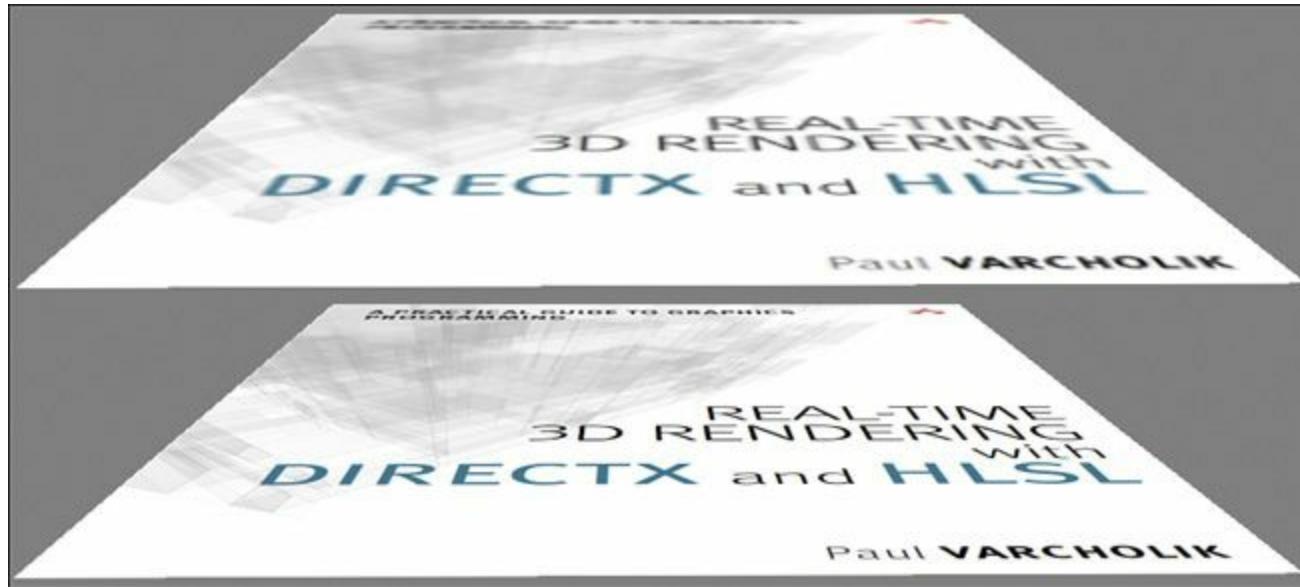
### Linear Interpolation

A higher-quality filtering option is linear interpolation, in which the color is interpolated between neighboring texels. For 2D textures, this is more correctly named **bilinear interpolation** because the interpolation takes place both horizontally and vertically. In this filtering technique, the four texels surrounding the pixel center are identified and two 1D linear interpolations are performed for the pixels along the u-axis. Then a third interpolation is performed along the v-axis to produce the final

color. The image on the right side of [Figure 5.5](#) shows the same magnified texture using bilinear interpolation. Notice the smoother appearance of this image compared to point filtering.

## Anisotropic Filtering

Anisotropy refers to the distortion of a texture projected onto a surface that is at an oblique angle to the camera. Anisotropic filtering reduces this distortion and improves the rendered output. This is the most expensive filtering option, but the results are compelling. [Figure 5.6](#) compares a texture mapped object with and without anisotropic filtering.



**Figure 5.6** A mapped texture with bilinear filtering (top) and anisotropic filtering (bottom).

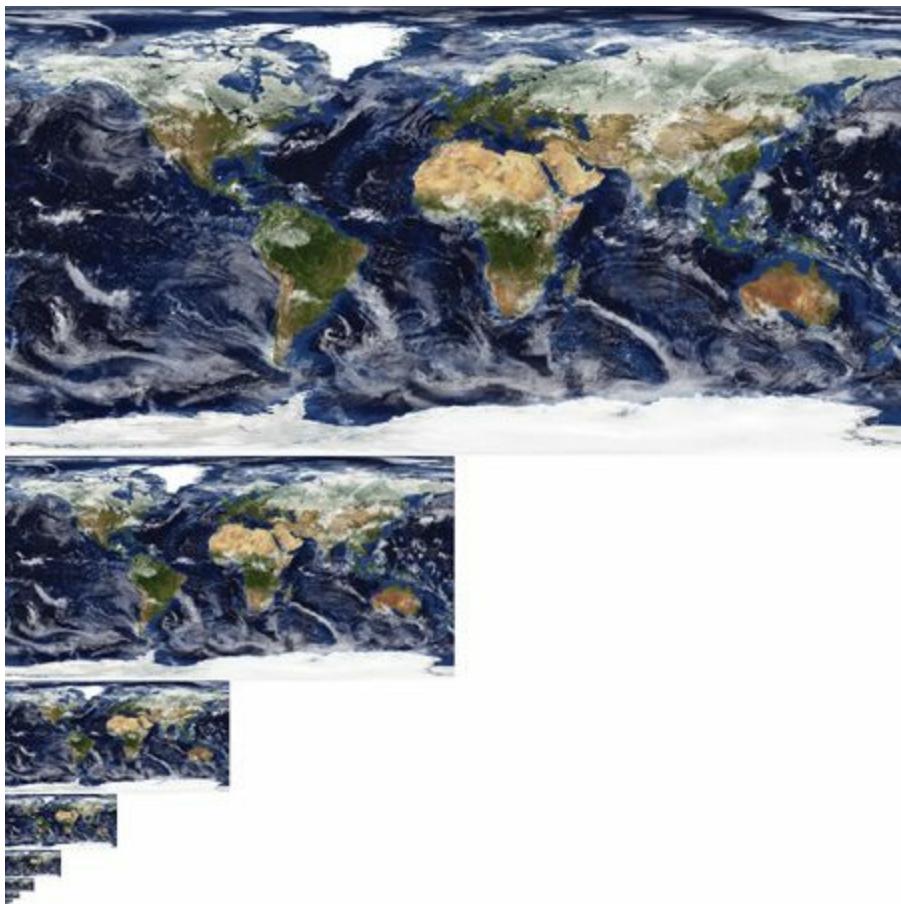
## Minification

**Minification** is the opposite of magnification and applies when a pixel covers more than one texel. This happens as the camera gets farther from a textured surface and Direct3D must choose the appropriate color. The same filtering options apply, but it's a bit more complicated for minification. With linear interpolation, for example, the four texels surrounding the pixel center are identified just as they were for magnification. However, the in-between texels (the multiple colors occupying the same lookup for the pixel center) are ignored and don't contribute to the final color. This can produce artifacts that decrease rendering quality.

You might envision an approach in which the texels are averaged to produce a better result. However, such computation is based on the resolution of the down-sampled texture, and an arbitrary number of “levels” can be computed. Building these levels on the fly would be impractical. Instead, this approach can be precomputed through a technique known as **mipmapping**.

## Mipmaps

Mipmaps are smaller versions of the original texture, typically precomputed and stored within the same file. Each mip-level is a progressive division by 2, down to a  $1 \times 1$  texture. [Figure 5.7](#) shows an example of the Earth texture at a size of  $512 \times 256$  with nine mip-levels.



**Figure 5.7** A mipmapped Earth texture with nine mip-levels. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

When employing mipmaps, two steps are required to produce a final color. First, the mip-level (or levels) must be selected for minification or magnification. Second, the selected mip-level(s) is filtered to derive a color.

Point or linear filtering can be used to select a mip-level. Point filtering simply selects the nearest mip-level, and linear filtering selects the two surrounding mipmaps. Then the selected mip-level is sampled through point, linear, or anisotropic filtering. If linear filtering was used to select the two neighboring mipmaps, both mip-levels are sampled and their results are interpolated. This technique produces the highest-quality results.

Finally, although mipmapping improves rendering quality, note that it also increases memory requirements by 33 percent.

## SamplerState Filtering Options

Refer to the `SamplerState` object from `TextureMapping.fx` (reproduced in [Listing 5.7](#)), and you'll notice the `Filter` member and its assigned value of `MIN_MAG_MIP_LINEAR`. This configures the sampler to use linear interpolation for minification, magnification, and mip-level sampling. Direct3D allows independent configurations for each of these elements. [Table 5.1](#) provides examples of the various permutations. You can find a full listing in the Direct3D documentation on the Microsoft Developer Network (MSDN) website.

### **Listing 5.7** The Sampler Object Declaration from `TextureMapping.fx`

[Click here to view code image](#)

```
SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

Enumeration	Description
MIN_MAG_MIP_POINT	Uses point filtering for minification, magnification, and mip-level sampling
MIN_MAG_MIP_LINEAR	Uses linear interpolation for minification, magnification, and mip-level sampling
ANISOTROPIC	Uses anisotropic filtering for minification, magnification, and mip-level sampling
MIN_MAG_POINT_MIP_LINEAR	Uses point filtering for minification and magnification; uses linear interpolation for mip-level sampling
MIN_LINEAR_MAG_POINT_MIP_LINEAR	Uses linear interpolation for minification and mip-level sampling; uses point filtering for magnification
MIN_MAG_LINEAR_MIP_POINT	Uses linear interpolation for minification and magnification; uses point filtering for mip-level sampling

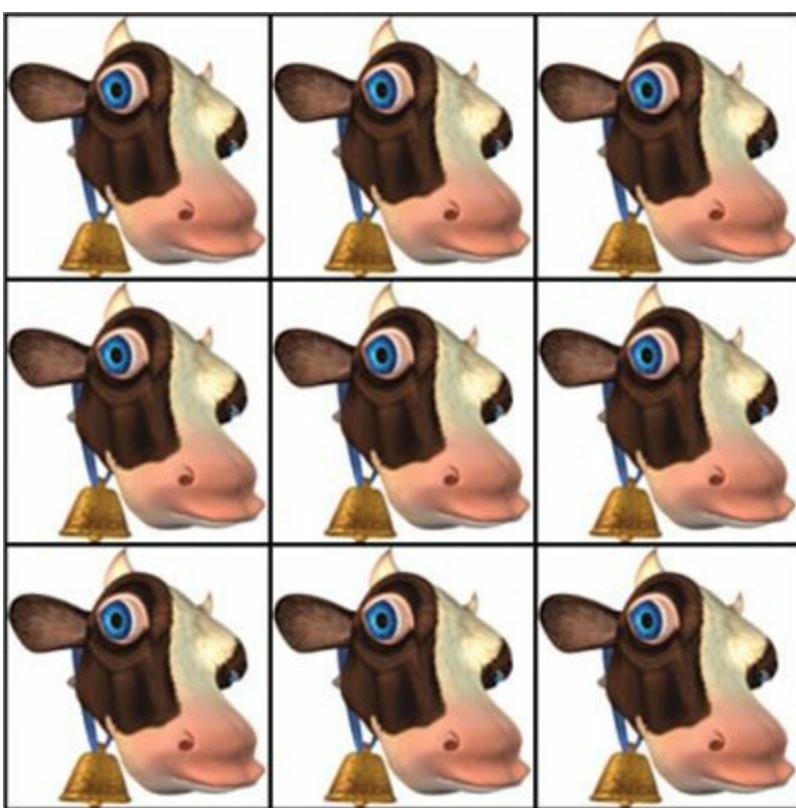
**Table 5.1** Texture Sampling Filter Options

## Texture Addressing Modes

You likely noticed the `AddressU = WRAP` and `AddressV = WRAP` settings from the `SamplerState` object (see [Listing 5.7](#)). Known as addressing modes, these enable you to control what happens when a texture coordinate is outside the range  $[0, 1]$ . Direct3D supports four addressing modes: Wrap, Mirror, Clamp, and Border.

### Wrap

In wrap texture address mode, your texture is repeated as vertex coordinates go below 0 or above 1. For example, mapping a texture to a quad with UVs of  $(0.0, 0.0)$ ,  $(3.0, 0.0)$ ,  $(0.0, 3.0)$ , and  $(3.3, 3.3)$  results in repeating the texture three times along both axes. [Figure 5.8](#) illustrates this scenario.



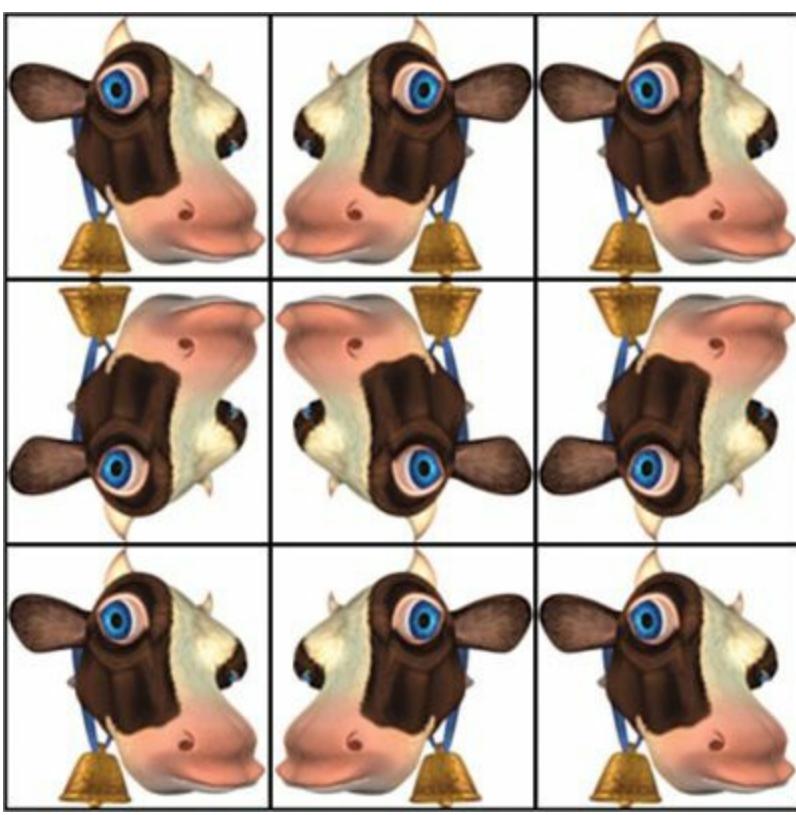
**Figure 5.8** A textured quad in wrap texture address mode. (*Texture by Brian Maricle.*)

#### Note

The black border surrounding the tiled textures in [Figure 5.8](#) is part of the actual texture and is not created as part of the wrapping process. This is included to help illustrate the integer boundaries where wrapping occurs.

## Mirror

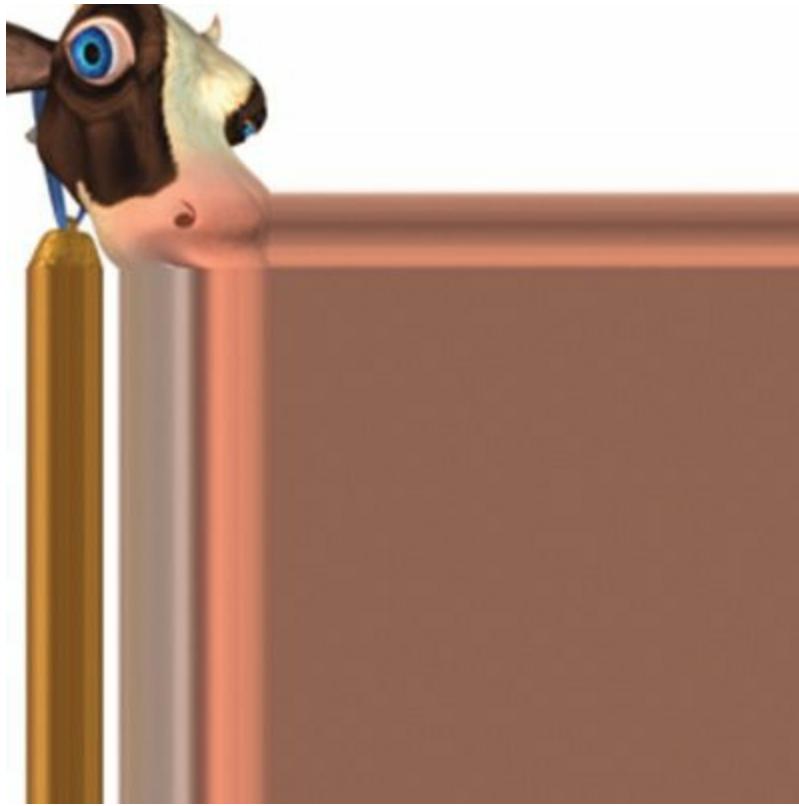
Mirror address mode is similar to wrap mode, except that the texture is mirrored along integer boundaries instead of duplicated. [Figure 5.9](#) shows the results of the same textured quad, but with mirror address mode enabled.



**Figure 5.9** A textured quad in mirror texture address mode. (*Texture by Brian Maricle.*)

## Clamp

In clamp address mode, the texture is not tiled. It is applied once, and all coordinates are clamped to the range  $[0, 1]$ . This has the effect of smearing the color of the texture's edges, as in [Figure 5.10](#).



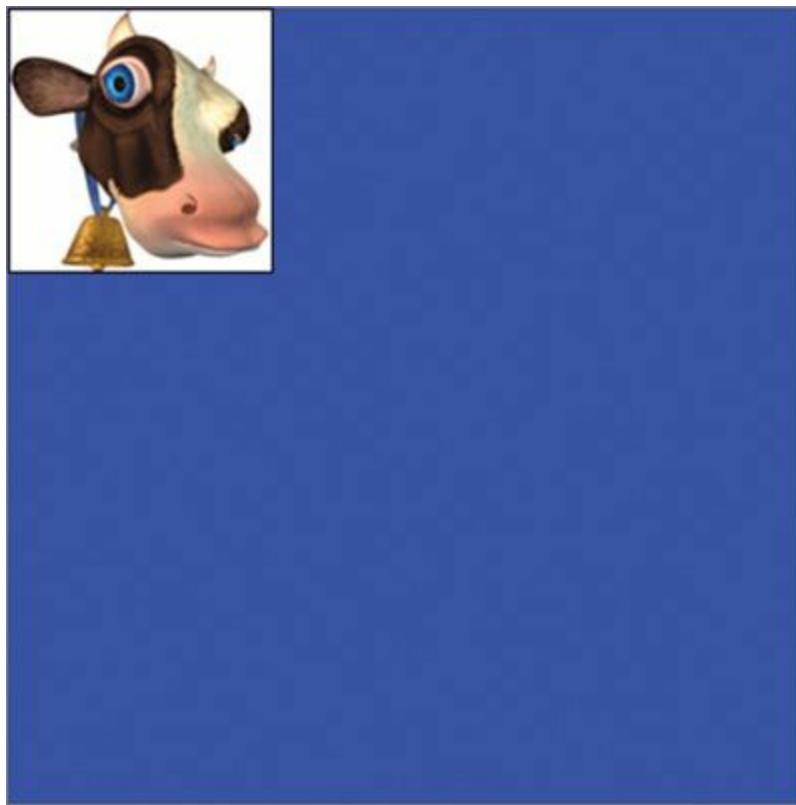
**Figure 5.10** A textured quad in clamp texture address mode. (*Texture by Brian Maricle.*)

## Note

The mapped texture in [Figure 5.10](#) is different than the texture in [Figures 5.8](#) and [5.9](#). The black border has been removed and the image has been cropped so that the edge pixels vary in color. If the original texture had been used, the black border would be the color for every edge pixel and would be the “clamped” value.

## Border

As in clamp mode, border address mode applies the texture just once. But instead of smearing the edge pixels, a border color is used for all coordinates outside the range [0, 1]. [Figure 5.11](#) shows the original, black-bordered cow texture, in border address mode with a blue border color.



**Figure 5.11** A textured quad in border texture address mode. (*Texture by Brian Maricle.*)

## Summary

In this chapter, you learned the details of texture mapping. You explored a complete texture mapping effect and uncovered additional HLSL syntax. You also learned about minification, magnification, mipmapping, and the filtering processes used to sample textures. Finally, you dove into the details of texture addressing modes and discovered how to produce various effects for geometry with texture coordinates that extend outside the range 0 to 1.

Texture mapping is fundamental to modern graphics, and future chapters build on this foundation.

## Exercises

1. Modify the `SamplerState` object in `TextureMapping.fx` to experiment with various permutations of filter modes. Use point, linear, and anisotropic filtering, and observe the results as you zoom in and out with a textured surface.
2. Import a quad with UVs that range outside [0, 1], and experiment with the texture addressing modes in the `SamplerState` object of `TextureMapping.fx`. Specifically, use wrap,

mirror, clamp, and border, and observe the results. The book's companion website has an example quad (in .obj format, a 3D model format that NVIDIA FX Composer supports).

# Chapter 6. Lighting Models

In the real world, you can't see objects without light; an object either reflects a light source or emits light itself. In computer rendering, you simulate the interaction of light to add nuance and interest to your 3D objects. But the interplay of light is an extremely complex process that you can't replicate at interactive frame rates (at least, not yet). Therefore, you use approximations, or models, of how light interacts with 3D objects to add more detail to your scenes. This chapter introduces some of these basic lighting models.

## Ambient Lighting

Ambient light is the seemingly ubiquitous “glow” that exists in a lit environment. Look under your desk, for example, and you’ll still be able to see into the farthest recesses and make out detail, even though no light source is directly illuminating that space. This glow exists because of the countless interactions of light between surfaces. As a ray of light reaches a surface, it is either reflected or absorbed, in total or in part, and that process continues innumerable times. So some light does reach under your desk, even if just a little.

You can achieve a simplified approximation of ambient light by modulating the color of a pixel by some constant value. You could consider this a brightness or intensity filter in which the color of each pixel is multiplied by a value between 0 and 1. As the ambient intensity value approaches 0, the object gets darker. Additionally, you can include color into your ambient lighting model, simulating a light source that isn’t pure white. This requires a modulation value for each of the red, green, and blue (RGB) channels of a pixel. [Listing 6.1](#) presents the code for an ambient lighting effect.

### **Listing 6.1** AmbientLighting.fx

[Click here to view code image](#)

```
***** Resources *****

#define FLIP_TEXTURE_Y 1

cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None";
>;
```

```
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/****** Utility Functions *****/

float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}
```

```

***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);

    return OUT;
}

***** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    OUT = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    OUT.rgb *= AmbientColor.rgb * AmbientColor.a; // Color (.rgb) *
Intensity (.a)

    return OUT;
}

***** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

## AmbientColor Shader Constant

You likely noticed that much of this code is identical to that of the texture mapping effect of the last chapter. Building off previous effects is a pattern going forward.

The first difference is the `AmbientColor` shader constant, a `float4` representing the color and

intensity of your ambient light. The color of the light is stored within the RGB channels, and the intensity is stored within the alpha channel. Also notice that this variable is contained within a new cbuffer, named CBufferPerFrame. Recall from the discussion of constant buffers in [Chapter 4](#), “[Hello, Shaders](#),” that cbuffers are commonly organized according to the intended update frequency of the contained data. In this case, the AmbientColor value will likely be shared across multiple objects and will need updating only with each frame. This is in contrast to the WorldViewProjection object, within CBufferPerObject, which differs for every object using the AmbientLighting effect.

Note also the AMBIENT semantic associated with the AmbientColor constant. As with all shader constants, this semantic is optional, but it’s not a bad idea to mark variables with semantics so that the CPU-side can look up the value by semantic instead of a hard-coded name.

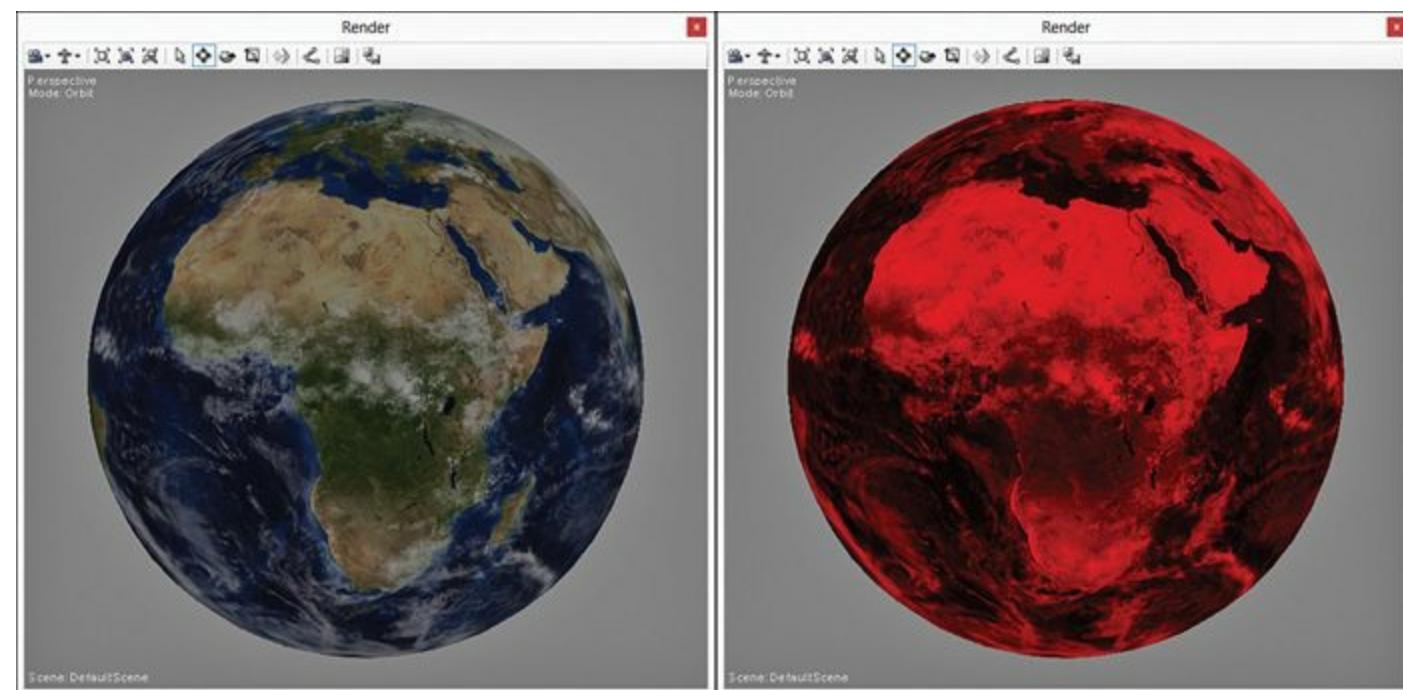
Finally, notice that the AmbientColor constant is initialized with the value `{1.0f, 1.0f, 1.0f, 1.0f}`. This yields a white, full-intensity ambient light that won’t change the output color of your objects. So if you ignore this variable from the CPU side, it won’t negatively impact your output. The rest of the code is the same as the texture mapping effect, up to the pixel shader.

## Ambient Lighting Pixel Shader

The pixel shader starts by sampling the color texture and then modulates the output. Specifically, the RGB channels of the output color are multiplied by `AmbientColor.rgb * AmbientColor.a`. Multiplying a vector (the `AmbientColor.rgb float3`) by a scalar (the `AmbientColor.a float`) performs the multiplication for each component of the vector. Thus, the color of your ambient light is first adjusted by the light’s intensity; then the output RGB channels are multiplied (component-wise) from the resulting product.

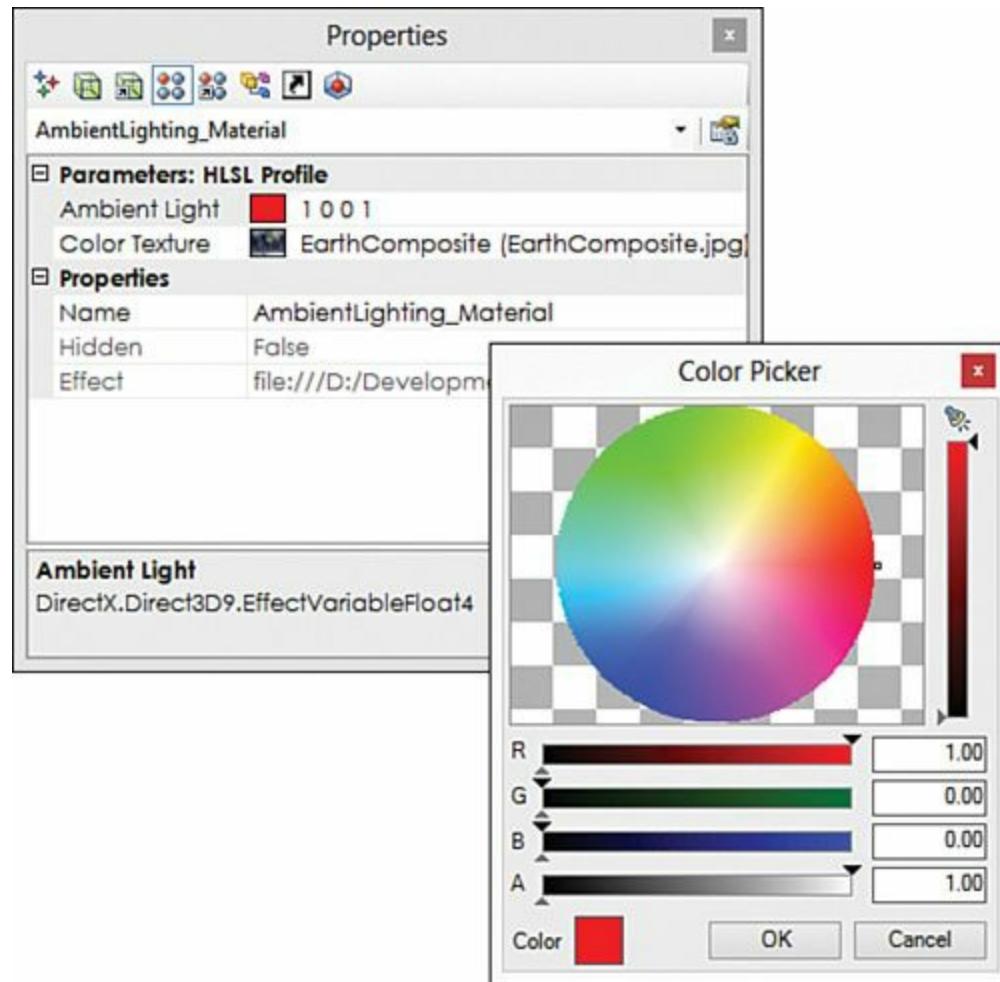
## Ambient Lighting Output

[Figure 6.1](#) shows the output of the ambient lighting effect applied to a sphere with the texture of Earth. On the left, the light is pure white at half intensity (an alpha channel value of `0.5`). On the right, the light is red (a value of `1.0` for the red channel and `0.0` for the blue and green channels) at full intensity (an alpha channel value of `1.0`).



**Figure 6.1** AmbientLighting.fx applied to a sphere with the texture of Earth with a pure-white, half-intensity ambient light (left) and a pure-red, full-intensity ambient light (right). (Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.)

You can modify the values of the ambient light through NVIDIA FX Composer's Properties panel (see [Figure 6.2](#)). Note that the name of the AmbientColor shader constant displays as Ambient Light because of the associated `UIName` annotation, and that the color picker is presented for editing because of the `UIWidget` annotation.



**Figure 6.2** NVIDIA FX Composer's Properties panel showing the Ambient Light constant and color picker dialog.

### Note

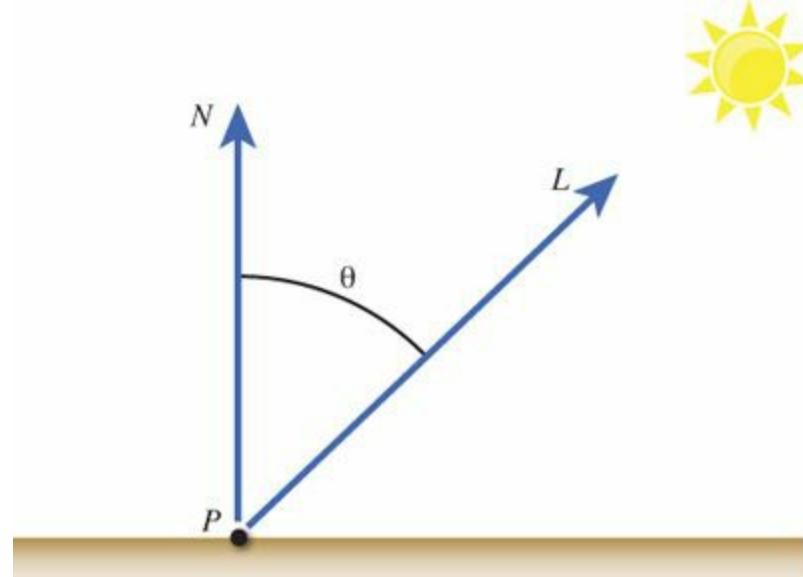
As a general rule, you won't perform mathematical operations from within a shader that produce a constant value. For example, the *ambient color \* ambient intensity* product should be multiplied on the CPU side and passed in as the ambient color shader constant. You do so here only because you're using NVIDIA FX Composer as your CPU-side application and you have limited access to computation before shader execution.

## Diffuse Lighting

Different surfaces reflect light in different ways. Mirrors reflect light with an angle of reflection equal to the angle of incidence. That spooky-looking “glow” you see when you shine a light in a cat’s eye

reveals its retroreflective properties: It reflects light back to the source along a parallel vector in the opposite direction. Diffuse surfaces reflect light equally in all directions.

Perhaps the simplest and most common model for approximating a diffuse surface is Lambert's cosine law. Lambert's cosine law states that the apparent brightness of a surface is directly proportional to the cosine of the angle between the light vector and the surface normal. The light vector describes the direction the light is coming from, and the normal defines the orientation of the surface (which way it's facing). [Figure 6.3](#) illustrates these terms.

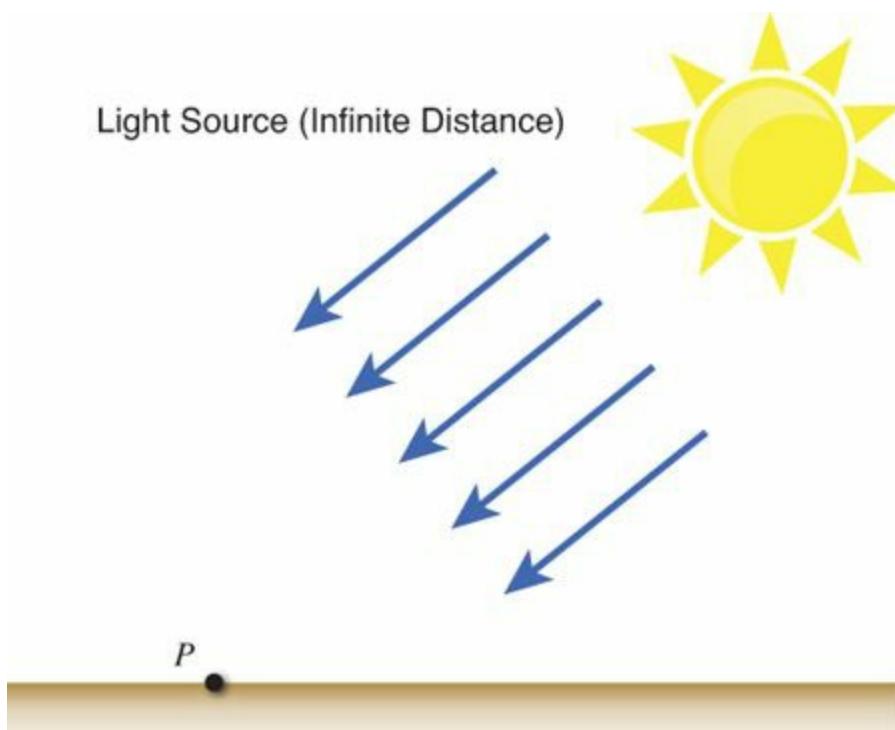


**Figure 6.3** An illustration of a surface normal, a light vector, and Lambert's cosine law.

Recall from our discussion of vectors in [Chapter 2, “A 3D/Math Primer,”](#) that you can use the dot product to find the cosine of the angle between the light vector and the surface normal (given that the two vectors are of unit length). The surface normal can be computed (through a cross-product of two of the edges of a triangle) or, more commonly, supplied for each vertex when loading a 3D object. This leaves the source of the light vector.

## Directional Lights

There are three common types of light that are defined in 3D graphics: directional lights, point lights, and spotlights. A directional light represents a light source that is infinitely far away—it has no position relative to your 3D objects. As such, the light rays that reach your objects are parallel to one another—they all travel in the same direction. The sun (while not infinitely distant) is a good example of a directional light. It's so far away that you cannot discern a difference in the direction of individual rays of light. [Figure 6.4](#) illustrates this concept.



**Figure 6.4** An illustration of a directional light.

To model a directional light, you simply need a three-dimensional vector describing where the light is coming from. You can also include the concept of color and intensity, just as you did for ambient lighting. [Listing 6.2](#) presents the code for a diffuse lighting effect using a single directional light. Quite a bit is going on here, so I recommend that you transpose the code into NVIDIA FX Composer and then examine the effect step by step. (Alternately, you can download the code from the book's companion website.)

## **Listing 6.2** DiffuseLighting.fx

[Click here to view code image](#)

```
#include "include\\Common.fhx"

/***************** Resources ********************

cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};
```

```

float3 LightDirection : DIRECTION <
    string Object = "DirectionalLight0";
    string UIName = "Light Direction";
    string Space = "World";
> = {0.0f, 0.0f, -1.0f};
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>;

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***************** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 LightDirection : TEXCOORD1;
}

```

}

\*\*\*\*\* Vertex Shader \*\*\*\*\*

```
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);
    OUT.LightDirection = normalize(-LightDirection);

    return OUT;
}
```

\*\*\*\*\* Pixel Shader \*\*\*\*\*

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float n_dot_l = dot(lightDirection, normal);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a * color.rgb;

    float3 diffuse = (float3)0;

    if (n_dot_l > 0)
    {
        diffuse = LightColor.rgb * LightColor.a * n_dot_l * color.rgb;
    }

    OUT.rgb = ambient + diffuse;
    OUT.a = color.a;

    return OUT;
}
```

```
***** Techniques *****
```

```
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

## Diffuse Lighting Effect Preamble

The first line of DiffuseLighting.fx performs a C-style #include of a text file containing some common functionality used by your growing library of effects. [Listing 6.3](#) presents the contents of Common.fxh. Notice the header guards surrounding the file, and note that the FLIP\_TEXTURE\_Y define and the get\_corrected\_texture\_coordinate() function have been moved to this file.

### Listing 6.3 Common.fxh

[Click here to view code image](#)

---

```
#ifndef _COMMON_FXH
#define _COMMON_FXH

***** Constants *****

#define FLIP_TEXTURE_Y 1

***** Utility Functions *****

float2 get_corrected_texture_coordinate(float2
textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 -
textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

#endif /* _COMMON_FXH */
```

Next, notice the new `CBufferPerFrame` members: `LightColor` and `LightDirection`. The `LightColor` shader constant has the same function as `AmbientColor`: It represents the color and intensity of the directional light. `LightDirection` stores the direction of the source light in world space. Note the `Object` annotations associated with these two new shader constants. This annotation denotes that the variable can be bound to a scene object within NVIDIA FX Composer. Specifically, you can place lights in the NVIDIA FX Composer Render panel and associate those lights to shader constants marked with the `Object` annotation. We discuss this further in a moment.

Now consider the `World` variable added to `CBufferPerObject`. This value is related to the new `Normal` member of the `VS_INPUT` struct. Surface normals are stored in object space, just like their associated vertices. You use the normal to compute the diffuse color of the pixel by dotting it with the light vector. Because the light is in world space, the normal must also reside in world space and the `World` matrix is used for this transformation. You can't use the `World-ViewProjection` matrix for the transformation because that concatenated matrix would transform the vector to homogeneous space instead of just world space. Note that the `World` potentially contains a scaling transformation, and the surface normal should be a unit vector; thus, normalizing the vector after transformation is important.

## Diffuse Lighting Vertex Shader

Next, inspect the `VS_OUTPUT` struct and its two new members: `Normal` and `LightDirection`. The first passes the transformed surface normal. The second is a little strange, considering the `LightDirection` shader constant. This member exists because the global `LightDirection` stores the direction of the light *from the source*, but you need the direction of the light *from the surface*. Therefore, you invert the global `LightDirection` within the vertex shader and assign it to the corresponding output member. Certainly, you could pass the data in from the CPU side, already in the appropriate direction, and skip the invert operation. Indeed, you should do this. However, NVIDIA FX Composer sends the bound light data *from the source*, so you need to invert the vector to properly preview the effect within the Render panel. Also notice the `normalize()` intrinsic invoked with the inverted light direction. This guarantees that the light direction is of unit length, but it could be omitted if the guarantee came from the CPU side.

## Diffuse Lighting Pixel Shader

Although some similarities to the ambient lighting effect arise, the diffuse lighting pixel shader (reproduced in [Listing 6.4](#)) introduces a lot of new code.

### **Listing 6.4** The Pixel Shader from DiffuseLighting.fx

[Click here to view code image](#)

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float n_dot_l = dot(lightDirection, normal);
```

```

float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
float3 ambient = AmbientColor.rgb * AmbientColor.a *
color.rgb;

float3 diffuse = (float3)0;

if (n_dot_l > 0)
{
    diffuse = LightColor.rgb * LightColor.a * n_dot_l *
color.rgb;
}

OUT.rgb = ambient + diffuse;
OUT.a = color.a;

return OUT;
}

```

---

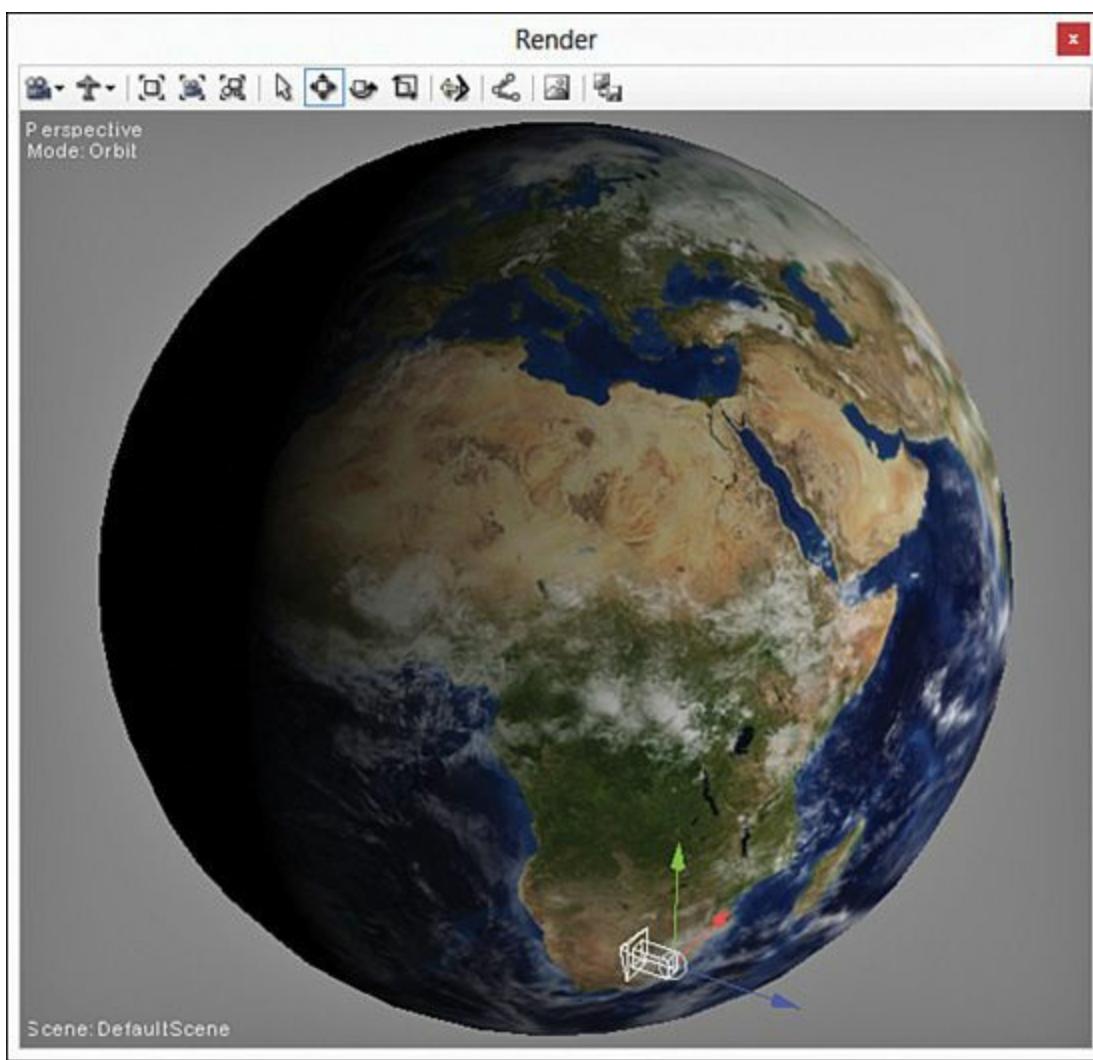
First, notice the texture sampling and the calculation of the ambient term. This is reformed a bit, but those are the same steps as before; I've just separated out the ambient term to clarify the additive process used to produce the final pixel color.

Next, examine the normalization of the incoming Normal and LightDirection members. Remember that data passing through the rasterizer stage is interpolated, and that process can yield non-unit-length vectors. The math errors here are minor, as are the corresponding visual artifacts. So if you're running up against performance issues, you can easily omit these operations.

Next, you take the dot product of the light direction and the surface normal and use that value to construct the diffuse color term. Notice the `if`-statement ensuring that `n_dot_l` is greater than zero. A negative value for the dot product indicates that the light source is *behind* the surface and, therefore, should not apply. Applicable dot product values are between `0.0` and `1.0`. A value of `0.0` indicates that the light vector is parallel to the surface (and, therefore, has no effect), and a value of `1.0` signifies that the light is orthogonal to the surface (and imparts the highest intensity of light). The diffuse term is then created by multiplying the sampled color by the directional light color and intensity and the dot product.

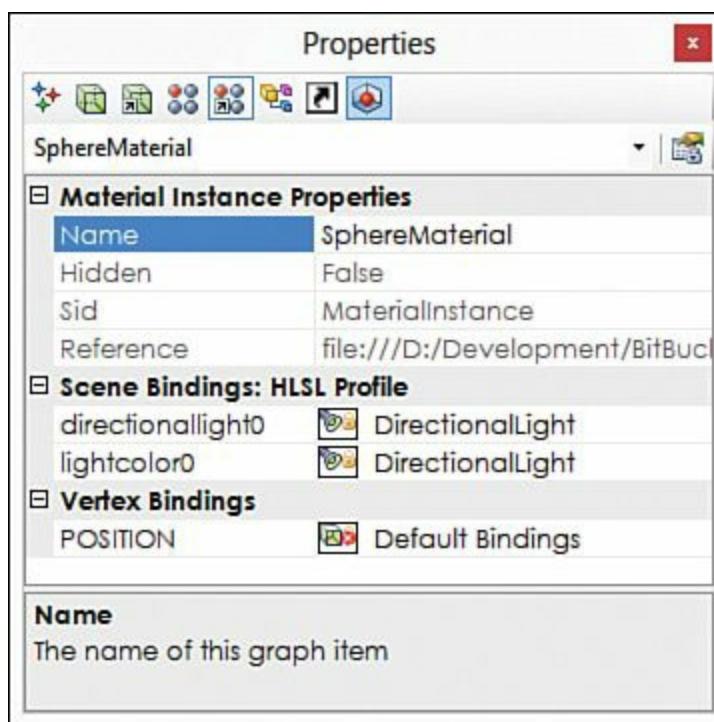
## Diffuse Lighting Output

The final pixel color is produced by adding the ambient and diffuse terms. The color texture supplies the alpha channel. [Figure 6.5](#) shows the diffuse lighting effect applied to a sphere with the same Earth texture as in [Figure 6.1](#). Observe how the image gets darker as the surfaces face farther from the light.



**Figure 6.5** DiffuseLighting.fx applied to a sphere with the texture of Earth. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

Also notice the directional light at the bottom of this image. NVIDIA FX Composer has options to create ambient, point, spot, and directional lights in the Render panel. To do so, click one of the main toolbar buttons or choose the appropriate selection from the **Create** menu. For the data of the directional light to be passed into your effect, you must bind your directional light to the **LightColor** and **LightDirection** shader constants. This is where the **Object** annotations come into play. To bind the light, select the sphere in the Render panel and click the **Material Instance Properties** button in the Properties panel. This is the fifth button in the Properties panel toolbar (see [Figure 6.6](#)). Then choose your directional light for the **directionallight0** and **lightcolor0** bindings. With your light bound to shader constants, any changes you make to the light will be immediately visible. For example, you can rotate your light and observe how the lit areas of the sphere coincide with the direction of the light. But if you translate the light, you'll see no impact because directional lights have no true position. The position you see for the proxy model of the directional light has no impact on the data passed into the effect.



**Figure 6.6** Material Instance Properties within NVIDIA FX Composer.

You can also change the color and intensity (stored in the alpha channel) of the directional light. To do so, select the light in the Render panel and open the picker for the **Color** property in the Properties panel. For the image in [Figure 6.5](#), the directional light is pure white with full intensity, and the ambient light has an intensity of 0 (effectively disabling the ambient light).

### Warning

NVIDIA FX Composer supports both automatic and manual binding. When automatic binding is enabled, NVIDIA FX Composer attempts to bind your lights to the appropriate shader constants. Unfortunately, this doesn't always succeed. Check your material instance properties to determine what has been bound, and manually change the settings, if necessary.

Note, however, that when you recompile your effect, all manual bindings are lost.

## Specular Highlights

When you simulate a diffuse surface, you provide a matte, nonglossy appearance. This is appropriate for myriad objects in a scene and forms the base for much of your lighting. But you'll also want to model shiny surfaces that simulate, for example, polished metal or marble flooring. You can achieve that shiny, glossy look through specular highlights.

### Phong

Several approaches approximate specular reflection. The first one we cover is the specular component of the Phong reflection model, named for its inventor, Bui Tuong Phong from the University of Utah.

Unlike diffuse shading, specular highlights depend on where the viewer (the camera) is with respect to the surface. You can observe this yourself; just look at a shiny object and notice how the sheen of it

alters as you change your perspective. Phong's model states that the specular highlight depends on the angle between this view direction and the light's reflection vector. In equation form:

$$Specular_{Phong} = (R \bullet V)^s$$

where,  $R$  is the reflection vector,  $V$  is the view direction, and  $s$  specifies the size of the highlight. A smaller highlight is produced with a larger specular exponent. The reflection vector is computed with the following equation:

$$R = 2 * (N \bullet L) * N - L$$

where  $N$  is the surface normal, and  $L$  is the light vector.

[Listing 6.5](#) presents the code for a Phong effect. As before, transpose this code into NVIDIA FX Composer, compile it, and assign the associated material to an object in the Render panel. The following sections review the specifics of the effect.

## **Listing 6.5** Phong.fx

[Click here to view code image](#)

---

```
#include "include\Common.fhx"

/********************* Resources ********************/

cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightDirection : DIRECTION <
        string Object = "DirectionalLight0";
        string UIName = "Light Direction";
        string Space = "World";
    > = {0.0f, 0.0f, -1.0f};

    float3 CameraPosition : CAMERAPOSITION < string
    UIWidget="None"; >;
}
```

```

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
    UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
    > = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***************** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

```



```

float3 specular = (float3)0;

if (n_dot_l > 0)
{
    diffuse = LightColor.rgb * LightColor.a *
saturate(n_dot_l) *
color.rgb;

    // R = 2 * (N.L) * N - L
    float3 reflectionVector = normalize(2 * n_dot_l * normal
- lightDirection);

    // specular = R.V^n with gloss map in color texture's
alpha
channel
    specular = SpecularColor.rgb * SpecularColor.a *
min(pow(saturate
(dot(reflectionVector, viewDirection)), SpecularPower),
color.w);
}

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}

technique10 main10
{
pass p0
{
SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
SetGeometryShader(NULL);
SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

SetRasterizerState(DisableCulling);
}
}

```

## Phong Preamble

Compared with the diffuse lighting effect, the `CBufferPerFrame` block has only one addition: `CameraPosition`. This shader constant stores the location of the camera and determines the view direction. NVIDIA FX Composer automatically binds this constant to the Render panel's camera when you specify the `CAMERAPOSITION` semantic.

The `CBufferPerObject` block has two additions: constants for `SpecularColor` and `SpecularPower`. `SpecularColor` has the same function as the colors for the ambient light and

directional light; it designates the color and intensity of the specular highlight. Having this extra “knob” enables you to tweak the specular highlight independently from the directional light. The SpecularPower refers to the exponent value  $s$  in the Phong specular equation.

Next, notice the newly added ViewDirection vector in the VS\_OUTPUT struct. This passes the calculated view direction along to the rasterizer stage.

## Phong Vertex Shader

Within the vertex shader, the view direction is calculated by subtracting the position of the vertex from the camera’s position. However, both positions must reside in the same coordinate space, so first you transform the vertex position (`IN.ObjectPosition`) by the world matrix.

## Phong Pixel Shader

The pixel shader now includes operations for computing the specular term. These steps apply only if the directional light “shines” on the surface, so you encapsulate these statements within the `n_dot_l > 0` conditional.

An added twist to computing the specular term requires explanation. [Listing 6.6](#) pulls out just these statements.

### **Listing 6.6** Computing the Specular Term Within Phong.fx

[Click here to view code image](#)

---

```
// specular = R.V^n with gloss map stored in color texture's
// alpha
channel
specular = SpecularColor.rgb * SpecularColor.a *
min(pow(saturate(dot
(reflectionVector, viewDirection)), SpecularPower), color.w);
```

---

First, notice the comment stating that the term is calculated through the Phong specular equation. This is performed through the `pow()` intrinsic, which accepts the dotted reflection and view direction vectors raised to the SpecularPower exponent. The `saturate()` intrinsic clamps the output of the dot product between 0.0 and 1.0, thereby eliminating negative angles.

The comment also indicates that the calculated value is modulated by a **gloss map** that is stored in the color texture’s alpha channel. A gloss map, or **specular map**, either is a stand-alone texture (typically just a single-channel texture format) or is included as part of another incoming texture (as in this example). A specular map limits the computed specular term according to the input of the texture artist. Consider the Earth texture used throughout this chapter. It makes sense that only the water portions of Earth should appear shiny and that the land should have a matte appearance. A specular map allows for this on a per-pixel basis. The `min()` intrinsic returns the lesser of the computed specular term and the value stored in the alpha channel (the w channel). Thus, a value of 0.0 in the specular map would eliminate any specular even if the computed value would otherwise yield the highest specular intensity (a value of 1.0). [Figure 6.7](#) shows the specular map of Earth used for this section.

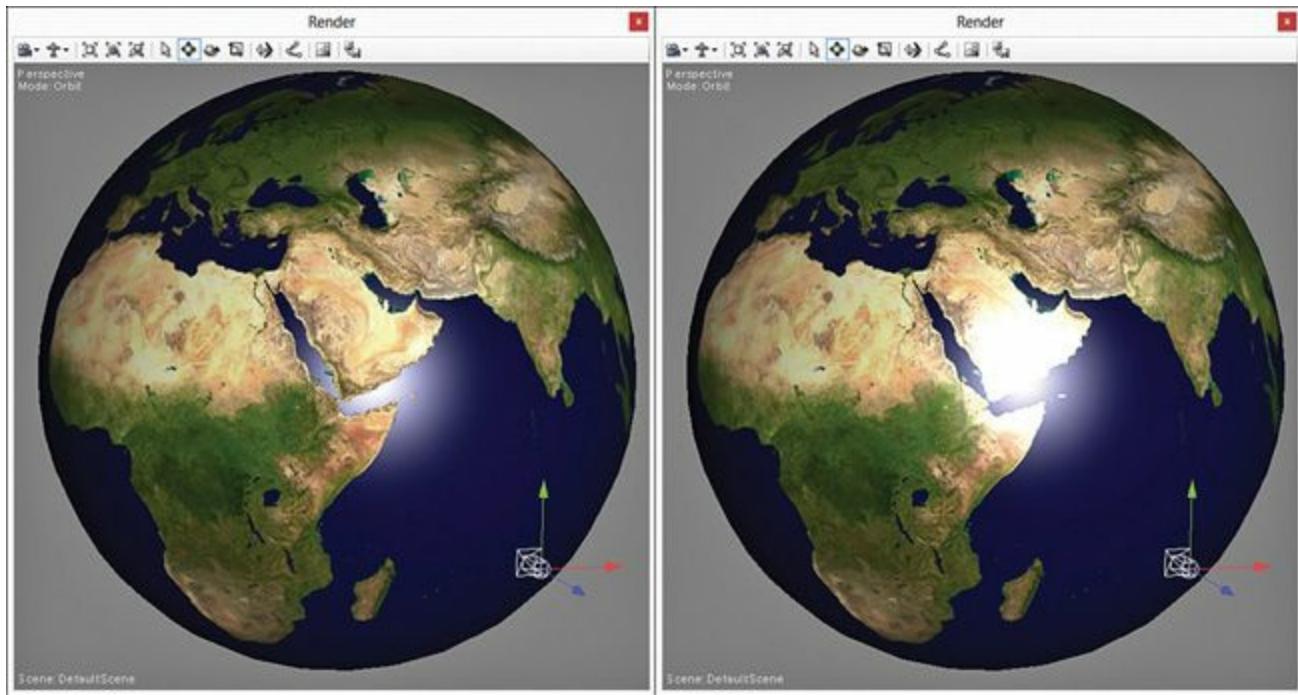


**Figure 6.7** Specular map for the Earth texture. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

The final pixel color is produced through the combination of the ambient, diffuse, and specular terms. The alpha channel is set to a value of 1.0 (fully opaque) rather than from the color texture because you are using the alpha channel as a specular map instead of opacity. If you require both opacity and a specular map, you need to supply a second texture.

## Phong Output

[Figure 6.8](#) shows the result of the Phong effect applied to a sphere with a texture of Earth (without cloud coverage) with and without a specular map. Both images share the same color channels, but notice the difference in the specular highlight between the land and water for the texture with a specular map (left).



**Figure 6.8** Phong .fx applied to a sphere with a texture of Earth with a specular map (left) and without one (right). (*Texture from Reto Stöckli, NASA Earth Observatory.*)

## Blinn-Phong

In 1977, Jim Blinn developed a modification to the Phong specular component that replaces the reflection vector with a **half-vector**. This vector lies halfway between the view and light vectors and is computed as:

$$H = \frac{L+V}{|L+V|}$$

Instead of dotting the Phong reflection vector against the view direction, Blinn-Phong dots the half-vector with the surface normal and raises the result to an exponent. The final equation is:

$$\text{Specular}_{\text{Blinn - Phong}} = (N \cdot H)^s$$

## Blinn-Phong Pixel Shader

The preamble and vertex shader for the Blinn-Phong effect is identical to the Phong effect. Therefore, [Listing 6.7](#) presents only the modified pixel shader. You can find the complete effect on the book's companion website.

### **Listing 6.7** The Pixel Shader from BlinnPhong.fx

[Click here to view code image](#)

---

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(lightDirection, normal);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a *
color.rgb;

    float3 diffuse = (float3)0;
    float3 specular = (float3)0;

    if (n_dot_l > 0)
    {
        diffuse = LightColor.rgb * LightColor.a *
saturate(n_dot_l) *
color.rgb;

        float3 halfVector = normalize(lightDirection +
viewDirection);

        //specular = N.H^s w/ gloss map stored in color
```

```

texture's alpha
channel
    specular = SpecularColor.rgb * SpecularColor.a *
min(pow(saturate(dot(normal, halfVector)), SpecularPower),
color.w);
}

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}

```

---

Notice the removal of the reflection vector and the addition of the half-vector calculation. As before, the specular map limits the calculated specular term. The output of Blinn-Phong is visually identical to that of Phong, although you have to tweak the value of the exponent.

## Blinn-Phong with Intrinsics

HLSL provides an intrinsic function, `lit()`, to compute the Lambertian diffuse and Blinn-Phong specular components for you. A good rule of thumb is to use built-in functions wherever possible because they are likely optimized or implemented in hardware. Thus, you can rewrite your Blinn-Phong pixel shader as in [Listing 6.8](#).

### **Listing 6.8** Utility Functions and the Pixel Shader from `BlinnPhongIntrinsics.fx`

[Click here to view code image](#)

---

```

float3 get_vector_color_contribution(float4 light, float3 color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float3 get_scalar_color_contribution(float4 light, float color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float3 viewDirection = normalize(IN.ViewDirection);

```

```

    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb);
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w));

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

```

The differences in this pixel shader are significant. Gone are the explicit calculations for the diffuse and specular terms, along with the containing conditional. New is the call to `lit()`, which accepts the values for `n_dot_l` and `n_dot_h` and the specular exponent. The `lit()` intrinsic performs the same diffuse and specular calculations and returns coefficients for these terms in a `float4`. The `x` and `w` components of the returned vector are always 1. The diffuse coefficient is stored in the `y` component and specular in `z`, which you use to produce the corresponding terms.

Also notice the utility functions that perform the multiplications for the light color and intensity. These just make code maintenance easier when, for example, you want to modulate the light color by the intensity on the CPU side instead of calculating a constant within the pixel shader. Store these functions in your `Common.fxh` file for use across your shader library.

## Blinn-Phong vs. Phong

Because the output between Phong and Blinn-Phong is identical, the difference between these models is a question of efficiency. Blinn-Phong could be slower because of the square root (from the normalization of the half-vector). However, because you'll use the HLSL `lit()` intrinsic, it's likely that the built-in implementation of Blinn-Phong yields a minor speed improvement over Phong. I show both models for the sake of instruction and completeness. We use the `lit()` intrinsic for all future effects that have diffuse and specular terms.

## Summary

In this chapter, you learned about a series of introductory lighting models, including ambient lighting, diffuse lighting, and specular highlights. You implemented effects to simulate such lighting and extended your knowledge of HLSL. The next chapter introduces you to even more lighting concepts, including point lights and spotlights, and explores how to use multiple lights in your scenes.

## Exercises

1. Experiment with the ambient lighting effect. Modify the ambient light's color and intensity from within the Properties panel of NVIDIA FX Composer.
2. Explore the output of the diffuse lighting effect. Create a directional light within your scene, and bind it to the color and directional light shader inputs. Rotate the light, modify its color, and observe the results. Additionally, create an Object annotation for the `AmbientLight` constant, and create and bind an ambient light to this object. Then modify the ambient light in the scene and observe the impact to the rendered output.
3. Vary the specular power in the Phong effect and observe the results. Experiment with different combinations of color and intensity for the ambient, directional, and specular color constants.
4. Implement the complete Blinn-Phong effect, and compare its output to the Phong effect.
5. Implement a Blinn-Phong effect using the HLSL `lit()` intrinsic, and compare its output to that of the Blinn-Phong effect you implemented in the previous exercise.

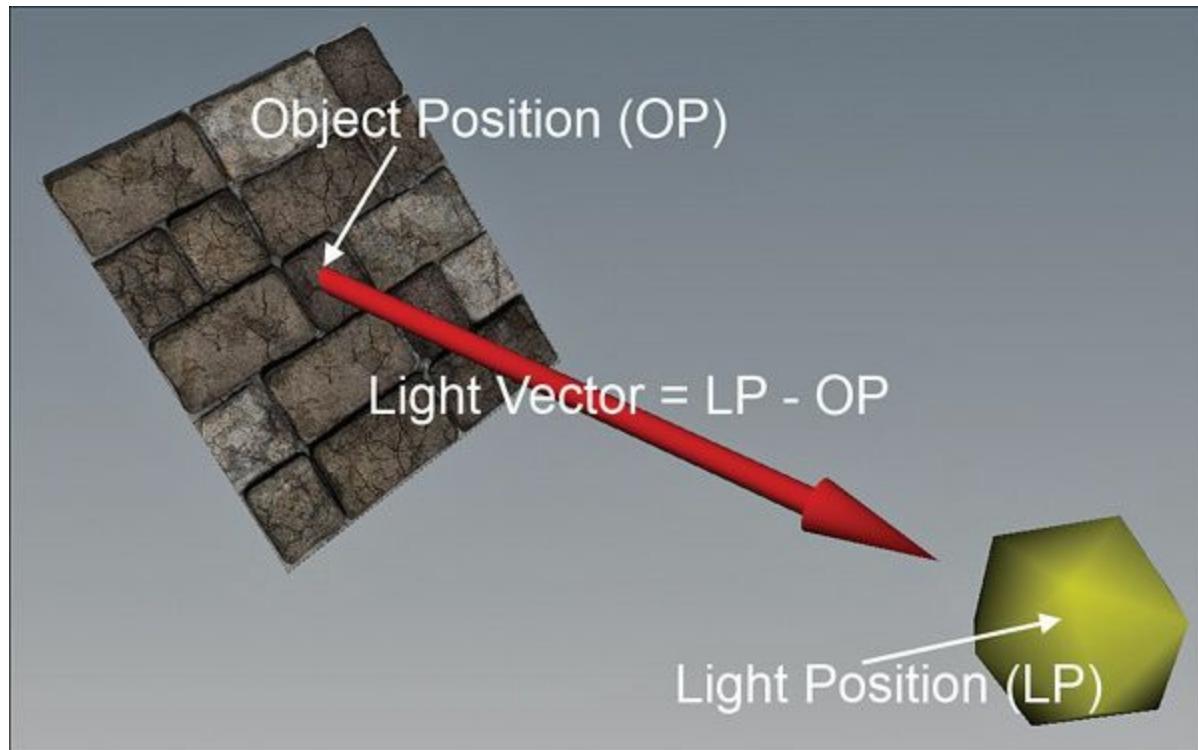
# Chapter 7. Additional Lighting Models

This chapter expands on the previous lighting models by introducing point lights, spotlights, and multiple lights for your scenes. Along the way, you'll refine your experience with HLSL and the effect framework.

## Point Lights

A point light in a scene is like a light bulb—it has a position that is local to your surroundings, and it radiates light in all directions. This is in contrast to a directional light, which is infinitely far away and whose light appears to come from a single direction. Directional lights have no concept of “moving the light;” point lights have no concept of “rotating the light.”

You can use the same diffuse and specular lighting models with point lights, just as with directional lights. But with point lights, you supply the light’s position and then must compute the light vector. This is a simple matter of subtracting the position of the object from the position of the light (where both positions are in world space). [Figure 7.1](#) illustrates a point light and the computed light vector.



**Figure 7.1** An illustration of a point light and the computed light vector. (*Stone wall texture by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

Additionally, because point lights have a specific location, you can attenuate the light with respect to the distance from the surface. The farther the light is from the surface, the less brightness it imparts. [Listing 7.1](#) presents an effect for a single point light.

### **Listing 7.1** PointLight.fx

[Click here to view code image](#)

---

```
#include "include\\Common.fxh"
```

```
***** Resources *****
```

```
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightPosition : POSITION <
        string Object = "PointLight0";
        string UIName = "Light Position";
        string Space = "World";
    > = {0.0f, 0.0f, 0.0f};

    float LightRadius <
        string UIName = "Light Radius";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 100.0;
        float UIStep = 1.0;
    > = {10.0f};

    float3 CameraPosition : CAMERAPOSITION < string
UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
    > = {1.0f};
}
```

```
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
    > = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float4 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
};

/****** Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
```

```

VS_OUTPUT OUT = (VS_OUTPUT)0;

OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);

float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
float3 lightDirection = LightPosition - worldPosition;
OUT.LightDirection.xyz = normalize(lightDirection);
OUT.LightDirection.w = saturate(1.0f -
(length(lightDirection) /
LightRadius)); // Attenuation

OUT.ViewDirection = normalize(CameraPosition -
worldPosition);

return OUT;
}

```

\*\*\*\*\* Pixel Shader \*\*\*\*\*

```

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection.xyz);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.LightDirection.w;
    float3 specular =
get_scalar_color_contribution(SpecularColor,

```

```

min(lightCoefficients.z, color.w) * IN.LightDirection.w;

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}

***** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

---

## Point Light Preamble

As before, this effect builds from previous work—specifically, the Blinn-Phong effect using the `lit()` intrinsic. The `CBufferPerFrame` block has been modified to remove the light direction and replace it with `LightPosition`. This constant represents the position of the light in world space. Furthermore, a `LightRadius` shader constant has been added to define the distance within which the light has influence. Outside this radius, the light has no impact.

The `CBufferPerObject`, `ColorTexture`, and `ColorSampler` objects have not changed. Nor has the `VS_INPUT` struct—it still accepts the vertex position (in object space), a set of texture coordinates, and the normal. But the `VS_OUTPUT` struct's `LightDirection` vector is now a `float4` instead of a `float3`, and it's now required in this effect; compare this with the Blinn-Phong effect, which could have used the global `LightDirection` shader constant instead. The additional channel in the `LightDirection` shader input stores the attenuation factor of the light. Both the direction and the attenuation are computed in the vertex shader.

### Note

Storing the attenuation factor in the `.w` component of the `LightDirection` vector is just an efficiency trick. Because that channel isn't otherwise employed, you can use it instead of occupying another output register unnecessarily. The attenuation factor bears no relation to the `xyz` components of the `LightDirection` vector.

## Point Light Vertex and Pixel Shader

The vertex shader stores the computed light direction in a temporary `float3`, which is then

normalized and copied to the corresponding output struct. Next, the light attenuation factor is calculated with this equation:

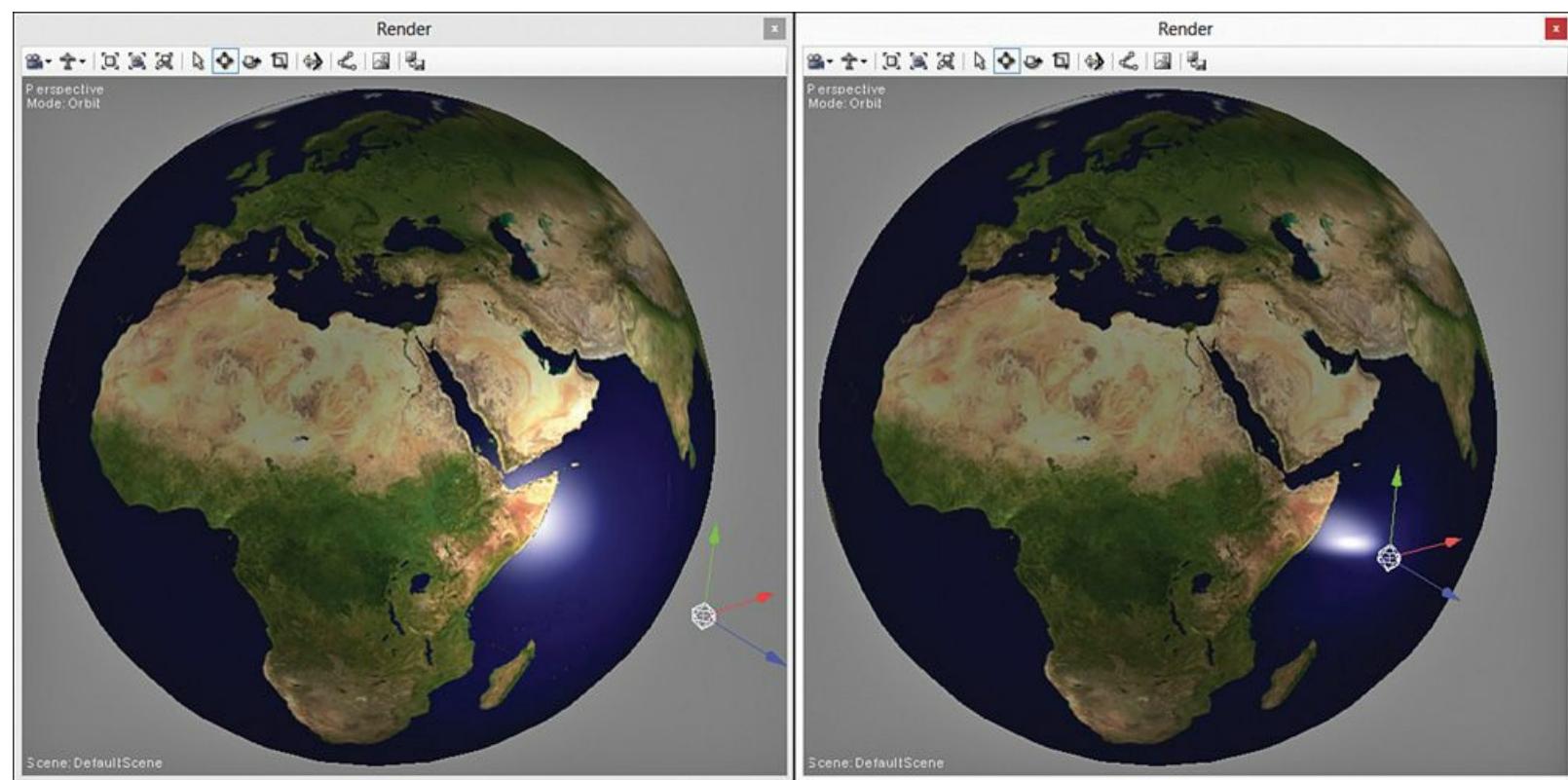
$$\text{Attenuation} = 1.0 - \frac{|\text{lightDirection}|}{\text{lightRadius}}$$

Because you stored the prenormalized vector in the `lightDirection` variable, you can use its length to find the distance between the surface and the light. Whenever the distance is greater than or equal to the light's radius, the calculated attenuation will be less than or equal to zero. The `saturate()` call keeps the final value out of negative territory.

The pixel shader is identical to that of Blinn-Phong, except that now the diffuse and specular terms are multiplied by the attenuation factor.

## Point Light Output

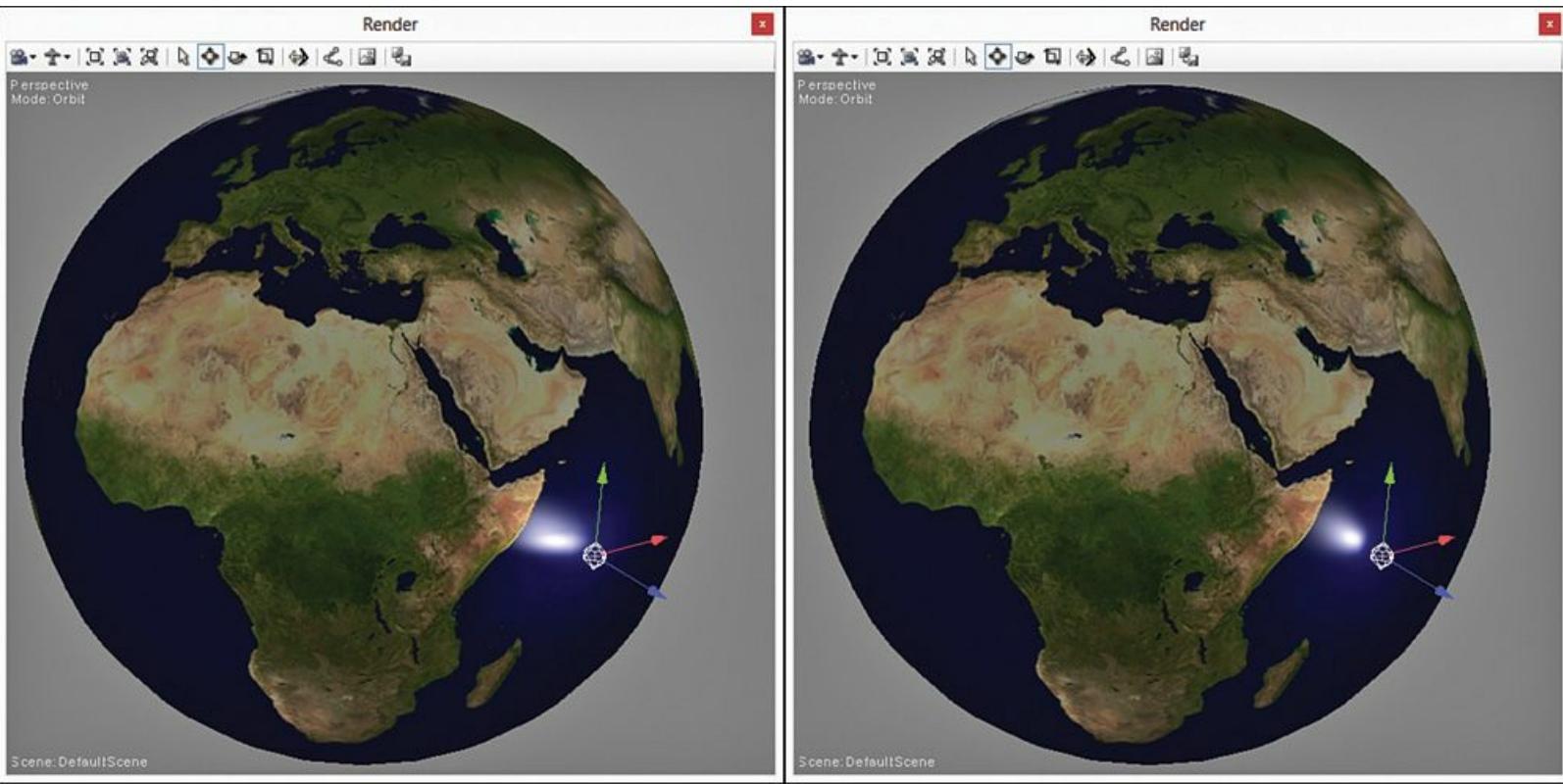
[Figure 7.2](#) shows the results of the point light applied to a sphere with the Earth texture using a half-intensity ambient light, a full-intensity specular highlight, and a full-intensity point light. The lights and the specular color are pure white.



**Figure 7.2** `PointLight.fx` applied to a sphere with a texture of Earth, using a half-intensity ambient light, a full-intensity specular highlight, and a full-intensity point light. All light colors are pure white. To the left, the point light is positioned farther away from the sphere; to the right, it is closer. (*Texture from Reto Stöckli, NASA Earth Observatory.*)

## Point Light Modifications

In the right-side image of [Figure 7.2](#), a glitch in the appearance of the specular highlight is caused by computing the light direction within the vertex shader. It's apparent only when the point light is close to the object because that's when the light vector varies (more significantly) between pixels. [Figure 7.3](#) shows a comparison of the output when the light direction is computed in the vertex shader and then in the pixel shader.



**Figure 7.3** Comparison of the specular highlight produced with a per-vertex calculation (left) of the light direction and a per-pixel calculation (right). (*Texture from Reto Stöckli, NASA Earth Observatory.*)

[Listing 7.2](#) presents an abbreviation of the effect for computing the light direction in the pixel shader. This listing displays only the code changes between the two iterations.

### **Listing 7.2** Per-Pixel Calculation of the View Direction

[Click here to view code image](#)

```
struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
};

/***************** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT) 0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
```

```

    OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));

    return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = normalize(LightPosition
- IN.WorldPosition);
    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

```

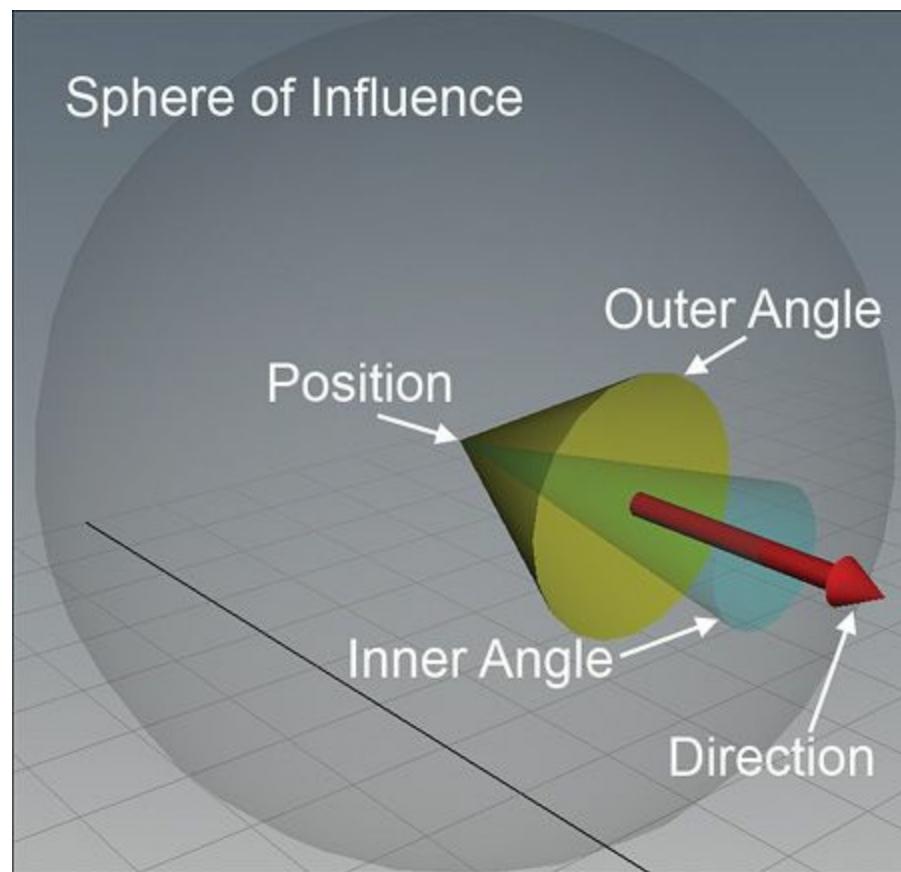
}

The computation for the light direction is in world space. Therefore, you must pass the world-space position of the surface through the `VS_OUTPUT` struct. And because the light vector is no longer computed in the vertex shader, you can remove it from the `VS_OUTPUT` struct. However, you don't need to move the attenuation calculation to the pixel shader (which would be much more expensive). At the close distance your point light will be, to have a visible impact on the light direction, any attenuation will be negligible. So you add/keep the `Attenuation` float in the vertex shader output struct. This does require that you compute the light direction twice: once for the attenuation calculation and again for the more accurate light direction in the pixel shader. But this cost is minimal compared to the expense of calculating the attenuation factor in the pixel shader.

## Spotlights

A spotlight is a combination of a directional light and a point light. It has a position in world space, but it shines light in a specific direction. Furthermore, a spotlight attenuates with distance, as a point light does, but its light also attenuates around its source direction. You can think of a spotlight as a virtual flashlight, complete with a focus beam that falls off as the light gets farther from the center.

You can model a spotlight with a position, a direction, a radius of influence, a color and intensity, and floating-point values for inner and outer angles that describe how light radiates around the focus beam. [Figure 7.4](#) illustrates these elements.



**Figure 7.4** An illustration of a spotlight.

[Listing 7.3](#) presents the code for an effect with a single spotlight. As before, I recommend that you transcribe this code into NVIDIA FX Composer. We discuss the specifics in the following sections.

## Listing 7.3 Spotlight.fx

[Click here to view code image](#)

```
#include "include\\Common.fhx"

/**************** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Spotlight Color";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightPosition : POSITION <
        string Object = "SpotLightPosition0";
        string UIName = "Spotlight Position";
        string Space = "World";
> = {0.0f, 0.0f, 0.0f};

    float3 LightLookAt : DIRECTION <
        string Object = "SpotLightDirection0";
        string UIName = "Spotlight Direction";
        string Space = "World";
> = {0.0f, 0.0f, -1.0f};

    float LightRadius <
        string UIName = "Spotlight Radius";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 100.0;
        float UIStep = 1.0;
> = {10.0f};

    float SpotLightInnerAngle <
        string UIName = "Spotlight Inner Angle";
        string UIWidget = "slider";
        float UIMin = 0.5;
        float UIMax = 1.0;
        float UIStep = 0.01;
```

```

> = {0.75f};

float SpotLightOuterAngle <
    string UIName = "Spotlight Outer Angle";
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 0.5;
    float UIStep = 0.01;
> = {0.25f};

    float3 CameraPosition : CAMERAPOSITION < string
UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
> = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

```

```

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
    float3 LightLookAt : TEXCOORD3;
};

/****** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);

    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - length(lightDirection) / LightRadius);

    OUT.LightLookAt = -LightLookAt;

    return OUT;
}

/****** Pixel Shader *****/

```

```

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float)0;

    float3 lightDirection = normalize(LightPosition
- IN.WorldPosition);
    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);
    float3 lightLookAt = normalize(IN.LightLookAt);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    float spotFactor = 0.0f;
    float lightAngle = dot(lightLookAt, lightDirection);
    if (lightAngle > 0.0f)
    {
        spotFactor = smoothstep(SpotLightOuterAngle,
SpotLightInnerAngle, lightAngle);
    }

    OUT.rgb = ambient + (spotFactor * (diffuse + specular));
    OUT.a = 1.0f;

    return OUT;
}

***** Techniques *****/
technique10 main10

```

```

{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

---

## Spotlight Preamble

The `CBufferPerFrame` block contains self-describing members for `LightColor`, `LightPosition`, and `LightRadius`. The member `LightLookAt` defines the direction of the focus beam. It's named this way to avoid confusion with the computed `lightDirection` vector that specifies where the light is in relation to the surface.

`CBufferPerFrame` also contains members for `SpotLightOuterAngle` and `SpotLightInnerAngle` and (through annotations) limits these values to `[0.0, 0.5]` and `[0.5, 1.0]`, respectively. As you'll see in the pixel shader, these values represent minimum and maximum values for a spotlight coefficient that is dependent on the angle between the focus beam and the computed `lightDirection`.

The only change to your shader inputs is the new `LightLookAt` vector in the `VS_OUTPUT` struct. This is a pass-through variable that stores the inverted `LightLookAt` global shader constant. As with directional lights, this is necessary because NVIDIA FX Composer sends a light's direction *from the light* and you need it *from the surface*. In your own CPU-side applications, you can send this data in properly and omit this shader input.

## Spotlight Vertex and Pixel Shader

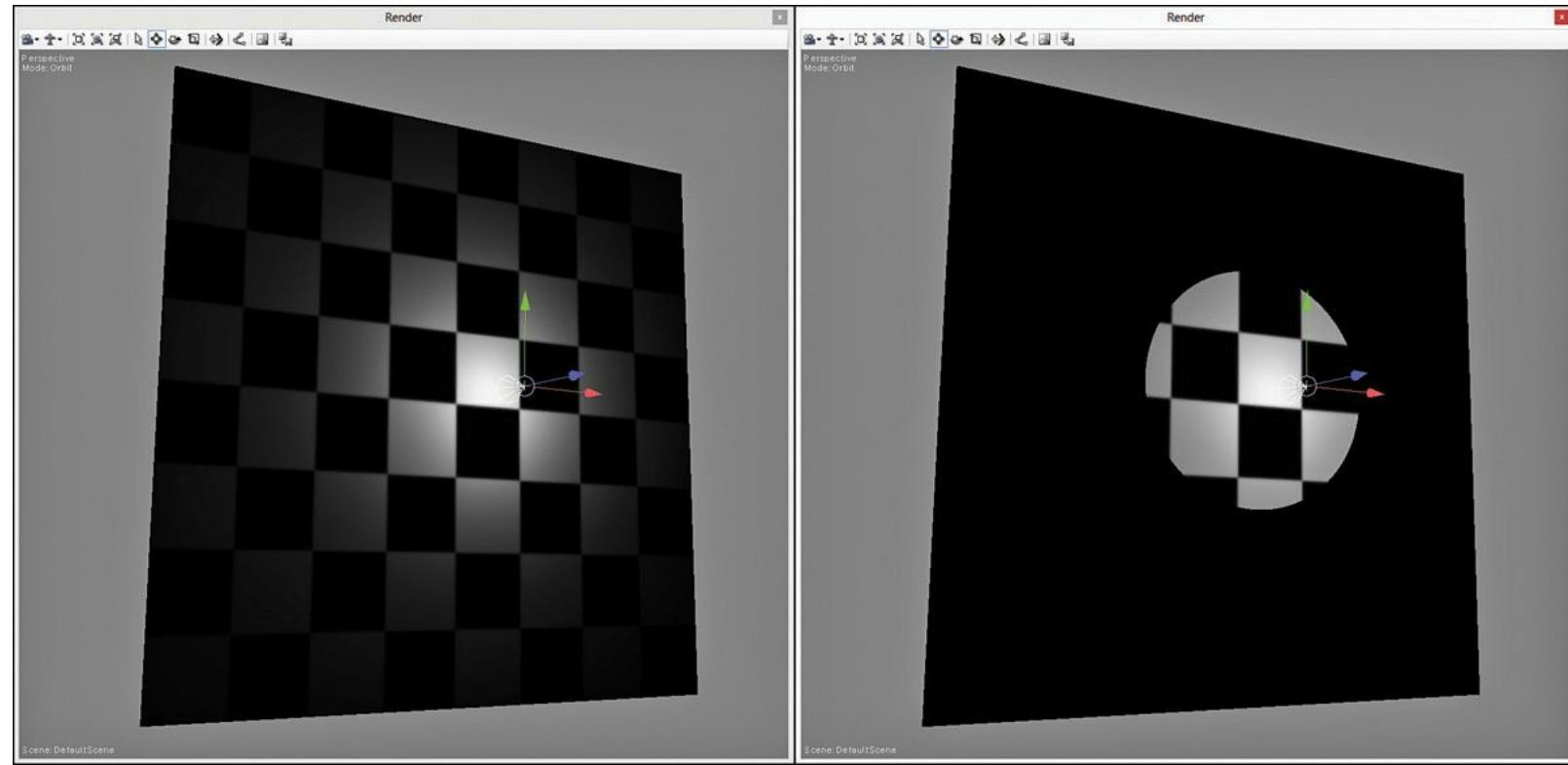
The spotlight vertex shader is identical to the second iteration of the point light effect, except for the inclusion of the inverted `LightLookAt` statement. Similarly, the pixel shader borrows heavily from the point light shader, except for the addition of the `spotFactor` calculation. Indeed, the diffuse and specular terms are computed as if the light were a point light. The spotlight's *look at* vector and its angle to the `lightDirection` determines the final color of these terms.

You determine the `lightAngle` with a dot product of the `lightLookAt` and `lightDirection` vectors. If it is positive, the `spotFactor` falls within the values specified as the outer and inner "angles" of the spotlight. Using the HLSL `smoothstep()` intrinsic, the calculated `lightAngle` is interpolated between `SpotLightOuterAngle` and `SpotLightInnerAngle`. Limiting the values of these constants to `[0.0, 0.5]` and `[0.5, 1.0]` should now make sense. If you specify `0.0` as the outer angle and `1.0` as the inner angle, your `spotFactor` moves from `1.0` to `0.0` as the surface faces farther *away* from the focus beam.

In the composition of the final pixel color, the `spotFactor` modulates the diffuse and specular terms. The ambient term is unaffected by any additional light sources.

## Spotlight Output

[Figure 7.5](#) presents the results of the spotlight effect. In this figure, the effect has been applied to a plane with a checkerboard texture. The ambient and specular terms have been disabled (by dialing their intensities to zero), and the spotlight is pure white at full intensity. In the left image, the inner and outer spotlight angles are  $0.0$  and  $1.0$ , respectively. In the right image, the inner and outer spotlight angles are both  $0.5$ . Note the hard-edged boundary these settings create.



**Figure 7.5** Spotlight .fx applied to a plane with a checkerboard texture. To the left, the outer-to-inner range is  $[0.0, 1.0]$ ; to the right, both values are set to  $0.5$ .

## Multiple Lights

So far, you have been working with single light sources (discounting ambient light). But there's no reason you cannot combine directional, point, and spotlights in the same effect, nor multiple lights of the same type. Your limitations are a matter of performance and the instruction count available with a given shader model.

[Listings 7.4](#) and [7.5](#) present an effect with support for multiple point lights. This code has quite a few new constructs, so we start by discussing the updated Common .fxh file, which contains new support structs and utility functions. Then we dive into the details of the effect.

### Listing 7.4 Common .fxh with Support for Multiple Point Lights

[Click here to view code image](#)

---

```
#ifndef _COMMON_FXH
#define _COMMON_FXH

***** Constants *****/
#define FLIP_TEXTURE_Y 1
```

```
***** Data Structures *****

struct POINT_LIGHT
{
    float3 Position;
    float LightRadius;
    float4 Color;
};

struct LIGHT_CONTRIBUTION_DATA
{
    float4 Color;
    float3 Normal;
    float3 ViewDirection;
    float4 LightColor;
    float4 LightDirection;
    float4 SpecularColor;
    float SpecularPower;
};

***** Utility Functions *****

float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

float3 get_vector_color_contribution(float4 light, float3 color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float3 get_scalar_color_contribution(float4 light, float color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float4 get_light_data(float3 lightPosition, float3
```

```

worldPosition, float
lightRadius)
{
    float4 lightData;
    float3 lightDirection = lightPosition - worldPosition;

    lightData.xyz = normalize(lightDirection);
    lightData.w = saturate(1.0f - length(lightDirection) /
lightRadius); // Attenuation

    return lightData;
}

float3 get_light_contribution(LIGHT_CONTRIBUTION_DATA IN)
{
    float3 lightDirection = IN.LightDirection.xyz;
    float n_dot_l = dot(IN.Normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
IN.ViewDirection);
    float n_dot_h = dot(IN.Normal, halfVector);

    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
IN.SpecularPower);
    float3 diffuse =
get_vector_color_contribution(IN.LightColor,
lightCoefficients.y * IN.Color.rgb) * IN.LightDirection.w;
    float3 specular =
get_scalar_color_contribution(IN.SpecularColor,
min(lightCoefficients.z, IN.Color.w)) * IN.LightDirection.w *
IN.LightColor.w;

    return (diffuse + specular);
}

#endif /* _COMMON_FXH */

```

## Support Structures and Utility Functions

Each point light requires a position, a color, and a radius. So if you want to support four point lights, you need four of each of these. The `POINT_LIGHT` struct bundles these members together so that you can create an array of point lights in a single declaration, such as `POINT_LIGHT PointLights [NUM_LIGHTS];`. Notice that the `POINT_LIGHT` data members are not marked with semantics or annotations. You have little reason to mark the members of a struct with semantics when the intent is to use the struct within an array, and annotations cannot be used for struct members.

The second support struct is `LIGHT_CONTRIBUTION_DATA`. You populate this structure with data for a surface's color and normal, along with the view direction, light color, light direction, specular

color, and specular power—the elements you've used thus far to implement diffuse and specular terms for a single light. This data type is used for the parameter to `get_light_contribution()`, which computes the diffuse and specular terms through similar steps as before and returns their combined color value as a single `float3`. An additional utility function, `get_light_data()`, computes the light direction and attenuation for subsequent inclusion in the `LIGHT_CONTRIBUTION_DATA` struct. [Listing 7.5](#) presents the code for the multiple point light effect, which uses these structures and utility functions.

## **Listing 7.5** MultiplePointLights.fx

[Click here to view code image](#)

---

```
#include "include\Common.fhx"

/********************* Resources *****/
#define NUM_LIGHTS 4

cbuffer CBufferPerFrame
{
    POINT_LIGHT PointLights[NUM_LIGHTS];

    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float3 CameraPosition : CAMERAPOSITION < string
UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
```

```
    float UIStep = 1.0;
> = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 WorldPosition : POSITION;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
};

/****** Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
```

```

    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0) World).xyz);

    return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN, uniform int lightCount) :
SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);
    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);

    LIGHT_CONTRIBUTION_DATA lightContributionData;
    lightContributionData.Color = color;
    lightContributionData.Normal = normal;
    lightContributionData.ViewDirection = viewDirection;
    lightContributionData.SpecularColor = SpecularColor;
    lightContributionData.SpecularPower = SpecularPower;

    float3 totalLightContribution = (float3)0;

    [unroll]
    for (int i = 0; i < lightCount; i++)
    {
        lightContributionData.LightDirection = get_light_
data(PointLights[i].Position, IN.WorldPosition, PointLights[i].
LightRadius);
        lightContributionData.LightColor = PointLights[i].Color;
        totalLightContribution += get_light_contribution
(lightContributionData);
    }

    OUT.rgb = ambient + totalLightContribution;
    OUT.a = 1.0f;

    return OUT;
}

```

```
}

***** Techniques *****

technique10 Lights1
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(1)));

        SetRasterizerState(DisableCulling);
    }
}

technique10 Lights2
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(2)));

        SetRasterizerState(DisableCulling);
    }
}

technique10 Lights3
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(3)));

        SetRasterizerState(DisableCulling);
    }
}

technique10 Lights4
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(4)));
    }
}
```

```
        SetRasterizerState(DisableCulling) ;  
    }  
}
```

---

## Multiple Point Lights Preamble

The multiple point lights effect begins with the `NUM_LIGHTS` definition, which controls the number of elements in the `PointLights` array. Extending or reducing the number of supported lights is as simple as modifying this macro and the corresponding techniques (discussed shortly). The ambient color and camera position remain the same for the `CBufferPerFrame` struct. Indeed, the remainder of the preamble is identical to that of the single point light effect, except that the `Attenuation` member has been removed from the `VS_OUTPUT` struct. The attenuation factor for each light is now computed within the pixel shader.

## Multiple Point Lights Vertex and Pixel Shader

The vertex shader is extremely simple and doesn't compute any data specific to an individual light. This responsibility falls to the pixel shader. The pixel shader begins by sampling the color texture and computing the ambient term. Then it populates a `LIGHT_CONTRIBUTION_DATA` variable with values that are immutable between lights—specifically, the sampled surface color, the surface normal, the view direction, and the specular color and power.

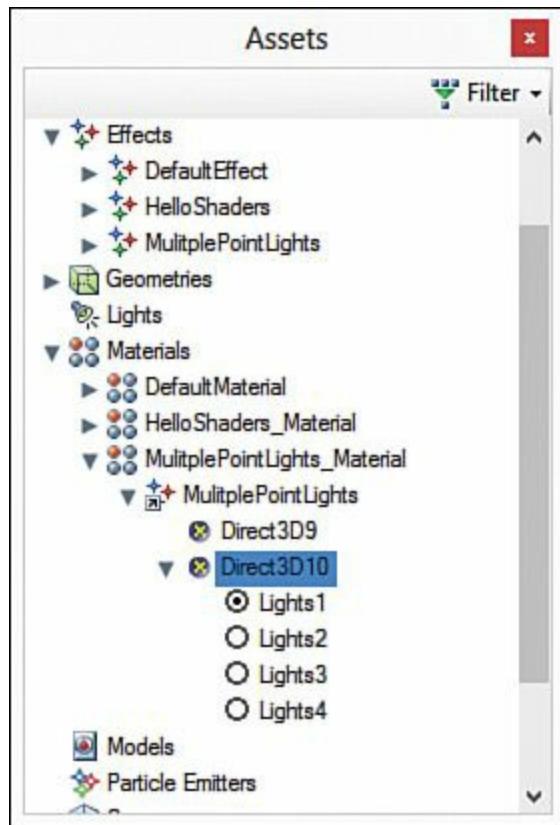
Next, the pixel shader performs a loop through the number of lights specified by the `uniform` shader parameter `lightCount`. This parameter doesn't come from the rasterizer stage and doesn't vary per pixel. Instead, the `lightCount` parameter is specified when the shader is compiled, which is done within a technique. So this value is constant at compile time but is accessed as if it were a regular variable. With uniform parameters, the HLSL compiler actually builds multiple versions of the pixel shader, one for each variation of the `lightCount` parameter supplied by a technique. The HLSL compiler then can **unroll** the loop and eliminate any performance penalties incurred by dynamic flow control.

Within the loop, the variable members of the `lightContribution` struct are assigned and the light's diffuse and specular terms are calculated with a call to `get_light_contribution()`. The influence of each light is added to `totalLightContribution`, which is finally combined with the ambient term to produce the final color of the pixel.

## Multiple Point Lights Techniques

This is the first effect with which you've used multiple techniques, and here they vary the `lightCount` parameter of the pixel shader. Specifically, this effect has four techniques, named `Lights1`, `Lights2`, `Lights3`, and `Lights4`; the name corresponds to the number of lights enabled within the effect. The `lightCount` parameter is specified inside the `Compiler-Shader()` call, which ignores the nonuniform shader inputs and matches the supplied value to the first uniform parameter.

The CPU-side application selects one of these techniques to execute. For instance, if three active point lights are in the vicinity of an object, the CPU-side application would select the `Lights3` technique. From within NVIDIA FX Composer, expand the multiple point lights material (from the Assets panel) to select a particular technique (see [Figure 7.6](#)).



**Figure 7.6** Options for selecting multiple techniques within the Assets panel of NVIDIA FX Composer.

You can also use this method of multiple techniques without the notion of uniform parameters. For example, you might have *Low-*, *Medium-*, and *High-Quality* vertex and/or pixel shaders that correspond to the capabilities of the video hardware. Or you could have multiple uniform parameters and myriad technique permutations. As with all programming, you can use multiple ways to achieve a result. I encourage you to experiment.

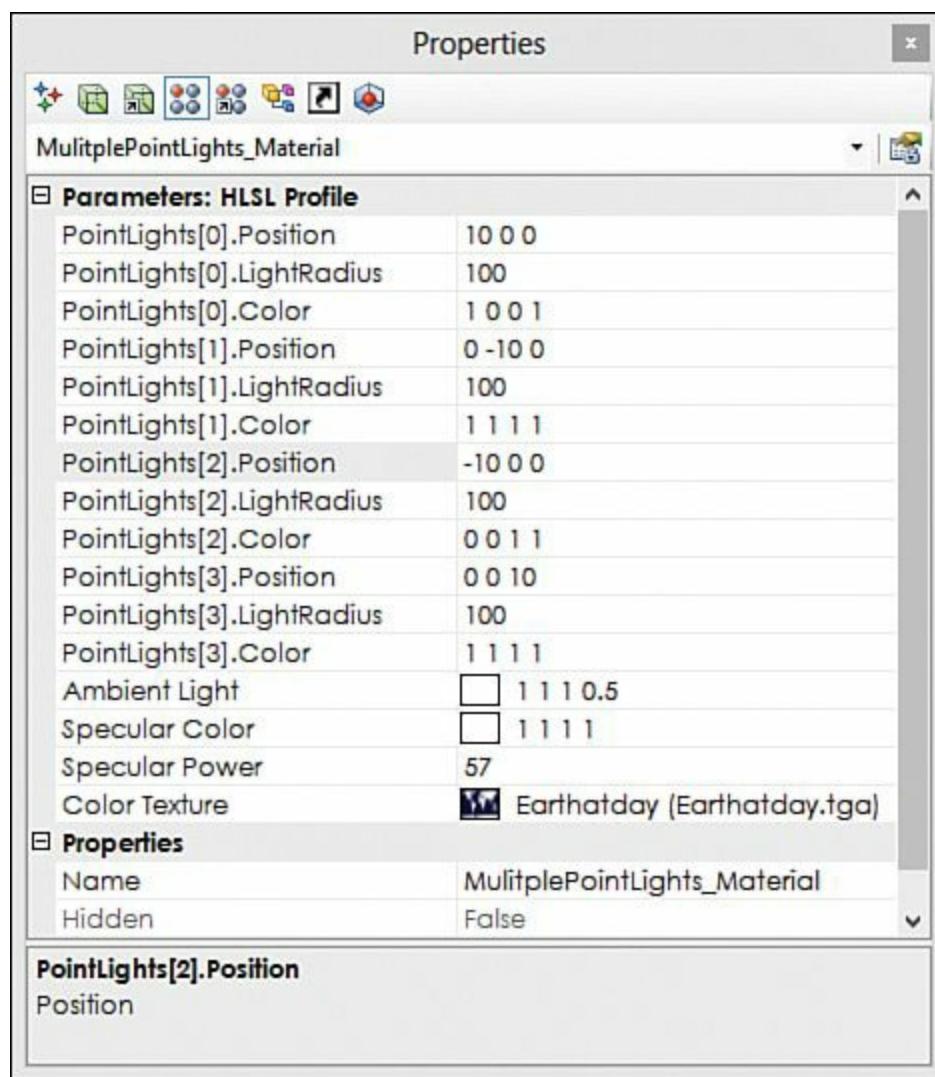
## Multiple Point Lights Output

[Figure 7.7](#) shows the results of the multiple point lights effect with four lights enabled.



**Figure 7.7** `MultiplePointLights.fx`, with four lights enabled, applied to a sphere with a texture of Earth. (*Texture from Reto Stöckli, NASA Earth Observatory.*)

Each light is positioned 10 units away from the origin, along the x-, y-, and z-axes. Note the reddish tint to Australia and bluish-purple hue on Africa. The lights illuminating those areas are pure red and blue, and the lights toward the bottom and at the center are pure white. [Figure 7.8](#) shows the specific positions, colors, and radii. Also notice that you must manually specify the positions and colors of the lights because binding annotations aren't permitted on struct members.



**Figure 7.8** The Properties panel of NVIDIA FX Composer, showing the data supplied to the multiple point lights effect.

## Summary

In this chapter, you learned about point lights, spotlights, and multiple lights for your scenes. You implemented effects for each of these light systems and refined your HLSL skills. You extended your set of common structures and utility functions to support your growing library of effects, and you discovered uniform shader parameters and their use with multiple HLSL techniques. You've come to the end of your introduction to HLSL; the coming chapters introduce you to more intermediate and advanced topics.

## Exercises

1. Experiment with the point light effect. Create a point light in NVIDIA FX Composer's Render panel, and then bind it to your material. Change the light's position, rotation, color, and intensity, and observe the results.
2. Implement the modification to the point light effect, which performs the light direction calculation in the pixel shader. Compare the output to the original point light effect.
3. Explore the spotlight effect. Create a plane in NVIDIA FX Composer's Render panel, and assign it your spotlight material. Then create and bind a spotlight. Change the light's position, rotation, color, and intensity, and observe the results.

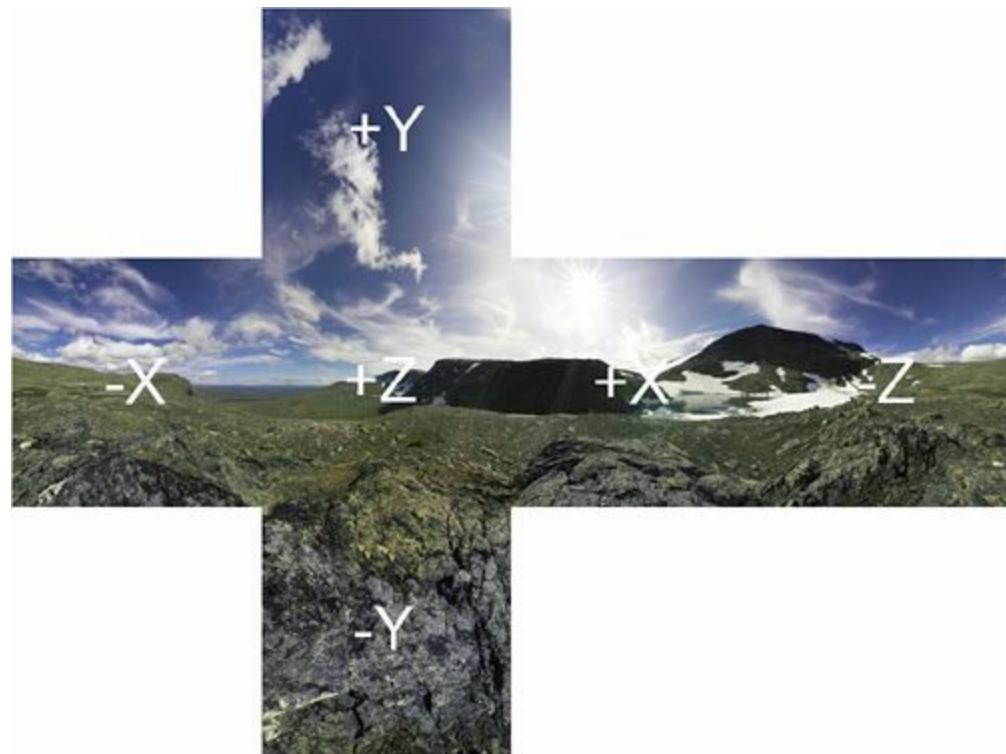
4. Experiment with the multiple point lights effect. Select different techniques to enable one, two, three, and four lights. Vary the data for each of the lights, and observe the results.
5. Extend the multiple point lights effect to support up to eight lights.
6. Write an effect that supports directional, point, and spotlights—one of each type. Ensure that the specular highlight is modulated by the intensity of each light.

# Chapter 8. Gleaming the Cube

This chapter presents an assortment of graphics topics, including skybox rendering, environment mapping, fog, and color blending. The first couple effects share a common theme: They are implemented through texture cubes. And all the material takes you farther along the path of graphics enlightenment (pun intended).

## Texture Cubes

A texture cube (also known as a cube map) is a set of six 2D textures. Each texture corresponds to a face of an axis-aligned cube (typically) centered on the world space origin. These textures can be stored individually or can all reside in the same file using a format such as DDS (discussed in [Chapter 3, “Tools of the Trade”](#)). [Figure 8.1](#) shows a texture cube stored in a single file, with each of its faces labeled accordingly.

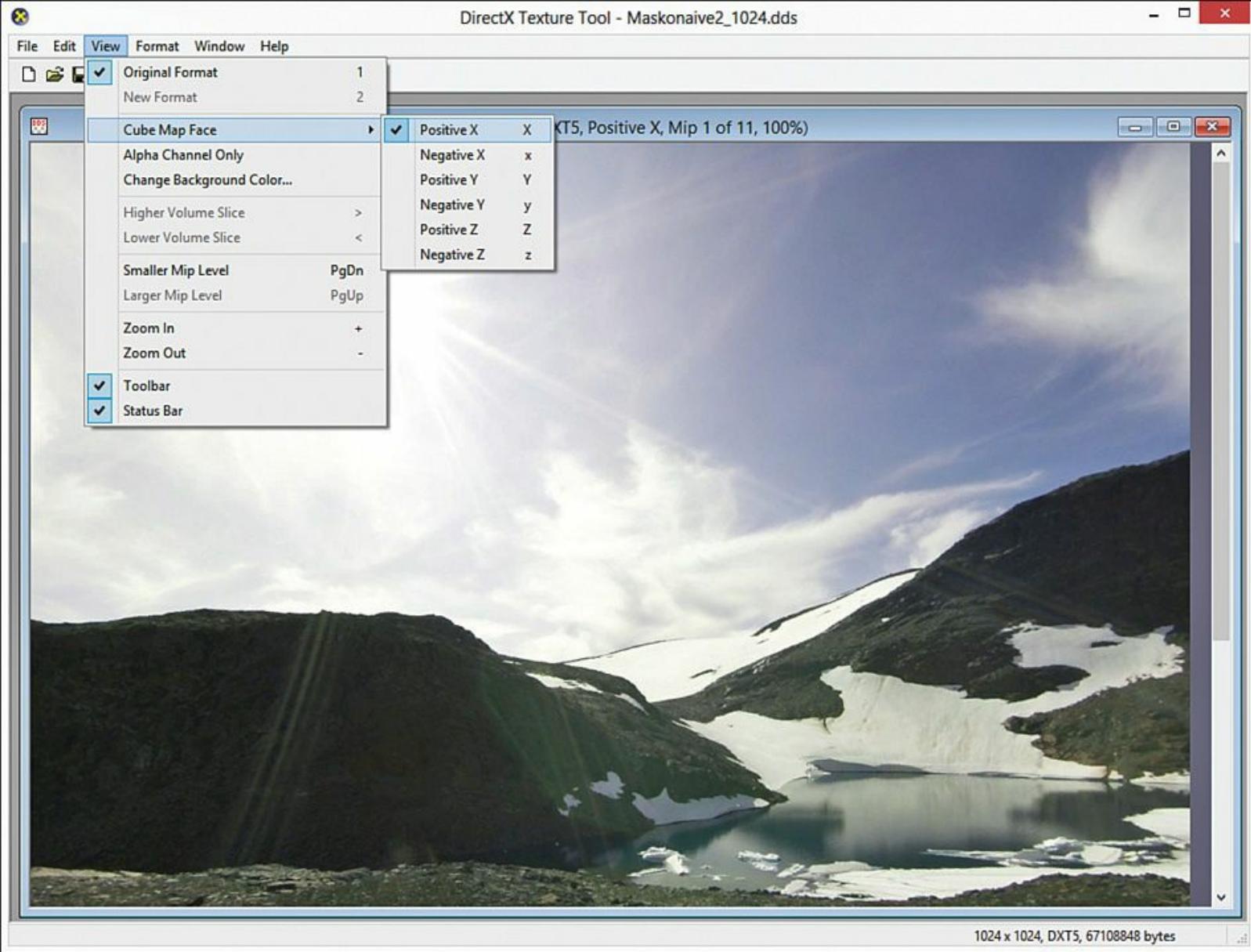


**Figure 8.1** A texture cube. Each face is labeled with the corresponding axis. (*Texture by Emil Persson.*)

## Creating Texture Cubes

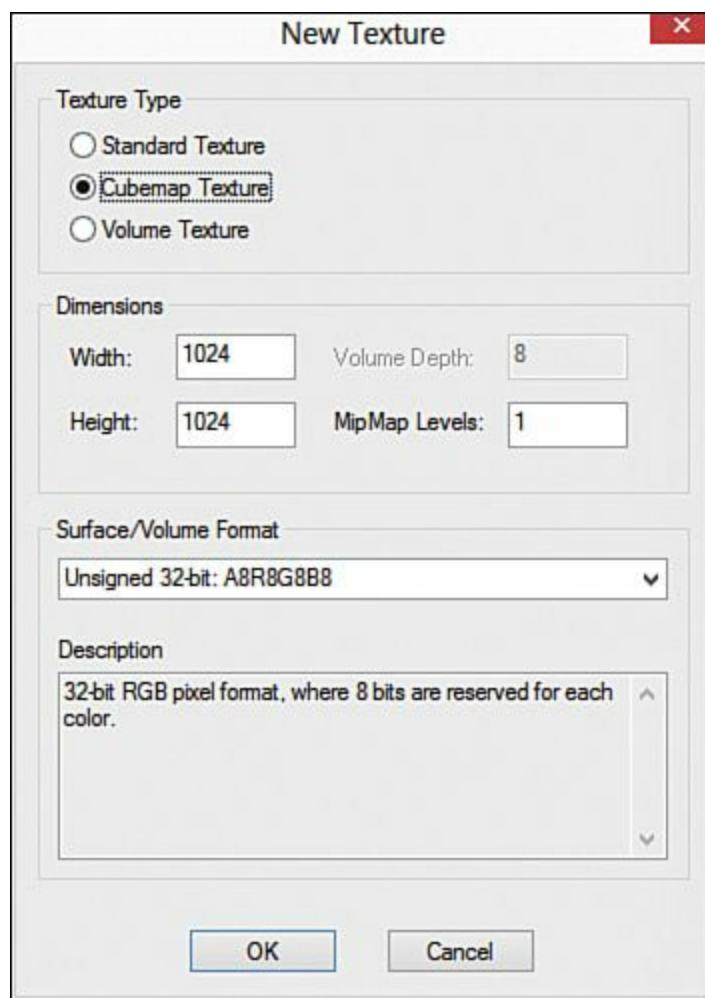
You can use a variety of tools to create texture cubes. NVIDIA, for example, includes the nvDXT command-line tool as part of its DDS Utilities package (the book’s companion website provides a link). This tool accepts a list of individual textures to compile into a texture cube and store in DDS format.

Microsoft includes the DirectX Texture Tool as part of the stand-alone DirectX SDK installation (although not as part of the Windows SDK). [Figure 8.2](#) shows the DirectX Texture Tool with a texture cube loaded and the **Cube Map Face** menu open. Use this menu to view a specific face of the cube map.



**Figure 8.2** The DirectX Texture Tool. (*Texture by Emil Persson.*)

To create a new cube map, choose **File**, **New Texture** and specify **Cubemap Texture** for the texture type (see [Figure 8.3](#)). Then specify the resolution of your texture, the number of mip levels, and the texture format. With a new *empty* texture cube created, you populate the faces by selecting a face through **View**, **Cube Map Face** and then choosing **File**, **Open Onto This Cubemap Face** to assign an image.



**Figure 8.3** The New Texture dialog box of the DirectX Texture Tool.

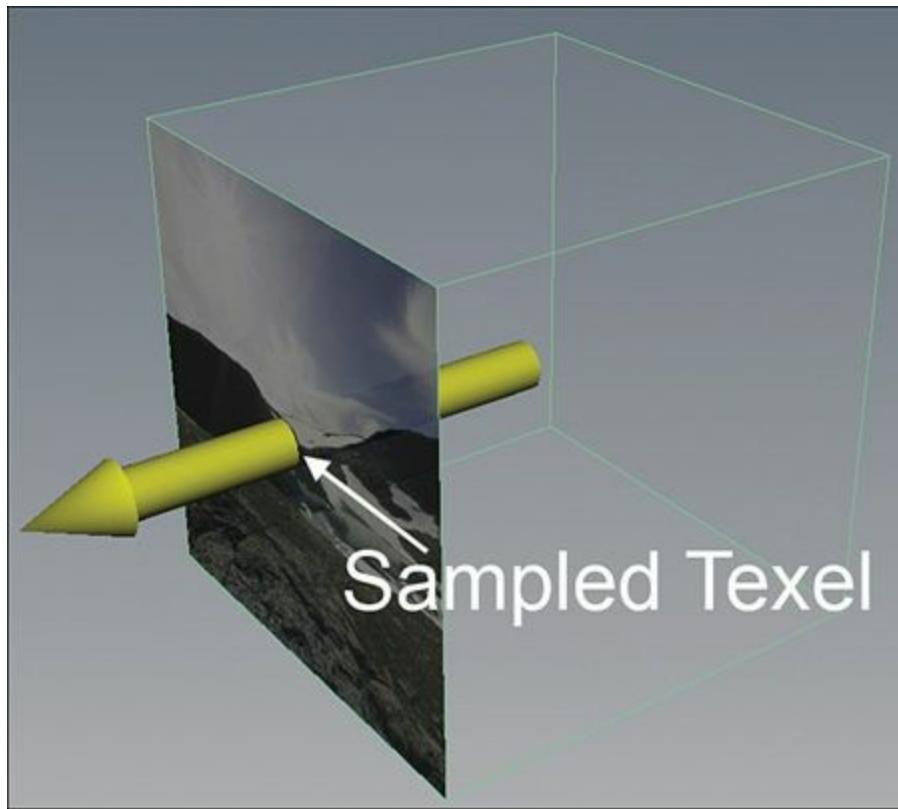
### Warning

I've seen the DirectX Texture Tool add a 1-pixel border color to cube map faces. If your resulting output has visible seams, you might want to inspect the texture cube for such defects. To do so, open the DDS file in an application such as Adobe Photoshop and zoom into the texture.

Aside from constructing a texture cube out of a set of textures, the question of how to create the individual textures themselves also arises. You have a variety of tools to assist with this, including *Terragen* from Planetside Software and *Vue* from e-on software. Websites also offer free or for-fee textures. The book's companion website has links to some of these resources.

## Sampling Texture Cubes

You sample a texture cube with a three-dimensional direction vector rooted at the center of the cube. The chosen texel is where the vector intersects a face of the cube. [Figure 8.4](#) illustrates this concept. The texture filtering settings, which [Chapter 5, “Texture Mapping,”](#) discusses, still apply for coordinates that don't map directly to a texel.



**Figure 8.4** An illustration of texture cube sampling. (*Texture by Emil Persson.*)

## Skyboxes

A common application of a texture cube is a skybox, a box or sphere mapped with a texture cube that surrounds the camera and provides the illusion of an environment. [Listing 8.1](#) presents the code for a skybox effect.

### **Listing 8.1** Skybox.fx

[Click here to view code image](#)

```
***** Resources *****

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
}

TextureCube SkyboxTexture <
    string UIName = "Skybox Texture";
    string ResourceType = "3D";
>

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
```

```

};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 TextureCoordinate : TEXCOORD;
};

/****** Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = IN.ObjectPosition;

    return OUT;
}

/****** Pixel Shader *****/

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return SkyboxTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
}

/****** Techniques *****/

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
}

```

```

        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

---

## Skybox Preamble

Notice that this effect is quite simple, compared to the effects of the last chapter. The `CBufferPerFrame` block has been completely removed, and the `CBufferPerObject` block contains only a `WorldViewProjection` matrix; gone are the members to produce a specular highlight.

The single texture input is now named `SkyboxTexture` and is of type `TextureCube` instead of `Texture2D`. Additionally, the lone `SamplerState` object is now named `Trilinear-Sampler` and removes the explicit settings for addressing mode. The combination of linear interpolation for minification, magnification, and mip-level sampling is dubbed **trilinear sampling** (the name derives from bilinear interpolation, between texels in a mipmap, and an additional interpolation between mip-levels). The addressing mode settings aren't necessary for cube maps.

The `VS_INPUT` struct contains only the vertex position. The surface normal isn't needed without lighting calculations, and texture coordinates are determined within the vertex shader. Similarly, the `VS_OUTPUT` struct just contains the position in homogenous clip space, along with a member for passing the cube map texture coordinates.

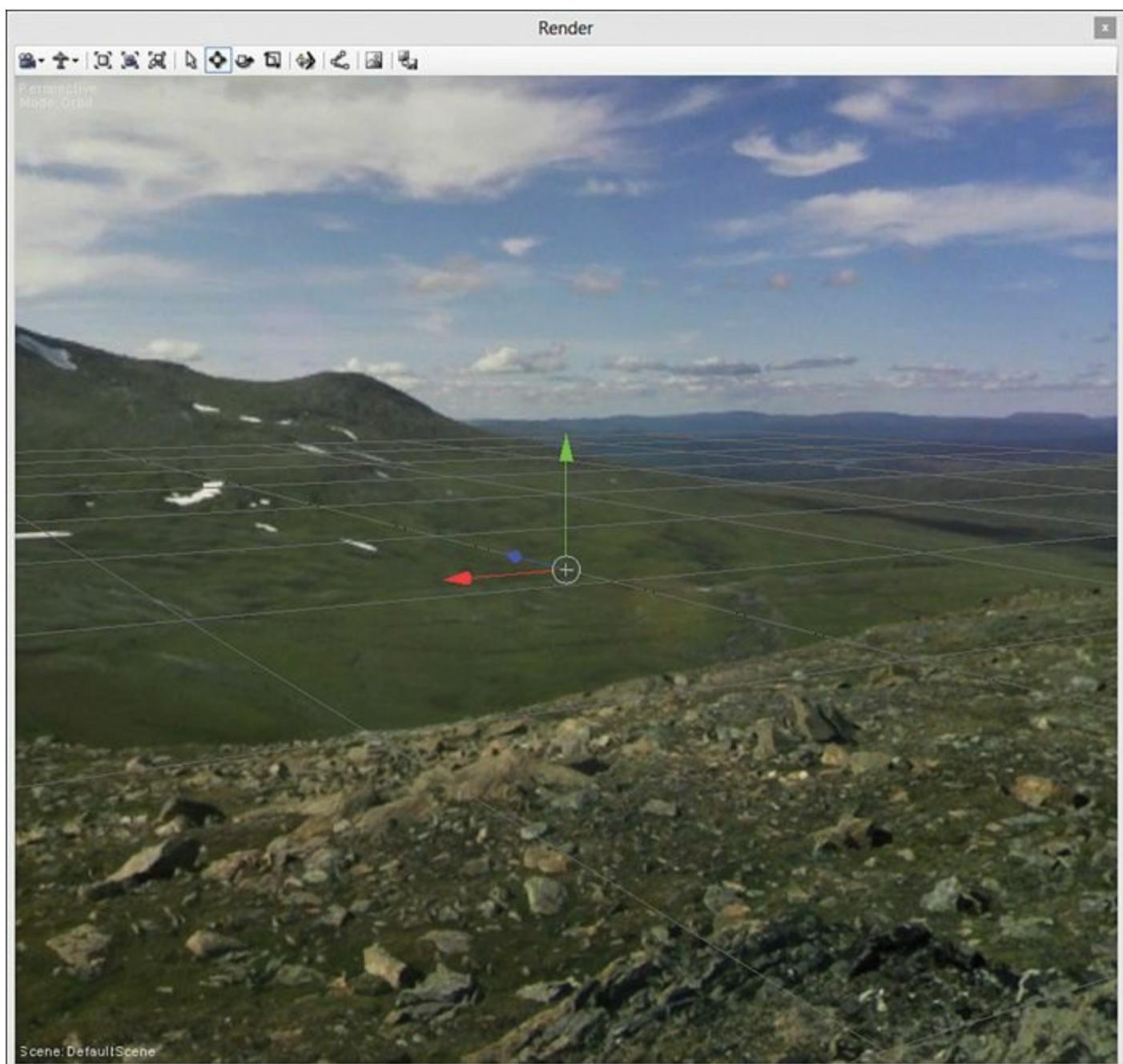
## Skybox Vertex and Pixel Shader

As usual, the vertex shader transforms the vertex position into homogeneous space. Then it passes the object position as texture coordinates. At first glance, this might seem odd, considering that a cube map is sampled through a direction, not a position. But recall from our discussion of vectors in [Chapter 2, “A 3D/Math Primer,”](#) that we can consider a position as a direction vector rooted at the origin.

The pixel shader just samples the texture cube and returns the result as the final output color.

## Skybox Output

[Figure 8.5](#) shows the output of the skybox effect applied to a sphere within NVIDIA FX Composer. In this image, the camera is inside the sphere and a reference grid is visible. Because a skybox surrounds the camera, you must disable backface culling, as [Listing 8.1](#) demonstrates. You also need to scale your sphere to a reasonable size.



**Figure 8.5** Skybox.fx looking outward from inside the associated sphere. (*Texture by Emil Persson.*)

#### Note

If you were using this effect in your own CPU-side application, you would guarantee that the skybox was always positioned in tandem with the camera. That way, the viewer could never reach the edge of the world and shatter the illusion of a surrounding environment.

## Environment Mapping

Another common application of texture cubes is environment mapping. Also known as reflection mapping, environment mapping approximates reflective surfaces, such as the chrome bumper on a car. The process involved is slightly more complicated than for a skybox because you compute a reflection vector for the light interacting with the surface. The reflection vector is dependent on the

view direction (the incident vector) and the surface normal. The specific equation follows:

$$R = I - 2 * N * (I \bullet N)$$

Here,  $I$  is the incident vector and  $N$  is the surface normal. [Listing 8.2](#) presents the code for an environment mapping effect.

## **Listing 8.2** EnvironmentMapping.fx

[Click here to view code image](#)

---

```
#include "include\Common.fhx"

/********************* Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 EnvColor : COLOR <
        string UIName = "Environment Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f };

    float3 CameraPosition : CAMERAPosition < string
UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >

    float ReflectionAmount <
        string UIName = "Reflection Amount";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 1.0;
        float UIStep = 0.05;
    > = {0.5f};
}

Texture2D ColorTexture <
```

```

    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>;

TextureCube EnvironmentMap <
    string UIName = "Environment Map";
    string ResourceType = "3D";
>;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD0;
    float3 ReflectionVector : TEXCOORD1;
};

/****** Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;

```

```

float3 incident = normalize(worldPosition - CameraPosition);
float3 normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);

// Reflection Vector for cube map: R = I - 2 * N * (I.N)
OUT.ReflectionVector = reflect(incident, normal);

return OUT;
}

***** Pixel Shader *****

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 environment = EnvironmentMap.Sample(TrilinearSampler,
IN.ReflectionVector).rgb;
    float3 reflection = get_vector_color_contribution(EnvColor,
environment);

    OUT.rgb = lerp(ambient, reflection, ReflectionAmount);

    return OUT;
}

***** Techniques *****

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

---

## Environment Mapping Preamble

In this effect, the `CBufferPerFrame` block contains members for ambient color and an environment color. This allows for a global ambient value and an independent color/intensity value that's specific to environment-mapped objects. It's just one more dial to give to an artist. Members for directional lighting, specular, point lights, or spotlights have been removed to keep the focus on the specifics of environment mapping. However, all the earlier lighting models could be used in conjunction with environment mapping. `CBufferPerFrame` also contains a `CameraPosition` object for computing the incident (view) vector.

New to the `CBufferPerObject` block is a `ReflectionAmount` member. This value is used for linear interpolation between the ambient term and the computed reflection color.

Also notice the two textures supplied to the environment mapping effect. The `ColorTexture` is the usual 2D texture for sampling the color of the surface. The `EnvironmentMap` shader constant is for the texture cube that supplies the reflected environment.

Finally, observe the `ReflectionVector` member of the `VS_OUTPUT` struct. Computed in the vertex shader, this vector is used to sample the texture cube. The neighboring 2D `TextureCoordinate` member is for sampling the color texture.

## Environment Mapping Vertex Shader

The vertex shader performs the usual steps of transforming the vertex into homogeneous space and passing along the color map's texture coordinates. Then it transforms the vertex into world space, to calculate the incident vector. After transforming the surface normal into world space, the reflection vector is calculated using the HLSL intrinsic `reflect()`. This function performs the same math as presented for the reflection vector, but whenever an intrinsic is available, it's a good idea to use it.

## Environment Mapping Pixel Shader

The pixel shader samples the color texture and computes the ambient term. Then it samples the environment map and modulates that value by the color and intensity of the `EnvColor` uniform. The final output color is produced by interpolating the ambient and reflection terms using the HLSL `lerp()` intrinsic. Linear interpolation uses the formula:

$$Value = x * (1 - s) + (y * s)$$

where,  $s$  is a value between 0.0 and 1.0 that describes how much of the computed value comes from  $x$  and how much comes from  $y$ . Thus, for the environment mapping effect, the final color is computed as:

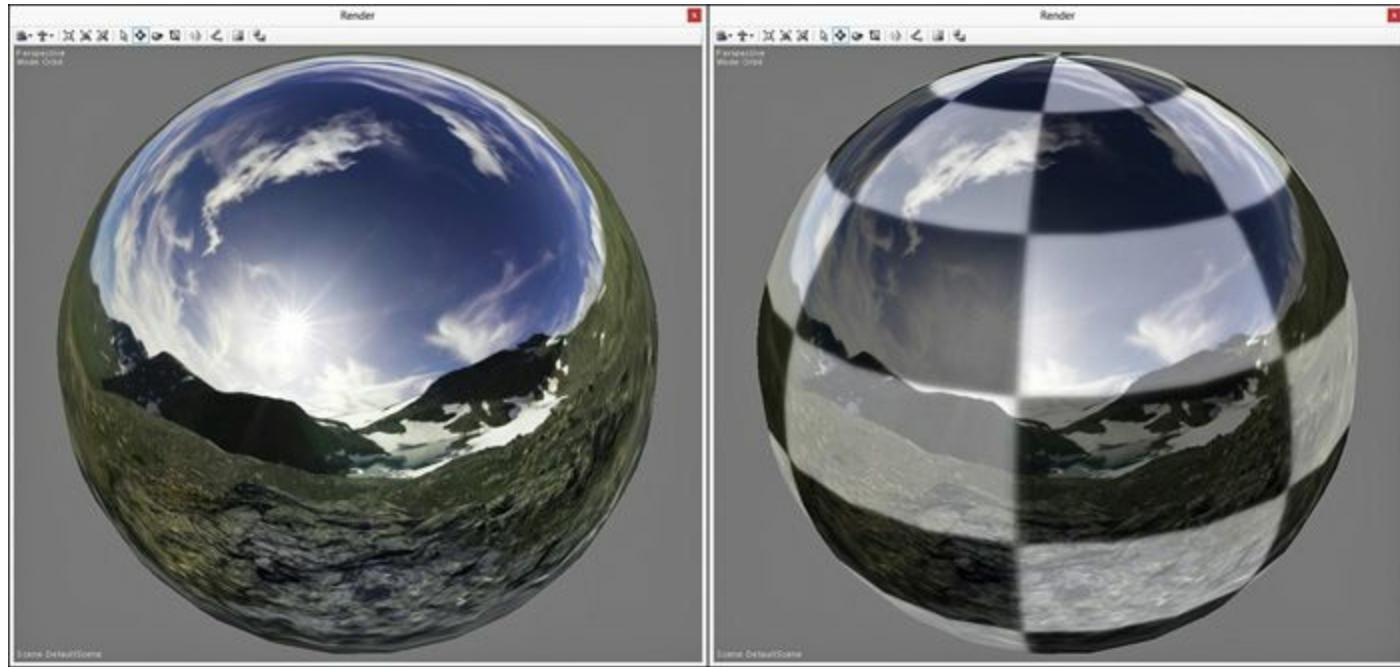
$$Color_{Final} = ambient * (1 - ReflectionAmount) + (reflection * ReflectionAmount)$$

Your `ReflectionAmount` member is therefore just a slider that defines the percentage contribution of the reflection versus the nonreflection colors. If you supply the value 1.0 for `ReflectionAmount`, the color of your object will be mirrorlike, reflecting all of the associated texture cube. Conversely, if you specify 0.0 for the `ReflectionAmount`, your object will reflect none of the environment.

## Environment Mapping Output

[Figure 8.6](#) shows the output of the environment mapping effect applied to a sphere with pure-white,

full-intensity ambient and environment light values. A checkerboard texture is used for the color map, and the environment map is that of [Figure 8.1](#) (without labels). In the left-side image, the reflection amount is assigned the value  $1.0$ ; for the right-side image, the reflection amount is  $0.5$ .



**Figure 8.6** `EnvironmentMapping.fx` applied to a sphere with pure-white, full-intensity ambient and environment light values, and reflection amounts of  $1.0$  (left) and  $0.5$  (right). (*Skybox texture by Emil Persson.*)

## Dynamic Environment Mapping

So far, we've been discussing static environment maps, texture cubes that don't change (or do so infrequently). Static environment maps provide interesting detail and good performance, but if viewers look closely, they may notice that the environment map doesn't exactly match the actual scene. Plenty of games use completely unrelated cube maps that have just reasonable color similarity. If the camera can't get close enough to a reflective surface, or if the shape of the surface distorts the image enough, no one's the wiser. However, if the camera *can* zoom in on a reflective surface, the viewer might see incongruities between the actual environment and the reflected map. For example, nearby objects aren't typically included in a "global" environment map.

One solution is to vary the environment map as the viewer moves from one area to the next, given that the areas are thematically different. Within the same area, you also could vary the environment map based on time of day or weather conditions. But the ultimate solution is to generate dynamic environment maps from within the scene itself. To do so, you position your camera at the location of an object and configure it with a 90-degree field of view (both horizontal and vertical). Then you render the scene six times per frame—looking down each axis—and build a texture cube from the resulting images. In this way, you capture every object in the scene within your reflected environments. However, at least at the time of this writing, this is a prohibitively expensive process to perform at interactive frame rates. Consider creating dynamic texture cubes only for key objects in your scene. Furthermore, consider rendering such cube maps at a lower resolution and perhaps at a fraction of the total frame rate.

## Fog

The next several effects are unrelated to texture cubes but help round out your exposure to various graphics techniques. The first we discuss is fog, fading out an object to a background color as its distance from the camera increases.

Fog can be modeled with a color, a distance from the camera at which the fog begins, and how far the fog extends before the object's color is entirely dominated by the fog color. The process of applying the final color to an object begins by determining how much of the color should come from the fog and how much should come from regular lighting. As with environment mapping, you can use linear interpolation for this process, and you can compute the interpolation value with the following equation:

$$FogAmount = \frac{(|V| - fogStart)}{fogRange}$$

where  $V$  is the view vector between the camera and the surface. Then you interpolate the final pixel color as follows:

$$Color_{Final} = lerp(litColor, fogColor, FogAmount)$$

You can use any lighting model to calculate `litColor` and perform the fog `lerp` as the last step in your pixel shader. [Listing 8.3](#) presents an abbreviated fog effect that highlights the fog-specific code.

### **Listing 8.3** Abbreviated Fog Effect

[Click here to view code image](#)

```
cbuffer CBufferPerFrame
{
    /* ... */

    float3 FogColor <
        string UIName = "Fog Color";
        string UIWidget = "Color";
    > = {0.5f, 0.5f, 0.5f};

    float FogStart = { 20.0f };
    float FogRange = { 40.0f };

    float3 CameraPosition : CAMERAPOSITION < string
UIWidget="None"; >;
}

***** Data Structures *****

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
```

```

    float3 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
    float FogAmount: TEXCOORD3;
};

/****** Utility Functions *****/

float get_fog_amount(float3 viewDirection, float fogStart, float
fogRange)
{
    return saturate((length(viewDirection) - fogStart) /
(fogRange));
}

/****** Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN, uniform bool fogEnabled)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);
    OUT.LightDirection = normalize(-LightDirection);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
    float3 viewDirection = CameraPosition - worldPosition;
    OUT.ViewDirection = normalize(viewDirection);

    if (fogEnabled)
    {
        OUT.FogAmount = get_fog_amount(viewDirection, FogStart,
FogRange);
    }

    return OUT;
}

/****** Pixel Shader *****/

float4 pixel_shader(VS_OUTPUT IN, uniform bool fogEnabled) :
SV_Target
{
    float4 OUT = (float4)0;

```

```

float3 normal = normalize(IN.Normal);
float3 viewDirection = normalize(IN.ViewDirection);
float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
float3 ambient = get_vector_color_contribution(AmbientColor,
color.rgb);

LIGHT_CONTRIBUTION_DATA lightContributionData;
lightContributionData.Color = color;
lightContributionData.Normal = normal;
lightContributionData.ViewDirection = viewDirection;
lightContributionData.LightDirection =
float4(IN.LightDirection, 1);
lightContributionData.SpecularColor = SpecularColor;
lightContributionData.SpecularPower = SpecularPower;
lightContributionData.LightColor = LightColor;
float3 light_contribution =
get_light_contribution(lightContributionData);

OUT.rgb = ambient + light_contribution;
OUT.a = 1.0f;

if (fogEnabled)
{
    OUT.rgb = lerp(OUT.rgb, FogColor, IN.FogAmount);
}

return OUT;
}

***** Techniques *****

technique10 fogEnabled
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0,
vertex_shader(true)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0,
pixel_shader(true)));

        SetRasterizerState(DisableCulling);
    }
}

```

```

technique10 fogDisabled
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0,
vertex_shader(false)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0,
pixel_shader(false)));

        SetRasterizerState(DisableCulling);
    }
}

```

---

## Fog Preamble

The CBufferPerFrame block contains new members for FogColor, FogStart, and FogRange. The VS\_OUTPUT struct contains a float for the FogAmount interpolation value. The scale of the start and range of the fog should be based on the scale selected for your CPU-side application.

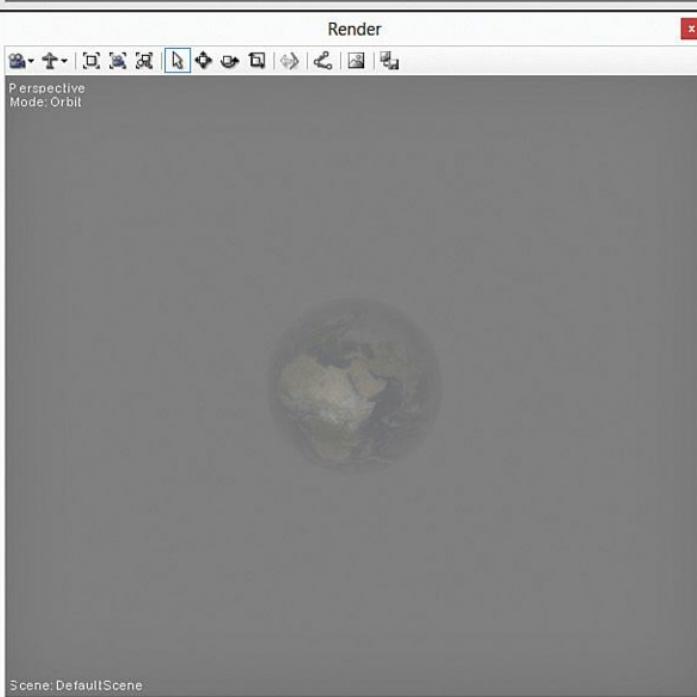
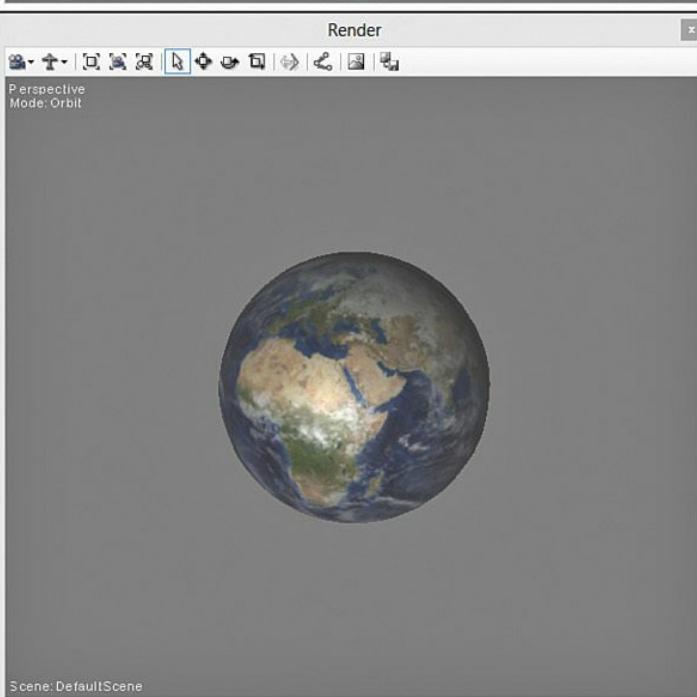
## Fog Vertex and Pixel Shader

The vertex and pixel shaders both include uniform parameters for fogEnabled, which is set by the fogEnabled and fogDisabled techniques. If enabled, the vertex shader calculates the FogAmount with a call to the get\_fog\_amount() utility function. This function should reside in your Common.fxh file for use across your library of effects.

The pixel shader lerps the object's lit color and the fog color to produce the final value for the pixel.

## Fog Output

[Figure 8.7](#) shows the fog effect applied to a sphere with a fog start of 5.0, a fog range of 10.0, and a gray fog color. The sphere is lit with a single directional light and a half-intensity specular highlight. The top image depicts the sphere closer to the camera than the fog start value; therefore, the fog color has no impact. The middle and bottom images show the object progressively farther from the camera, where the fog color begins to take over.



**Figure 8.7** Fog.fx applied to a sphere with a texture of Earth, with a fog start of 5.0, a fog range of 10.0, and a gray fog color. The three images show the object at progressively farther distances from the camera (top to bottom). (*Original texture from Reto Stöckli, NASA Earth Observatory.*

*Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.)*

## Color Blending

With the environment mapping and fog effects, you've been blending colors to produce a final output. And you've performed this blending within the same pixel shader, so the render target never sees the two independent colors; it sees just what's been blended. But a different form of color blending applies when the frame buffer already has a color for a particular pixel, and that pixel gets drawn a second time with the intent of mixing the two colors together. You specify how this happens through DirectX **blend states**.

In HLSL, BlendState objects are described much like the RasterizerState objects you've been using throughout your effects, just with different members. But before you delve into the specifics of creating and using a blend state object, we need to discuss some vocabulary. In color blending, the new color being written is called the **source color**. The **destination color** is the color that already exists in the render target. The source and destination colors are combined through the following formula:

$$(\text{source} * \text{sourceBlendFactor})\text{blendOperation}(\text{dest} * \text{destBlendFactor})$$

Each **blend factor** is one of the values in [Table 8.1](#), and the **blend operation** is one of the options in [Table 8.2](#).

Enumeration	Description
ZERO	Each component of the color is multiplied by (0, 0, 0, 0).
ONE	Each component of the color is multiplied by (1, 1, 1, 1).
SRC_COLOR	Each component of the color is multiplied by the source color.
INV_SRC_COLOR	Each component of the color is multiplied by the inverse of the source color.
SRC_ALPHA	Each component of the color is multiplied by the alpha channel of the source.
INV_SRC_ALPHA	Each component of the color is multiplied by the inverse of the alpha channel of the source.
DEST_ALPHA	Each component of the color is multiplied by the alpha channel of the destination.
INV_DEST_ALPHA	Each component of the color is multiplied by the inverse of the alpha channel of the destination.
DEST_COLOR	Each component of the color is multiplied by the destination color.
INV_DEST_COLOR	Each component of the color is multiplied by the inverse of the destination color.
SRC_ALPHA_SAT	Each component of the color is multiplied by the alpha channel of the source, clamped to a value of 1 or less.
BLEND_FACTOR	Each component of the color is multiplied by a constant set with OMSetBlendState().
INV_BLEND_FACTOR	Each component of the color is multiplied by the inverse of a constant set with OMSetBlendState().

**Table 8.1** Blend Factor Options

Enumeration	Description
ADD	Adds the destination to the source
SUBTRACT	Subtracts the destination from the source
REV_SUBTRACT	Subtracts the source from the destination
MIN	Gives the minimum of the source and the destination
MAX	Gives the maximum of the source and the destination

**Table 8.2** Blend Operation Options

### Note

Additional blend factors support dual-source color blending, but that topic is outside the scope of this book.

scope of this section. For a complete list and description of dual-source color blending, visit the DirectX documentation website.

Using the blend factor and blend operation enumerations, [Table 8.3](#) provides a list of common combinations.

Name	Blend Settings
Additive Blending	(source * ONE) + (dest * ONE)
Multiplicative Blending	(source * ZERO) + (dest * SRC_COLOR)
Alpha Blending	(source * SRC_ALPHA) + (dest * INV_SRC_ALPHA)

**Table 8.3** Common Color Blending Settings

As you can see from [Table 8.3](#), **additive blending** simply yields an addition of the source and destination colors, and **multiplicative blending** just multiplies them. A **2X multiplicative blending** would use `DEST_COLOR` for the source blend factor instead of zeroing it out.

**Alpha blending** creates a transparency effect in which the two colors are mixed based on the source alpha channel. As with the lerps we've discussed for environment mapping and fog, you can consider the source alpha channel as a percentage slider, with the contribution of each color determined by this value. For example, if the source alpha value is `1.0`, then 100 percent of the final color comes from the source; the destination has no impact. If the source alpha value is `0.7`, the result is made up of 70 percent of the source color and 30 percent of the destination.

[Listing 8.4](#) presents the code for an effect with alpha blending enabled. But to make the effect a bit more interesting, it uses a separate texture for the alpha channel, a **transparency map**. This allows for some interesting results by animating or swapping out the alpha texture at runtime. Instead of abbreviating the listing, all code is presented for this effect. It incorporates ambient lighting, a single point light, specular highlighting, and fog.

#### **Listing 8.4** TransparencyMapping.fx

[Click here to view code image](#)

```
#include "include\Common.fhx"

/***************** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
```

```

        string UIName = "Light Color";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float3 LightPosition : POSITION <
    string Object = "PointLight0";
    string UIName = "Light Position";
    string Space = "World";
> = {0.0f, 0.0f, 0.0f};

float LightRadius <
    string UIName = "Light Radius";
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 100.0;
    float UIStep = 1.0;
> = {10.0f};

float3 FogColor <
    string UIName = "Fog Color";
    string UIWidget = "Color";
> = {0.5f, 0.5f, 0.5f};

float FogStart = { 20.0f };
float FogRange = { 40.0f };

float3 CameraPosition : CAMERAPOSITION < string
UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
> = {1.0f};
}

```

```
> = {25.0f};  
}  
  
Texture2D ColorTexture <  
    string ResourceName = "default_color.dds";  
    string UIName = "Color Texture";  
    string ResourceType = "2D";  
>;  
  
Texture2D TransparencyMap <  
    string UIName = "Transparency Map";  
    string ResourceType = "2D";  
>;  
  
SamplerState TrilinearSampler  
{  
    Filter = MIN_MAG_MIP_LINEAR;  
    AddressU = WRAP;  
    AddressV = WRAP;  
};  
  
RasterizerState DisableCulling  
{  
    CullMode = NONE;  
};  
  
BlendState EnableAlphaBlending  
{  
    BlendEnable[0] = True;  
    SrcBlend = SRC_ALPHA;  
    DestBlend = INV_SRC_ALPHA;  
};  
  
/****** Data Structures *****/  
  
struct VS_INPUT  
{  
    float4 ObjectPosition : POSITION;  
    float2 TextureCoordinate : TEXCOORD;  
    float4 Normal : NORMAL;  
};  
  
struct VS_OUTPUT  
{  
    float4 Position : SV_Position;  
    float3 Normal : NORMAL;  
    float2 TextureCoordinate : TEXCOORD0;  
}
```

```

    float4 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
    float FogAmount : TEXCOORD3;
};

#008000">***** Vertex Shader *****

VS_OUTPUT vertex_shader(VS_INPUT IN, uniform bool fogEnabled)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;

    OUT.LightDirection = get_light_data(LightPosition,
worldPosition,
LightRadius);
    float3 viewDirection = CameraPosition - worldPosition;

    if (fogEnabled)
    {
        OUT.FogAmount = get_fog_amount(viewDirection, FogStart,
FogRange);
    }

    OUT.ViewDirection = normalize(viewDirection);

    return OUT;
}

#008000">***** Pixel Shader *****

float4 pixel_shader(VS_OUTPUT IN, uniform bool fogEnabled) :
SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 viewDirection = normalize(IN.ViewDirection);
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor,

```

```

color.
rgb);

LIGHT_CONTRIBUTION_DATA lightContributionData;
lightContributionData.Color = color;
lightContributionData.Normal = normal;
lightContributionData.ViewDirection = viewDirection;
lightContributionData.SpecularColor = SpecularColor;
lightContributionData.SpecularPower = SpecularPower;
lightContributionData.LightDirection = IN.LightDirection;
lightContributionData.LightColor = LightColor;
float3 light_contribution =
get_light_contribution(lightContributionData);

OUT.rgb = ambient + light_contribution;
OUT.a = TransparencyMap.Sample(TrilinearSampler,
IN.TextureCoordinate).a;

if (fogEnabled)
{
    OUT.rgb = lerp(OUT.rgb, FogColor, IN.FogAmount);
}

return OUT;
}

technique10 alphaBlendingWithFog
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0,
vertex_shader(true)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0,
pixel_shader(true)));

        SetRasterizerState(DisableCulling);
        SetBlendState(EnableAlphaBlending, (float4)0,
0xFFFFFFFF);
    }
}

technique10 alphaBlendingWithoutFog
{
    pass p0
    {
}

```

```

        SetVertexShader(CompileShader(vs_4_0,
vertex_shader(false)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0,
pixel_shader(false)));

        SetRasterizerState(DisableCulling);
        SetBlendState(EnableAlphaBlending, (float4)0,
0xFFFFFFFF);
    }
}

```

---

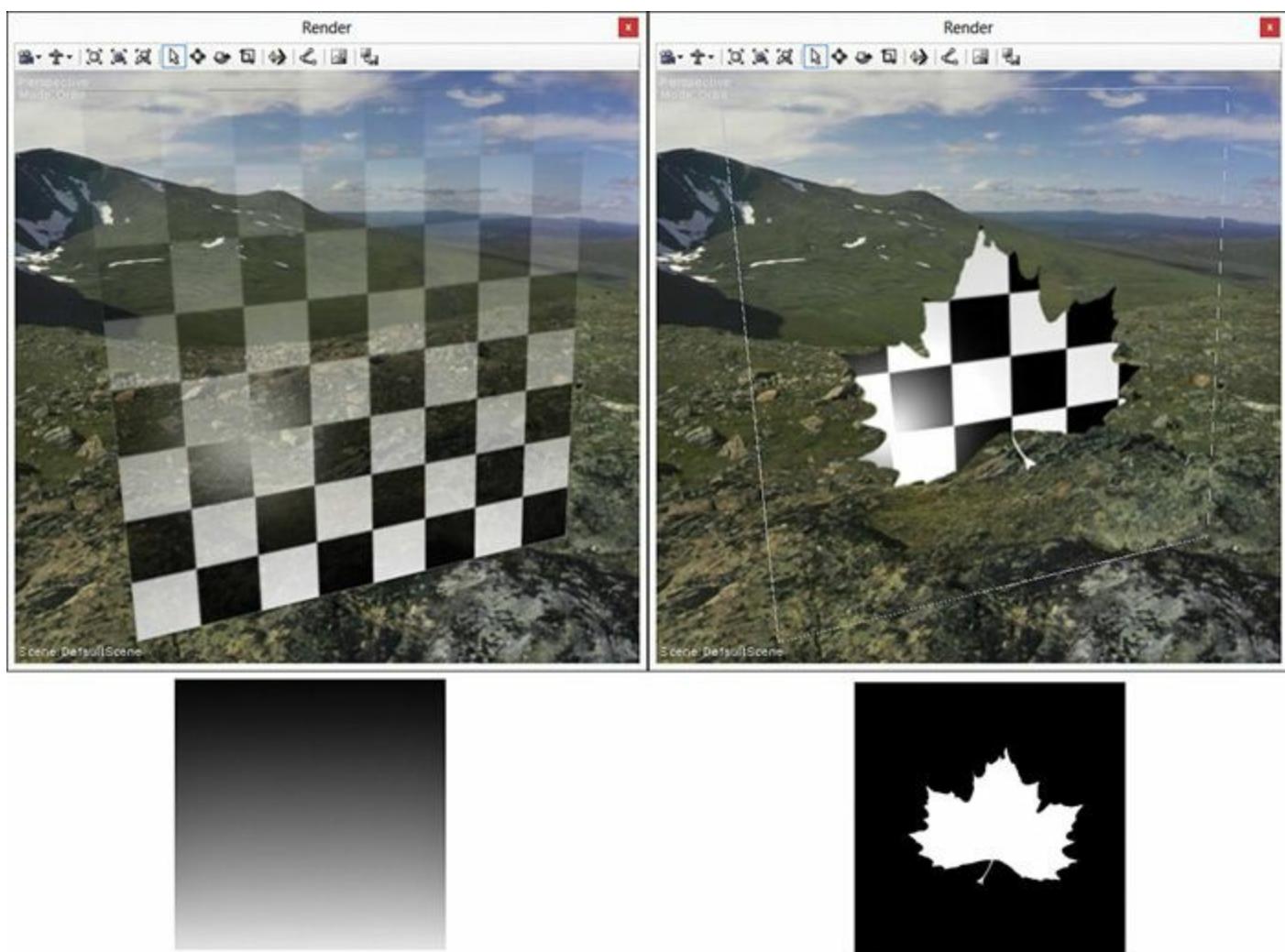
## Transparency Mapping Effect

Much of this code you've seen before, although separated out into individual effects. New is the `EnableAlphaBlending` blend state object. You set the `SrcBlend` and `DestBlend` members for alpha blending and then enable color blending for the first render target bound to the output-merger stage with `BlendEnable[0] = True`. You can bind eight render targets to the output-merger stage at one time. [Part IV, “Intermediate-Level Rendering Topics,”](#) discusses multiple render targets further.

You apply the blend state from within a technique through a call to `SetBlendState()`. The first parameter is your blend state object. The second parameter is the constant color used when either the source or the destination blend factor is set to the `BLEND_FACTOR` enumeration (see [Table 8.1](#)). The third parameter is a 32-bit multisample coverage mask that determines which samples are updated for the active render targets. [Chapter 11, “Direct3D Initialization,”](#) covers multisampling.

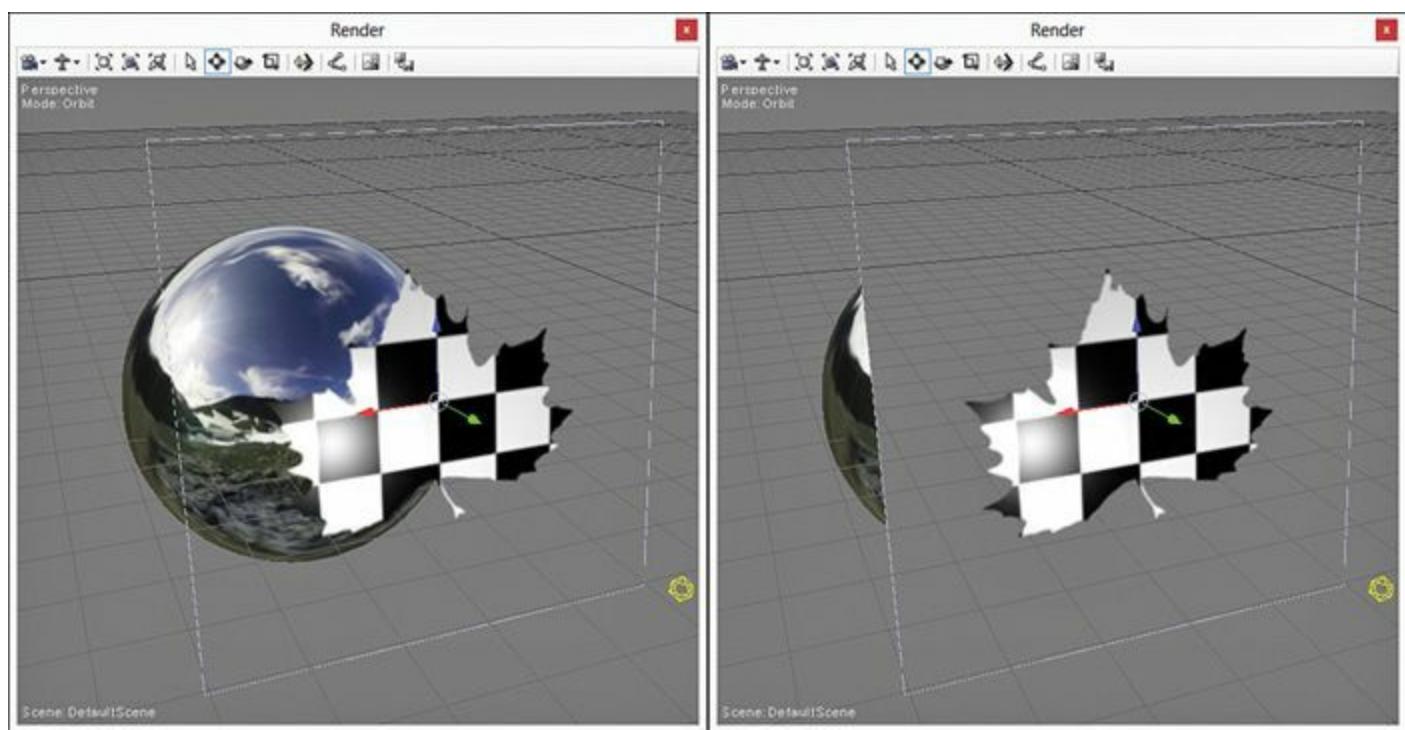
## Transparency Mapping Output

[Figure 8.8](#) shows the output of the transparency mapping effect applied to a plane with a checkerboard color texture surrounded by a skybox. Beneath each image is the texture used for the alpha channel. In the image to the left, the alpha map is a gradient transitioning from transparent to opaque (from 0.0 to 1.0). In grayscale, these values are visualized as a gradient from black (0.0) to white (1.0). Note that the single-pixel border around the right-side image is apparent because the plane is selected in the NVIDIA FX Composer Render panel. This is to demonstrate the transparency effect and is not a rendering artifact.



**Figure 8.8** TransparencyMapping.fx applied to a plane with a checkerboard color texture using a gradient alpha map (left) and an alpha map in the shape of a maple leaf (right). (*Skybox texture by Emil Persson.*)

Alpha blending is dependent on the order in which objects are rendered. Transparent objects should be rendered from back to front (farthest from the camera to nearest); otherwise, the “bleed through” of the background pixels will be incorrect. [Figure 8.9](#) shows two iterations of the same scene, in which a plane is in front of a sphere. In both images, the render target is cleared to a gray color and then a reference grid is drawn. Afterward, in the image to the left, the environment-mapped sphere is drawn and then the alpha-blended plane is drawn. For the image to the right, the plane is drawn before the sphere. Notice that the image to the right is mixed with the gray background color overlaid with the grid instead of the sphere because the sphere hasn’t yet been drawn to the render target when the plane is blended.



**Figure 8.9** A scene depicting the impact of draw order on alpha-blended objects. On the left, the sphere is rendered before the plane, and vice versa for the image to the right. (*Skybox texture by Emil Persson.*)

### Note

The draw order for NVIDIA FX Composer is determined by the sequence in which the scene elements are created. The first object created in the scene is drawn first.

For opaque images, the color isn't affected by the draw order, but rendering opaque objects from front to back has performance gains. We discuss this further in [Part IV](#).

## Summary

In this chapter, you discovered several graphics techniques. You learned about texture cubes and their application for skyboxes and environment mapping. You wrote shaders for producing a fog effect and explored color blending. Finally, you wrote an effect for transparency mapping and learned a bit about draw order.

In the next chapter, you explore effects for normal mapping and displacement mapping, the final topics in [Part II](#), “[Shader Authoring with HLSL](#).”

## Exercises

1. Create a texture map, either by hand or using a tool such as *Terragen* from Planetside Software. Use the texture map with the skybox effect, and observe the output as you manipulate the camera.
2. Experiment with the environment mapping effect. Modify the ambient and environment light values and the reflection amount, and observe the results.
3. Implement a complete fog effect that incorporates a single directional light.

4. Explore the transparency mapping effect with multiple objects in a scene. Vary the creation order of objects, and observe the results.

# Chapter 9. Normal Mapping and Displacement Mapping

In this chapter, you discover techniques for adding detail to your scenes without necessarily increasing the poly count of your objects. With normal mapping, you create “fake” geometry for simulating light interaction. And with displacement mapping, you create bumps and recesses by actually moving vertices according to texture data.

## Normal Mapping

In previous chapters, you learned about specular maps, environment maps, and transparency maps—textures that provide additional information to your shaders. Specular maps limit the specular highlight, environment maps contribute colors for reflective surfaces, and transparency maps control alpha blending in the output-merger stage. This information is provided per pixel, a much higher resolution than if the data were supplied per vertex. Similarly, a normal map provides surface normals per pixel, and this extra data has many applications.

One application of normal maps is to fake the detail of a bumpy surface, such as a stone wall. You could model such a wall with geometry for actual bumps and recesses—and the more vertices you used, the greater detail you would achieve. This geometry would respond well to lights in the scene, showing dark areas where the stone faced away from the light. However, added geometry comes at a cost. Instead, it’s possible to create a low-poly object (even a flat plane) and use a normal map to simulate the lighting effects that would be present if the added geometry were there. This application is called **normal mapping**.

## Normal Maps

[Figure 9.1](#) shows a color map (left) and normal map (right) for a stone wall. Note the relatively odd look of the normal map. Although you visualize a normal map in color, what’s stored are 3D directions vectors. When you sample the normal map, the results you find in the RGB channels are the x, y, and z components of the direction. You can use these normals in calculating, for example, the diffuse lighting term.



**Figure 9.1** A color map (left) and normal map (right) for a stone wall. (*Textures by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

The channels of an RGB texture store unsigned 8-bit values and, therefore, have a range of [0,

255]. But the  $xyz$  components of a normalized direction vector have a range of  $[-1, 1]$ . Consequently, the normals must be shifted for storage in the texture and shifted back when the vector is sampled. You transform the floating-point vector from  $[-1, 1]$  to  $[0, 255]$  using the following function:

$$f(x) = (0.5x + 0.5) * 255$$

And you transform it back with the function:

$$f(x) = \frac{2x}{255} - 1$$

In practice, you use an image-manipulation tool such as Adobe Photoshop to encode a normal map in RGB texture format. But you're responsible for shifting the value back into the range  $[-1, 1]$  when you sample the texture from within your shader. The floating-point division (by 255) is done for you during the sampling process so that the sampled values will exist in the range  $[0, 1]$ . Therefore, you only need to shift the sampled vector with this function:

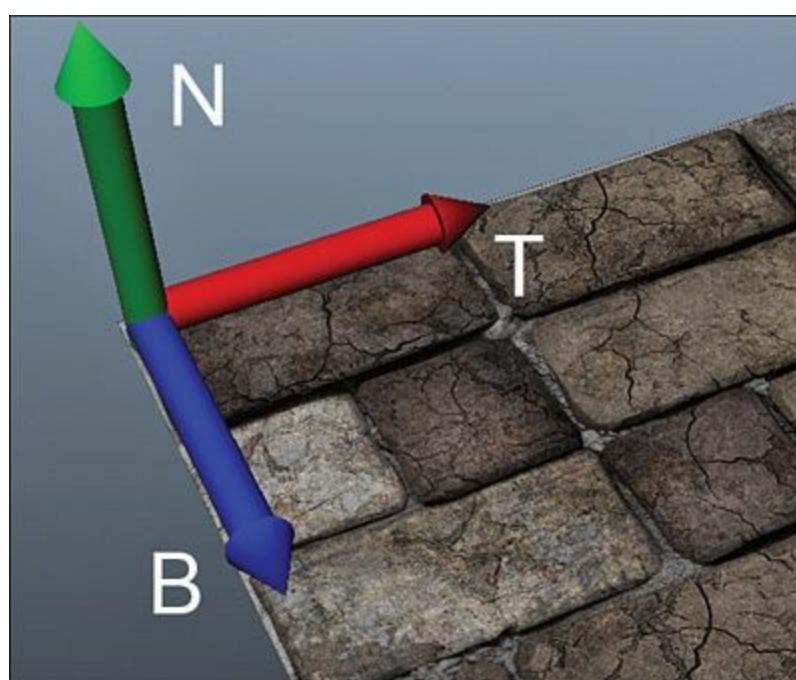
$$f(x) = 2x - 1$$

Alternately, you can use 16- or 32-bit floating-point texture formats for your normal maps, which yields better results at the expense of performance.

## Tangent Space

You can use a per-pixel normal to compute the diffuse term, just as you would with a per-vertex normal. However, the normal must be in the same coordinate space as your light. For per-vertex normals, the data is provided in object space. But for normal maps, the data is provided in **tangent space**.

Tangent space (or **texture space**) is a coordinate system with respect to the texture and is described by three orthogonal vectors: the surface normal, tangent, and binormal. [Figure 9.2](#) illustrates these vectors.



**Figure 9.2** An illustration of tangent space. (*Texture by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

The normal vector,  $N$ , is the regular surface normal of a vertex. The tangent vector,  $T$ , is orthogonal to the surface normal and is in the direction of the  $u$ -axis of the texture. The binormal,  $B$ , runs along the texture's  $v$ -axis.

You can use these three vectors to construct a *tangent, binormal, normal* (TBN) transformation matrix, as follows:

$$TBN = \begin{bmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \\ N.x & N.y & N.z \end{bmatrix}$$

You can use this matrix to transform vectors from tangent space to object space. However, because the light vector is (commonly) in world space, you need to transform the sampled normal from tangent space to world space. You can do this by transforming the normal from tangent space to object space and then to world space. Alternatively, you can build the TBN matrix from vectors that are already in world space.

### Note

You can encode normals directly in world space, which removes the need for transformation. However, then the objects that used those normals would need to remain static (no transformation). Furthermore, world space-encoded normals couldn't be easily reused between objects (again, because they can't be transformed).

An interesting property of the TBN matrix is that it is made from orthonormal vectors (orthogonal unit vectors) and therefore forms a **basis set** (it defines a coordinate system). Moreover, the inverse of this matrix is its transpose. Thus, to transform a vector from object or world space back into tangent space (**inverse mapping**), you can just multiply the vector by the transpose of the TBN matrix. Additionally, the orthonormal property of the TBN matrix implies that, if you have two of the vectors, you can derive the third. The normal and tangent vectors are commonly stored alongside the geometry, with the binormal computed (within the vertex shader) as a cross product between these two vectors. Performing such computation is often a desirable tradeoff between added GPU computation and the high cost of data transfer between the CPU and GPU.

## A Normal Mapping Effect

With that bit of terminology in hand, [Listing 9.1](#) presents the code for a normal mapping effect.

### **Listing 9.1** NormalMapping.fx

[Click here to view code image](#)

```
#include "include\\Common.fxh"

cbuffer CBufferPerFrame
{
```

```

float4 AmbientColor : AMBIENT <
    string UIName = "Ambient Light";
    string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float4 LightColor : COLOR <
    string Object = "LightColor0";
    string UIName = "Light Color";
    string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float3 LightDirection : DIRECTION <
    string Object = "DirectionalLight0";
    string UIName = "Light Direction";
    string Space = "World";
> = {0.0f, 0.0f, -1.0f};

float3 CameraPosition : CAMERAPOSITION < string
UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
> = {25.0f};

    Texture2D ColorTexture <
        string ResourceName = "default_color.dds";
        string UIName = "Color Texture";
        string ResourceType = "2D";
>;
}

```

```
Texture2D NormalMap <
    string ResourceName = "default_bump_normal.dds";
    string UIName = "Normap Map";
    string ResourceType = "2D";
>

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
    float3 Tangent : TANGENT;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float3 Tangent : TANGENT;
    float3 Binormal : BINORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
};

/****** Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0),
```

```

World).xyz);
    OUT.Tangent = normalize(mul(float4(IN.Tangent, 0),
World).xyz);
    OUT.Binormal = cross(OUT.Normal, OUT.Tangent);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.LightDirection = normalize(-LightDirection);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
    float3 viewDirection = CameraPosition - worldPosition;
    OUT.ViewDirection = normalize(viewDirection);

    return OUT;
}

```

**\*\*\*\*\* Pixel Shader \*\*\*\*\***

```

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 sampledNormal = (2 *
NormalMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xyz) - 1.0; // Map normal from [0..1] to
[-1..1]
    float3x3 tbn = float3x3(IN.Tangent, IN.Binormal, IN.Normal);

    sampledNormal = mul(sampledNormal, tbn); // Transform normal
to
world space

    float3 viewDirection = normalize(IN.ViewDirection);
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);

    LIGHT_CONTRIBUTION_DATA lightContributionData;
    lightContributionData.Color = color;
    lightContributionData.Normal = sampledNormal;
    lightContributionData.ViewDirection = viewDirection;
    lightContributionData.LightDirection =
float4(IN.LightDirection, 1);
    lightContributionData.SpecularColor = SpecularColor;
    lightContributionData.SpecularPower = SpecularPower;
    lightContributionData.LightColor = LightColor;

```

```

    float3 light_contribution = get_light_contribution
(lightContributionData);

    OUT.rgb = ambient + light_contribution;
    OUT.a = 1.0f;

    return OUT;
}

/****** Techniques *****/

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

## Normal Mapping Preamble

The preamble for this effect employs an ambient light, specular highlight, and a single directional light. A new `Texture2D` object exists for the normal map, and the `VS_INPUT` struct accepts a tangent (in object space) along with the surface normal. The `VS_OUTPUT` struct has new members for the tangent and binormal, which will be transformed to world space before they are passed.

## Normal Mapping Vertex and Pixel Shader

The vertex shader transforms the normal and tangent into world space and then performs a cross product between the vectors to produce the binormal. The pixel shader samples the normal map and shifts the sampled normal from  $[0, 1]$  to  $[-1, 1]$ . Then it builds the TBN matrix and uses it to transform the sampled normal into world space. When the normal is in world space, the remaining lighting calculations are the same as with previous effects.

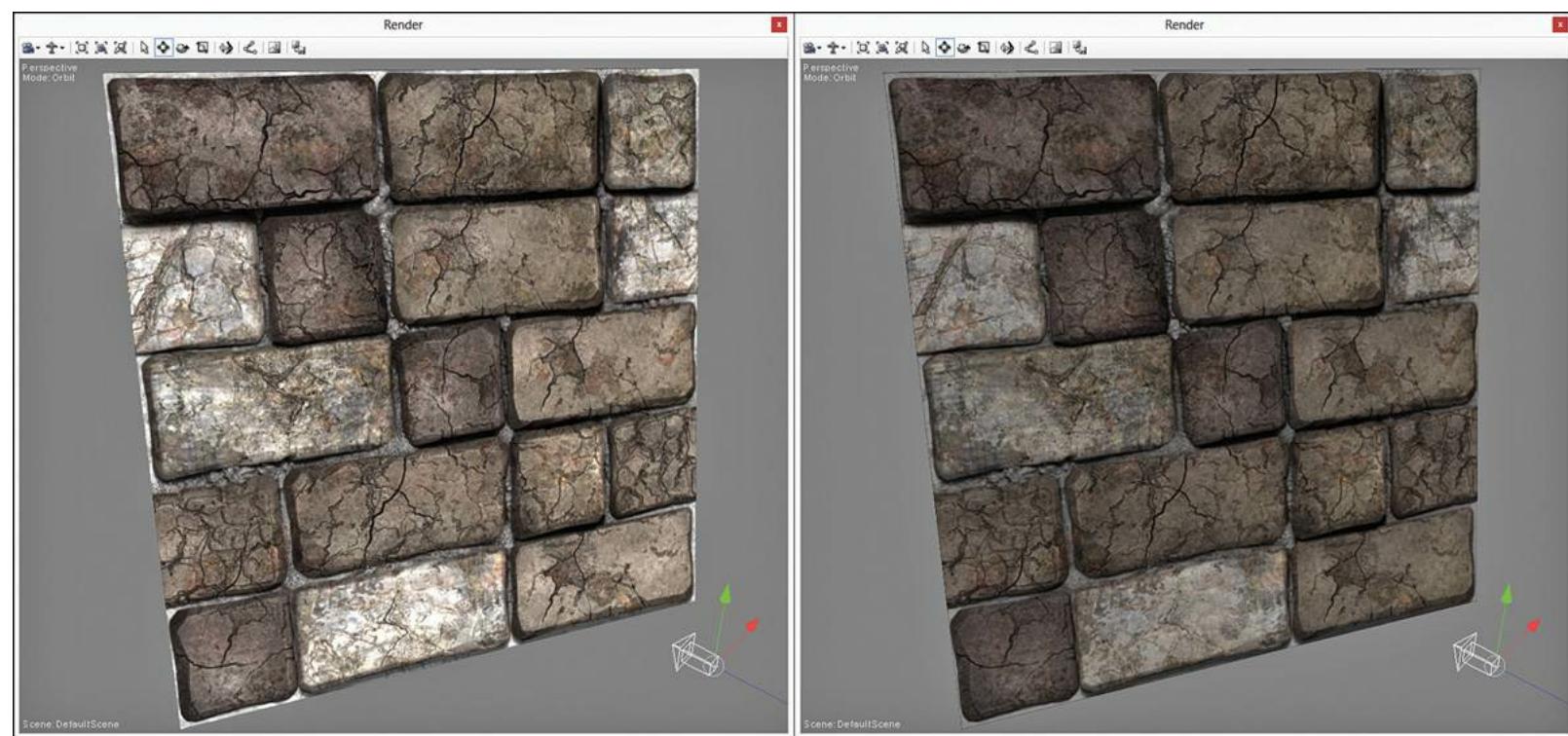
### Note

If you find that your vertex shader is a performance bottleneck, you can provide the binormal as input (alongside the surface normal and tangent). It's a tradeoff between the performance of the vertex shader and the data passed over the graphics bus. More commonly, the graphics bus is the bottleneck.

## Normal Mapping Output

[Figure 9.3](#) shows the results of the normal mapping effect applied to a plane with a stone wall texture.

The normal map from [Figure 9.1](#) is used for the left-side image. For the image to the right, a normal map is applied that contains no perturbations. For both images, the ambient light is disabled, the directional light is pure white at full intensity, and the specular highlight has a power of 100 with an intensity of 0.35.



**Figure 9.3** `NormalMapping.fx` applied to a plane with a stone wall texture using a normal map (left) and without a normal map (right). (*Textures by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

## Displacement Mapping

Another application of normal maps is **displacement mapping**, in which detail is added not by faking lighting, but by actually perturbing the vertices of a model. With displacement mapping, a vertex is translated along its normal by an amount specified in a **displacement map**. A displacement map is just a **height map**, a single-channel texture that stores magnitudes. [Figure 9.4](#) shows a displacement map for the stone wall texture used throughout this chapter. Because the displacement map has just a single 8-bit channel, it's visualized in grayscale.



**Figure 9.4** A color map (left) and displacement map (right) for a stone wall. (*Textures by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

When displacing a vertex, you do so either *inward* or *outward* along its normal, with the magnitude sampled from the displacement map. For outward displacement, you use the following equation:

$$\text{Position} = \text{Position}_0 + (\text{Normal} * \text{Scale} * \text{DisplacementMagnitude})$$

Here, *Scale* is a shader constant that scales the magnitudes stored within the displacement map. For inward displacement, the equation is rewritten as:

$$\text{Position} = \text{Position}_0 + (\text{Normal} * \text{Scale} * \text{DisplacementMagnitude} - 1)$$

## A Displacement Mapping Effect

[Listing 9.2](#) presents the code for an abbreviated displacement mapping effect. This listing displays only the code specific to displacement mapping within the vertex shader.

### **Listing 9.2** An Abbreviated Displacement Mapping Effect

[Click here to view code image](#)

---

```
cbuffer CBufferPerObject
{
    /* ... */

    float DisplacementScale <
        string UIName = "Displacement Scale";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 2.0;
        float UIStep = 0.01;
    > = {0.0f};

    Texture2D DisplacementMap <
        string UIName = "Displacement Map";
        string ResourceType = "2D";
    >

    VS_OUTPUT vertex_shader(VS_INPUT IN)
    {
        VS_OUTPUT OUT = (VS_OUTPUT)0;

        float2 textureCoordinate =
get_corrected_texture_coordinate(IN.
TextureCoordinate);
```

```

if (DisplacementScale > 0.0f)
{
    float displacement = DisplacementMap.
SampleLevel(TrilinearSampler, textureCoordinate, 0);
    IN.ObjectPosition.xyz += IN.Normal * DisplacementScale *
(displacement - 1);
}

OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
OUT.TextureCoordinate = textureCoordinate;
OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);

float3 worldPosition = normalize(mul(IN.ObjectPosition,
World)).xyz;
    OUT.ViewDirection = normalize(CameraPosition -
worldPosition);

    OUT.LightDirection = get_light_data(LightPosition,
worldPosition,
LightRadius);

return OUT;
}

```

---

## Displacement Mapping Preamble

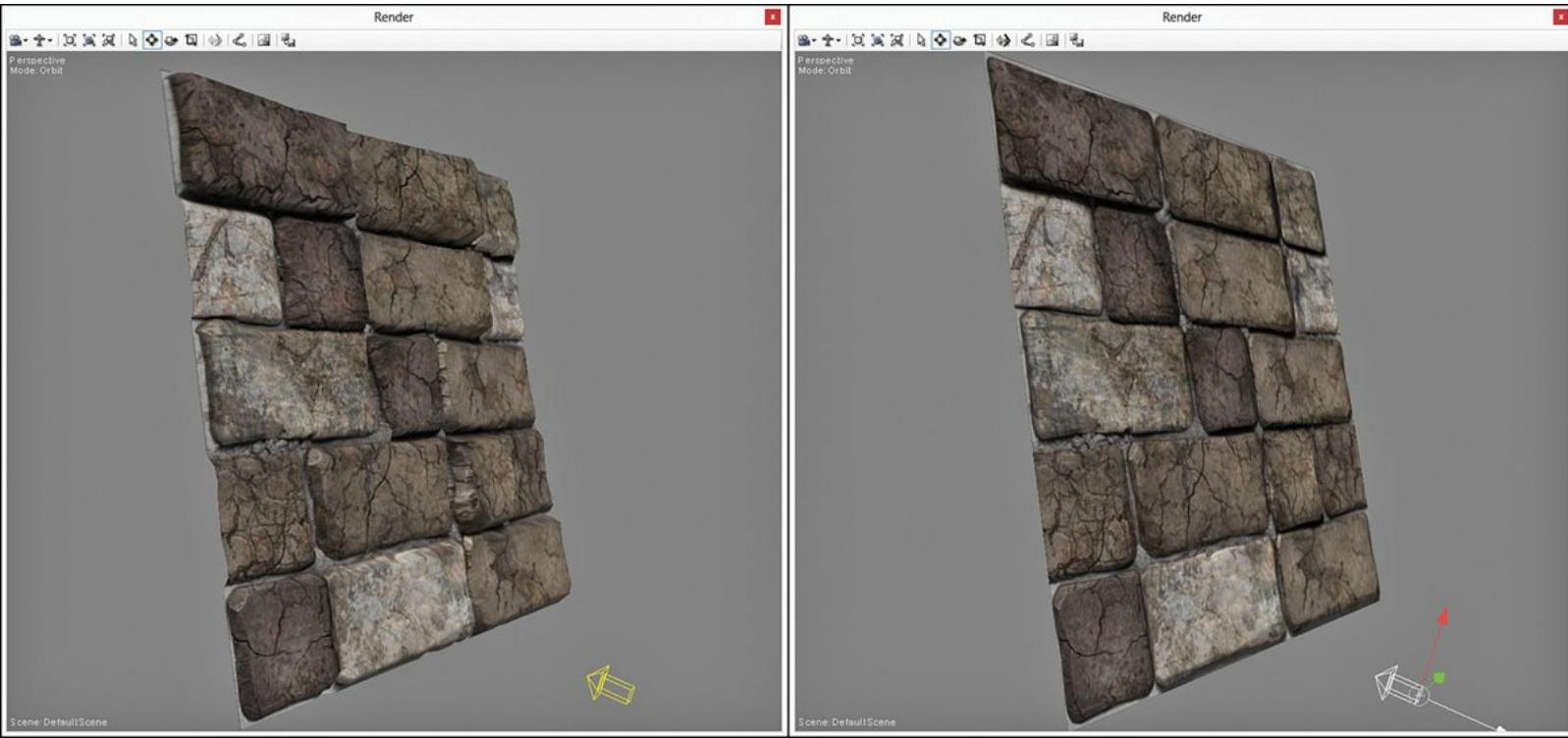
The CBufferPerFrame block contains a new member for displacement scale. A new Texture2D object also exists for the displacement map. You can combine these new members with any of the lighting effects you've developed thus far.

## Displacement Mapping Vertex Shader

The displacement mapping-specific code samples the displacement magnitude and then displaces the vertex inward along its normal. When sampling a texture from within the vertex shader, you use the `SampleLevel()` method, where the second parameter is the mip-level. Note that the normal mapping code has been removed from [Listing 9.2](#), to point out that the normal is not coming from a separate texture. A better iteration of this effect is to employ tessellation to add vertices and sample the normal from a normal map. [Part IV “Intermediate-Level Rendering Topics,”](#) explores this topic further.

## Displacement Mapping Output

[Figure 9.5](#) shows the results of the displacement mapping effect applied to a plane with the stone wall texture. In the image to the left, the displacement map in [Figure 9.4](#) is applied with a scale of 0.2. For the image to the right, the displacement is nullified with a scale of 0.0.



**Figure 9.5** DisplacementMapping.fx applied to a plane with a stone wall texture using a displacement map (left) and without a displacement map (right). (*Textures by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

### Note

The plane used in [Figure 9.5](#) was created using Autodesk Maya, with 50 horizontal and vertical subdivisions. It has a higher vertex count than the built-in plane model NVIDIA FX Composer includes.

## Summary

In this chapter, you learned about normal mapping and displacement mapping, techniques for adding detail to an object without necessarily increasing geometry. This chapter marks the end of [Part II, “Shader Authoring with HLSL.”](#) In the next section, you begin working with the DirectX API using C++ and incorporate the shaders you’ve authored into a C++ rendering engine.

## Exercises

1. Find or create a color texture with an associated normal map (resource links are available on the book’s companion website). Use these textures to experiment with the normal mapping effect. Modify the lighting, manipulate the camera, and observe the output.
2. Implement a complete displacement mapping effect that incorporates a single point light.
3. Find or create a color texture with an associated displacement map. Use these textures to experiment with the displacement mapping effect. Modify the displacement scale, and observe the results.

# Part III: Rendering with DirectX

[10 Project Setup and Window Initialization](#)

[11 Direct3D Initialization](#)

[12 Supporting Systems](#)

[13 Cameras](#)

[14 Hello, Rendering!](#)

[15 Models](#)

[16 Materials](#)

[17 Lights](#)

# Chapter 10. Project Setup and Window Initialization

In this chapter, you establish the foundation of your rendering engine. You create the necessary Visual Studio projects, implement a game loop, and present a window to the screen.

## A New Beginning

Now that you've reached the C++ chapters (the sections of the book that focus on the DirectX API using C++), it's important to set your expectations about how quickly you can get to rendering something to the screen. Put simply, it's going take awhile. You need to build quite a bit of scaffolding, but the net result will pay dividends in your capability to reuse the code over multiple projects. If you'd prefer to use the existing codebase from the book's companion website and skip ahead a couple chapters to where you are actually rendering, you are welcome to do so. But if you're interested in how the supporting systems are developed, you'll find the next few chapters particularly interesting.

Also, although these chapters expect that you are already familiar with C++ programming and Visual Studio, they do not expect that you are familiar with writing Windows applications. Thus, this initial material walks you through setting up the projects that you'll be able to use for all the work going forward.

## Project Setup

The rendering engine that's developed over the next several chapters is split into two Visual Studio projects, a Library project and a Game project. The Library project contains common code that you can use across any number of games or rendering applications. The Game project houses code that is application specific.

## Directory Structure

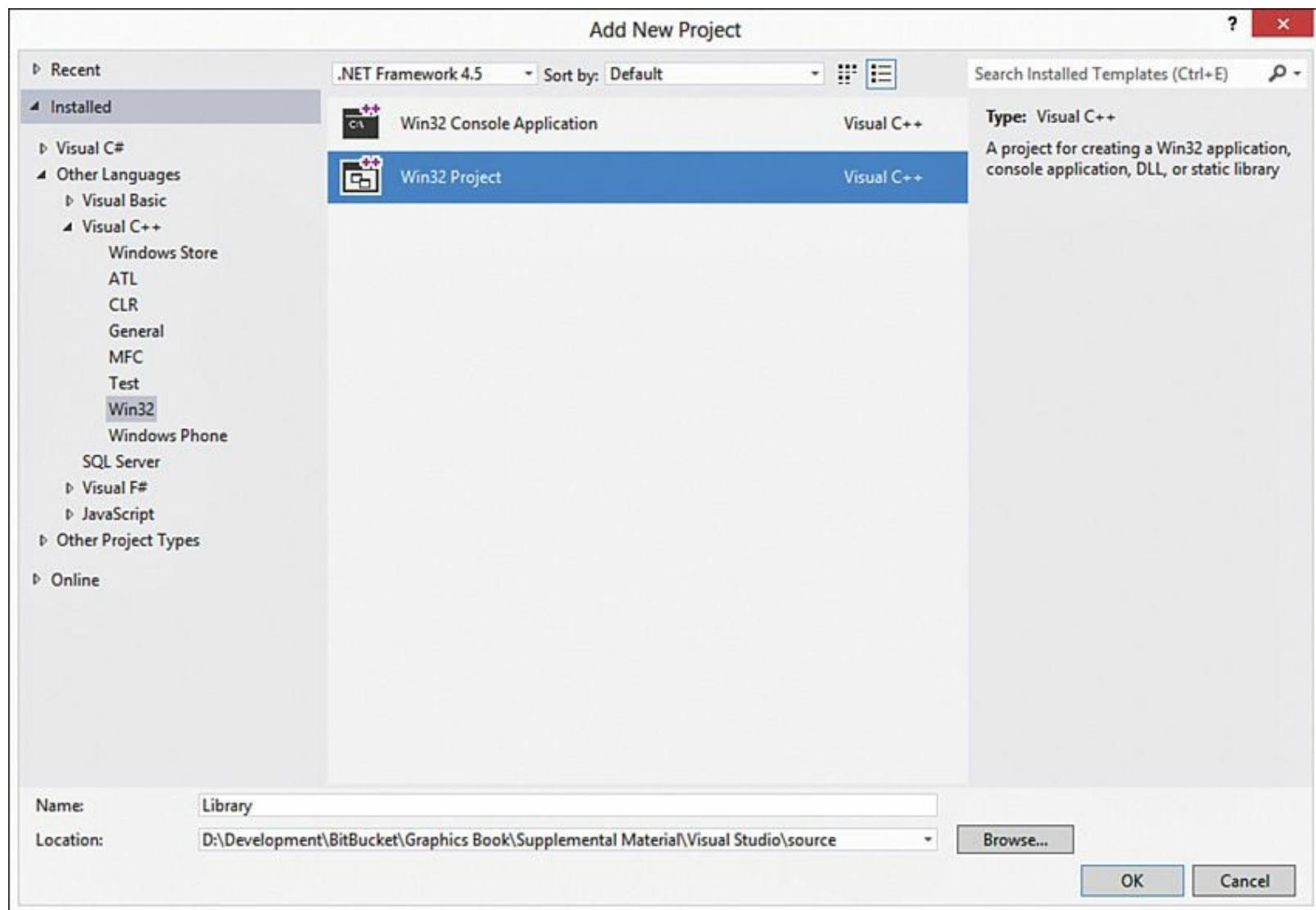
Establishing a directory structure for your projects is a good idea. [Table 10.1](#) presents the structure used for the code base found on the companion website.

Directory	Description
build	Location of the Visual Studio solution.
content	Location for nonsource content (such as models, textures, and effects) that the Library project uses. This directory is created in a post-build step for the Library project.
external	Location for third-party libraries (for example, for the DirectXTK and Effects11 libraries that Chapter 3, "Tools of the Trade," discusses).
lib	Output location for the Library project's .lib file. You create this directory in a post-build step.
source\Game	Location for the Game project's source code.
source\Library	Location for the Library project's source code.

**Table 10.1** Project Directory Structure

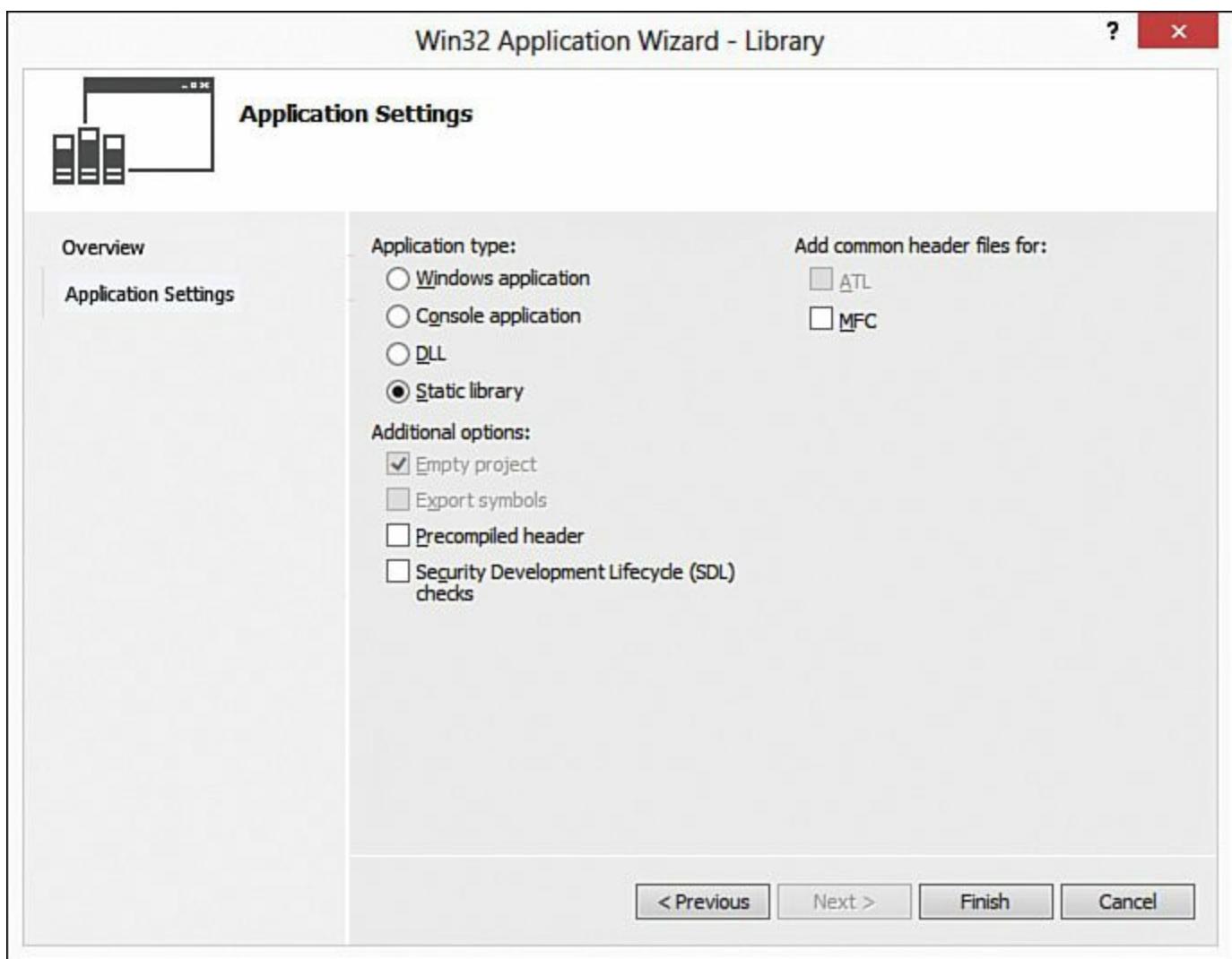
## Project Creation

To adopt this directory structure, create a blank Visual Studio solution inside the build directory. Then right-click the solution within the Solution Explorer panel and choose **Add, New Project** from the context menu to launch the Add New Project Wizard. Choose the Win32 Project template with name and location options, such as those in [Figure 10.1](#).



**Figure 10.1** The Visual Studio 2013 Add New Project Wizard.

This selection launches the Win32 Application Wizard. For the Library project, choose Static Library for the application type and Empty Project for the additional options (see [Figure 10.2](#)). In this image, precompiled headers are disabled; if you'd like to use precompiled headers (for increased compilation speed), just enable that option.

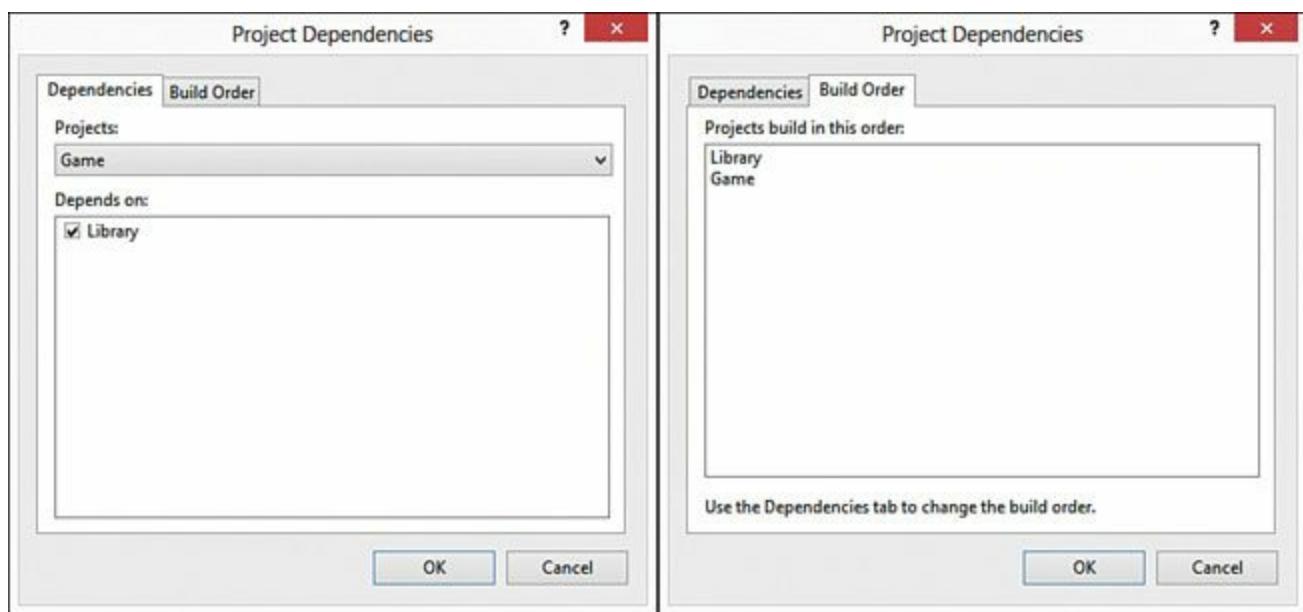


**Figure 10.2** The Visual Studio 2013 Win32 Application Wizard.

Run through these steps a second time to add the Game project, but choose Windows Application for the application type.

## Project Build Order

Next, establish the correct build order by right-clicking the solution or one of the projects in the Solution Explorer panel and choosing **Project Dependencies** from the context menu. Your Game project should depend on your Library project. This configuration builds the Library project first and then builds the Game project (see [Figure 10.3](#)).



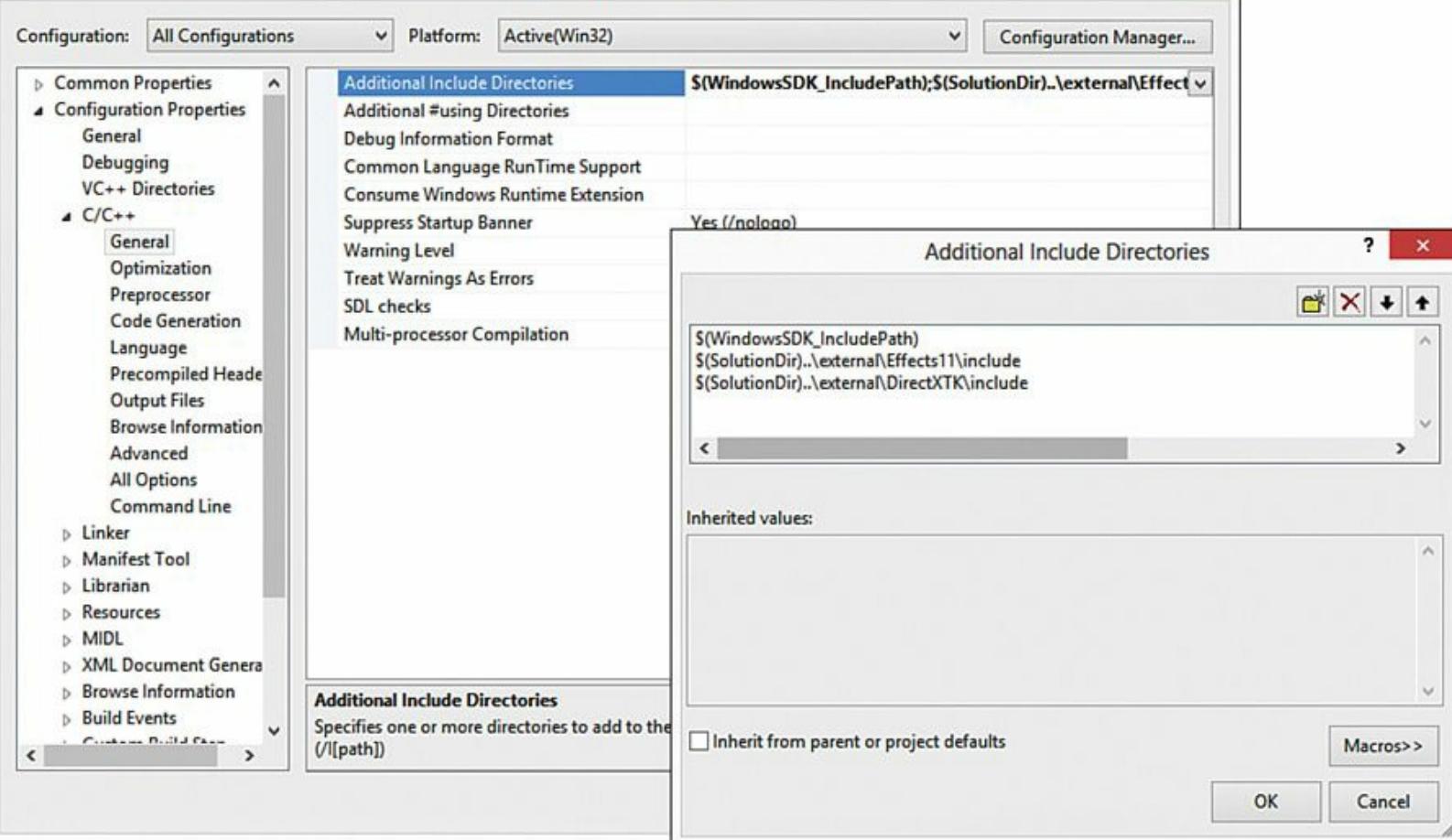
**Figure 10.3** The Visual Studio 2013 Project Dependencies dialog box.

### Note

An alternative to using the Project Dependencies dialog box and manually specifying the Library project as input to the Game project (which is done in the upcoming Linking Libraries section) is to employ the Visual Studio 2013 **Common Properties, References** dialog box. With that system, you specify the Library project as a reference to the Game project. See the online MSDN documentation for more information about project references.

## Include Directories

For your projects to find the header files for DirectX, you need to specify a few directory paths within the project configurations. To do this, open the Property Pages for the Library project by right-clicking the project in the Solution Explorer and choosing **Properties**. Navigate to the section labeled **Configuration Properties, C/C++, General**, and edit the **Additional Include Directories** field to match [Figure 10.4](#). Be sure to do this for both debug and release builds.



**Figure 10.4** The Visual Studio 2013 Additional Include Directories dialog box.

The \$(WindowsSDK\_IncludePath) macro is available after you install the Windows SDK. The other two directories refer to the Effects11 and DirectXTK libraries. Extract those packages to the *external* directory, or update the include directory to match their installed locations.

### Note

Recall the discussion of the Effects 11 and DirectX Tool Kit libraries from [Chapter 3](#). Both libraries are distributed in source form and require that you build the packages before you can use them in your projects.

As the text suggests, you should place the built libraries in the *external* directory, or in some other directory that's readily accessible by the Game and Library projects.

Duplicate these settings for the Game project, but add the following to the list of directories:

[Click here to view code image](#)

```
$(SolutionDir)..\source\Library
```

This setting allows source files in your Game project to include header files from the Library project.

### caution

The **Configuration Properties**, **C/C++** section will not appear unless the project contains at least one .cpp file. If you are following along with your own projects, your

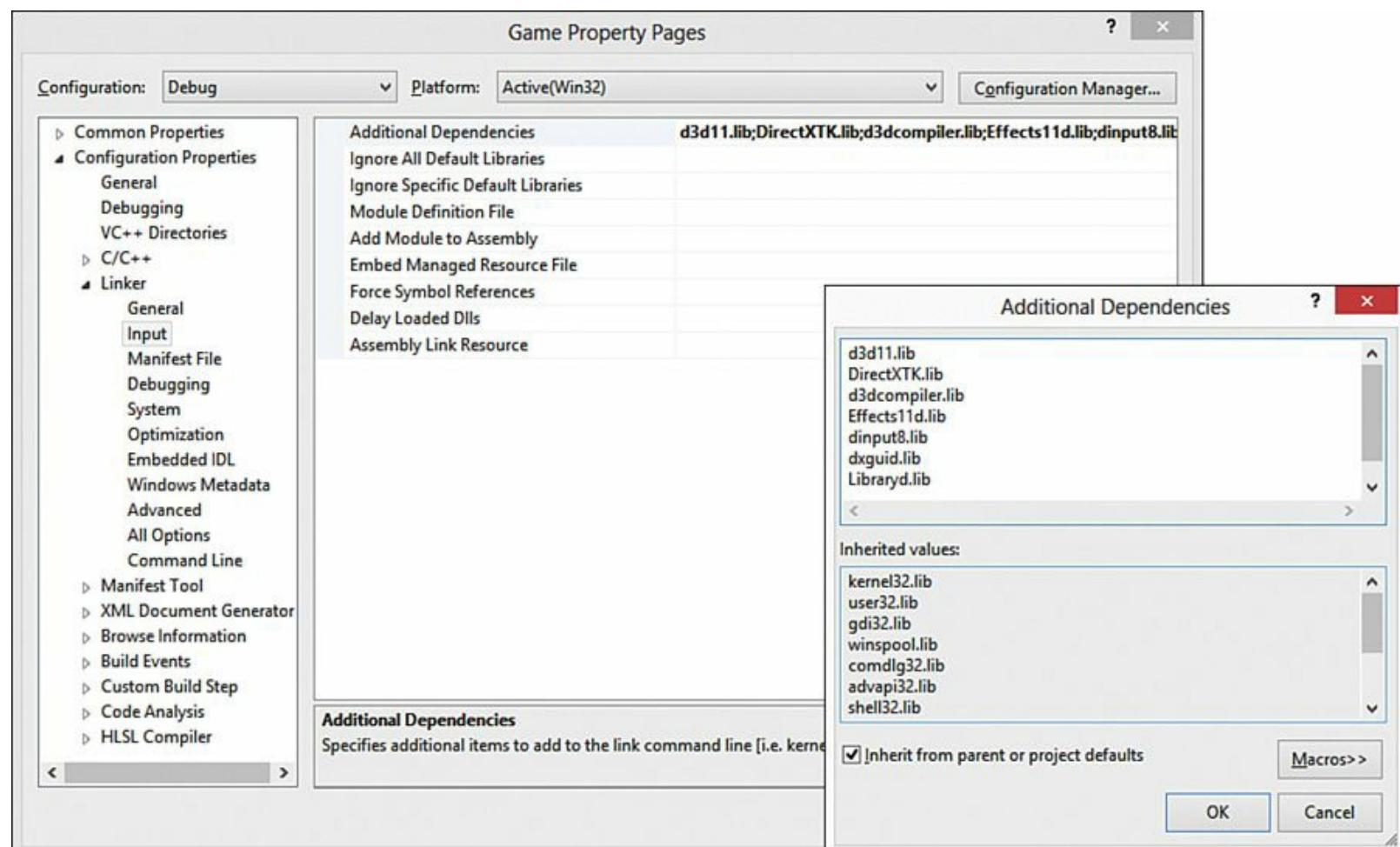
Library project will not contain any .cpp files at this point, and thus the C/C++ section will be hidden. To fix this, add a temporary, empty .cpp file to your Library project. You'll be able to remove the temporary file when you begin adding code.

## Linking Libraries

Next, open the Property Pages for the Game project and navigate to the section labeled **Configuration Properties, Linker, Input**. Edit the **Additional Dependencies** field to match the settings in [Table 10.2](#). Note the differences between the debug and release configurations. For the debug configuration, the Effects11 and Library .lib files have a trailing d to denote that they are debug builds. This is a common convention and allows both debug and release libraries to exist in the same output directory. [Figure 10.5](#) shows the associated Visual Studio dialog boxes.

Configuration	Libraries
Debug	d3d11.lib DirectXTK.lib d3dcompiler.lib Effects11d.lib dinput8.lib dxguid.lib Libraryd.lib
Release	d3d11.lib DirectXTK.lib d3dcompiler.lib Effects11.lib dinput8.lib dxguid.lib Library.lib

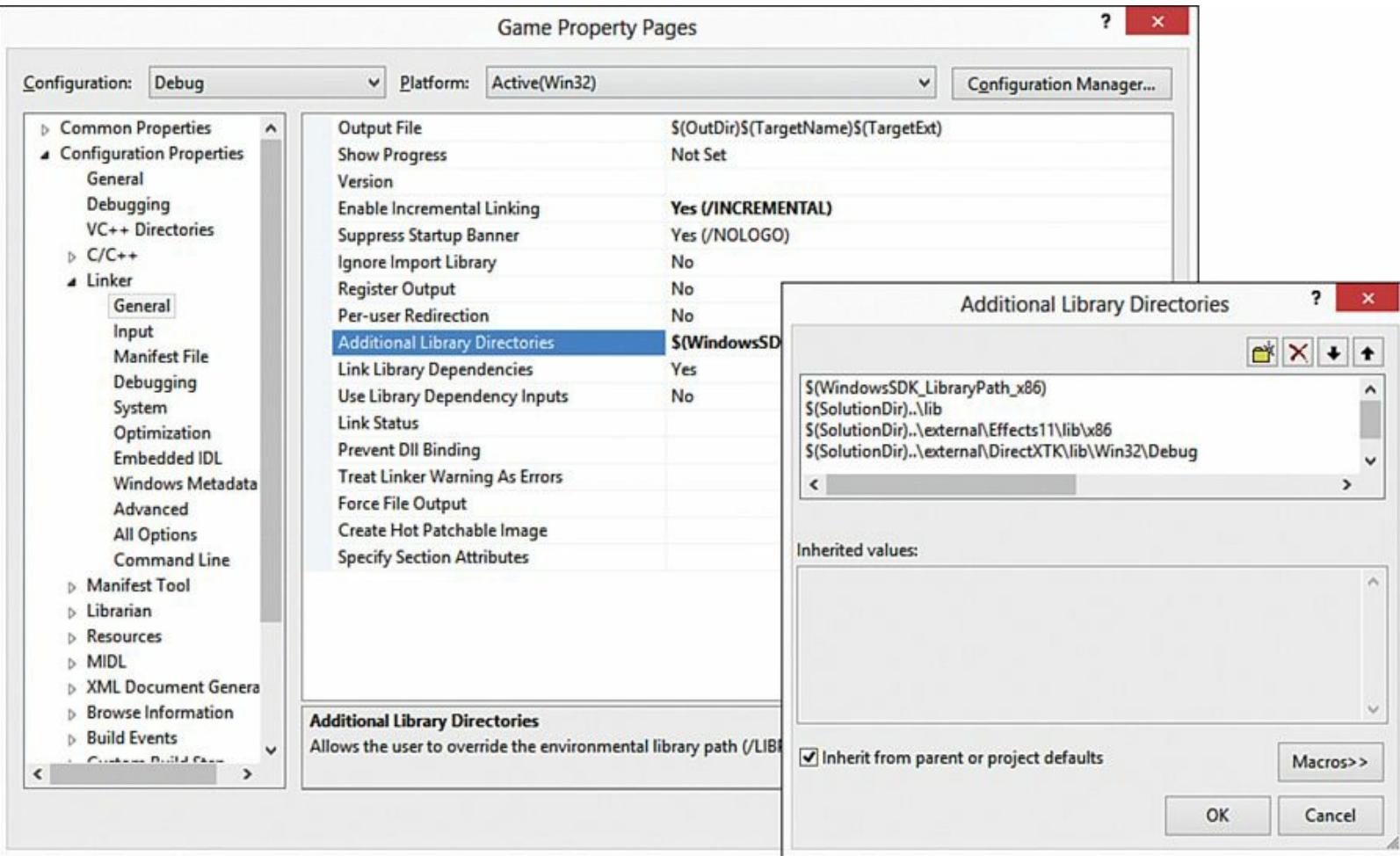
**Table 10.2** Game Project Input Libraries



**Figure 10.5** The Visual Studio 2013 Additional Dependencies dialog box.

As with the include paths, you must provide search locations for Visual Studio to find the specified

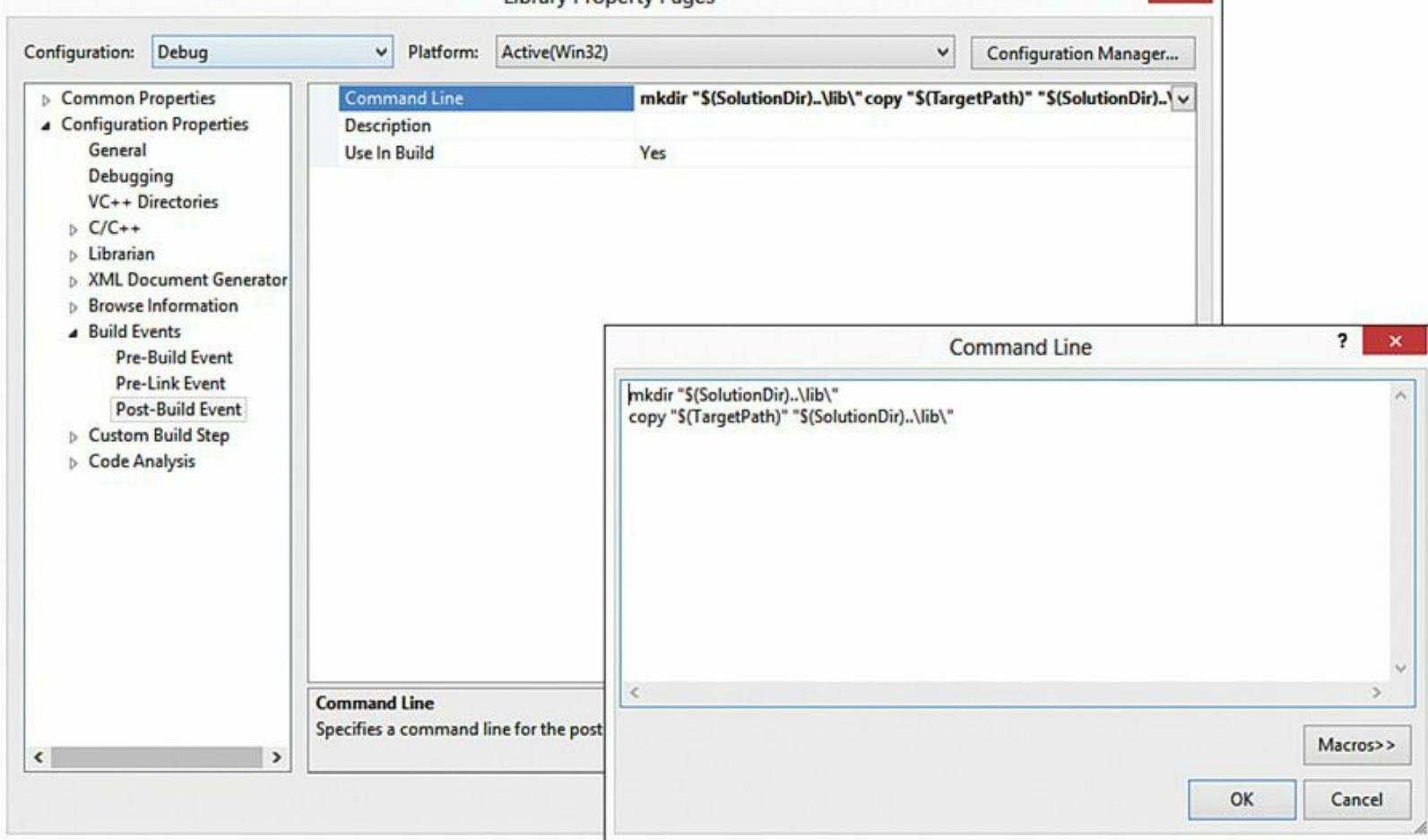
libraries. These directories are provided through the **Additional Library Directories** field under the **Configuration Properties**, **Linker**, **General** section of the Property Pages dialog box. Edit this field to match [Figure 10.6](#). As before, be sure you do this for both debug and release configurations, and note that the DirectXTK library has different directories for debug and release builds.



**Figure 10.6** The Visual Studio 2013 Additional Library Directories dialog box.

## Final Project Settings

With just a bit more configuration, you'll be ready to build your projects. Because the Library project is separate from the Game project (to allow for multiple Game projects), you must make the output of the Library build available to a common location. A location common to both projects is the solution directory, and you can write your .lib file to, for example, \$(SolutionDir)..\\lib. You can configure your Library project to output directly to this location, or you can copy the .lib file through a post-build event. A **build event** is just a set of DOS batch instructions that can be triggered before or after the build. You can find these settings under the section labeled **Configuration Properties**, **Build Events**, and you can edit them through the **Command Line** field for a specific event. [Figure 10.7](#) shows the post-build event used to copy the output .lib file to a directory that the Game project can more readily access.



**Figure 10.7** The Visual Studio 2013 Post-Build Event/Command Line dialog boxes.

If you intend to use a different file name for the debug build of your Library project (such as `Libraryd.lib`), be sure to update the **Target Name** field under the **Configuration Properties, General** section of the Property Pages dialog box.

Finally, the code base presented in the coming chapters uses Unicode strings. To match this, update the **Character Set** field to **Use Unicode Character Set**. You can find this setting under **Configuration Properties, General**.

### Note

You can simplify this setup by collapsing the Library and Game projects, but you won't be able to reuse the common Library code as readily. The demos presented over the next several chapters assume the previous configuration, which allows the demos to be presented independently.

For your own work, you are encouraged to modify the configuration to suit your taste.

## Application Startup

Your project configuration is now complete, and you can start adding source code. Begin by adding a `Program.cpp` file to your Game project, which will house the application entry point. Instead of the traditional `main()` function, Windows programs use `WinMain()` as the startup function. [Listing 10.1](#) presents the startup code used as the base for all the demos of the upcoming chapters.

## Listing 10.1 The Program.cpp File

[Click here to view code image](#)

---

```
#include <memory>
#include "Game.h"
#include "GameException.h"

#if defined(DEBUG) || defined(_DEBUG)
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtDBG.h>
#endif

using namespace Library;

int WINAPI WinMain(HINSTANCE instance, HINSTANCE previousInstance,
LPSTR commandLine, int showCommand)
{
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF |
_CRTDBG_LEAK_CHECK_DF);
#endif

    std::unique_ptr<Game> game(new Game(instance,
L"RenderingClass",
L"Real-Time 3D Rendering", showCommand));

    try
    {
        game->Run();
    }
    catch (GameException ex)
    {
        MessageBox(game->WindowHandle(), ex.whatw().c_str(),
game-
>WindowTitle().c_str(), MB_ABORTRETRYIGNORE);
    }

    return 0;
}
```

---

The `<memory>` include allows the use of `std::unique_ptr`, a smart pointer that destroys the object it maintains when the pointer goes out of scope. The next two includes are for `Game` and `GameException` classes you'll create within the `Library` project. We discuss the `Game` class shortly. The `GameException` class is an extension of `std::exception`. The code for this

class isn't listed here, but is available on the companion website.

Next, note the conditional includes for the CRT debug library (from `crtdbg.h`). When this library is enabled, any memory leaks are listed in Visual Studio's Output panel after the program exits. The call to `_CrtSetDbgFlag()` enables the library.

Just before the `WinMain()` declaration, you'll notice the `using namespace` directive. All the code for the rendering engine is encapsulated in a C++ namespace called `Library` (for example, the `Game` and `GameException` classes). All the demo-related code is housed in the `Game` project and contained within the `Rendering` namespace. I encourage you to make friends with C++ namespaces if you haven't already; they are part of the light side of the Force.

The parameters to the `WinMain()` function are mostly related to window initialization, which you'll handle within the `Game` class. Thus, these values are passed as arguments to the `Game` class constructor. We discuss window initialization momentarily. You can use the `commandLine` parameter to pass information to the startup method from the DOS command line or Windows shortcut.

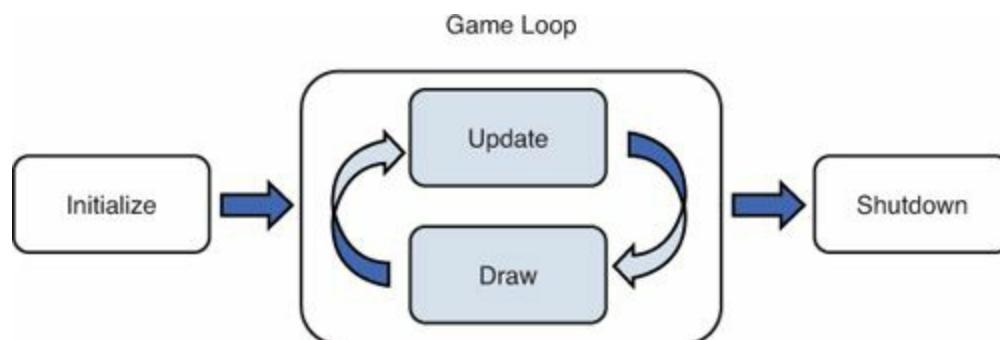
The body of `WinMain()` is quite short. It contains the `Game` class instantiation followed by a call to `Game::Run()`. This call is wrapped within a `try/catch` block that presents a message box whenever there's a problem within `Game::Run()`. The `Run()` method encapsulates the application's **game loop**.

### Note

The code in this text is written specifically for the Windows platform, and you'll find Windows-specific references throughout the code base. You'll also encounter various Visual Studio language extensions that might not be happy in other environments. I chose this route so as not to get mired in cross-platform complexity, when the focus of the text is on rendering. With a bit of effort, the framework in these chapters will port to Windows RT and Xbox One. With a bit more effort, this work could be independent of the graphics API (that is, it could support either DirectX or OpenGL).

## The Game Loop

The game loop is the system that makes calls to update the objects in your scene and draw them to the screen. Typically, the game loop is initialized and executed shortly after program startup and runs continuously until the program is terminated. A variety of game loop designs exist, but [Figure 10.8](#) illustrates the most basic.



**Figure 10.8** A basic game loop.

From a code perspective, this game loop looks something like [Listing 10.2](#).

## Listing 10.2 A Basic Game Loop in C++

[Click here to view code image](#)

---

```
void Game::Run()
{
    Initialize();

    while(isRunning)
    {
        mGameClock.UpdateGameTime(mGameTime);

        Update(mGameTime);
        Draw(mGameTime);
    }

    Shutdown();
}

void Game::Exit()
{
    isRunning = false;
}
```

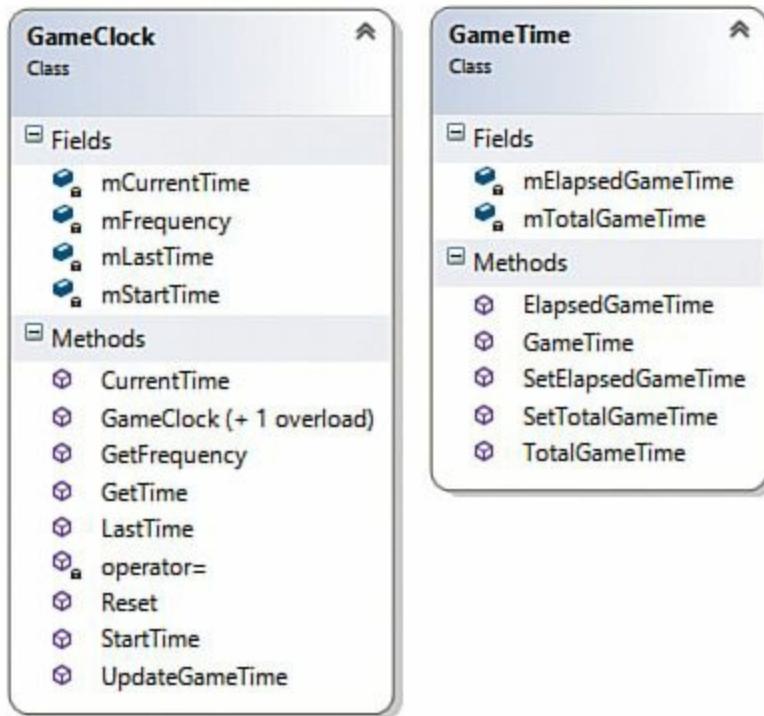
---

The `Game::Initialize()` method refers to window and Direct3D initialization (which you'll actually split into two separate functions), along with game-specific initialization steps. The `Game::Update()` method performs any non-rendering-related operations. For example, the `Update()` method might process keyboard and mouse input, rotate an object, or update its position. Conversely, the `Draw()` method handles all graphics-related instructions. These methods accommodate updating and drawing for all the objects in a scene.

## Time

Notice the `UpdateGameTime()` method in [Listing 10.2](#) and the associated `mGameTime` member that's passed to the `Update()` and `Draw()` methods. These elements are about time, which is important in the context of game and graphics programming. Two pieces of time-related information are of interest: the time elapsed since the clock started and the time elapsed since the last iteration of the game loop. The latter applies to what is commonly known as the game's **frame rate**. An iteration of the game loop counts as one **frame**, and the frame rate is the number of frames executed in a second.

The elapsed-time data and associated functionality are encapsulated within the `GameClock` and `GameTime` classes. [Figure 10.9](#) shows class diagrams for these types. [Listing 10.3](#) presents the `GameClock` header file.



**Figure 10.9** Class diagrams for the `GameClock` and `GameTime` classes.

### Listing 10.3 The `GameClock.h` Header File

[Click here to view code image](#)

---

```

#pragma once

#include <windows.h>
#include <exception>

namespace Library
{
    class GameTime;

    class GameClock
    {
public:
    GameClock();

    const LARGE_INTEGER& StartTime() const;
    const LARGE_INTEGER& CurrentTime() const;
    const LARGE_INTEGER& LastTime() const;

    void Reset();
    double GetFrequency() const;
    void GetTime(LARGE_INTEGER& time) const;
    void UpdateGameTime(GameTime& gameTime);
}

```

```

private:
    GameClock(const GameClock& rhs);
    GameClock& operator=(const GameClock& rhs);

    LARGE_INTEGER mStartTime;
    LARGE_INTEGER mCurrentTime;
    LARGE_INTEGER mLastTime;
    double mFrequency;
};

}

```

---

The `GameClock` class wraps calls to the Windows API high-resolution performance counter, a timer that increments a value at a frequency expressed in *counts per second*. You determine the frequency through a call to `QueryPerformanceFrequency()`, which modifies a parameter of type `LARGE_INTEGER` (a 64-bit integer, or `long long`). You store the frequency in a class member and use it for subsequent elapsed-time calculations.

You can get the current state of the high-resolution timer through the `QueryPerformanceCounter()` function. To compute the elapsed time of an iteration of the game loop, you call `QueryPerformanceCounter()` twice, once at the beginning of the loop (before the `Update()` call) and once at the end of the loop (after the `Draw()` call). The elapsed time (in seconds) is calculated as:

$$time_{elapsed} = (stopTime - startTime) / frequency$$

To calculate the total elapsed time since the start of the program, you can either accumulate the elapsed time per frame or save off the result of an initial call to `QueryPerformanceCounter()` and then use it with the aforementioned equation. [Listing 10.4](#) presents the implementation of the `GameClock` class.

#### **Listing 10.4** The `GameClock.cpp` File

[Click here to view code image](#)

---

```

#include "GameClock.h"
#include "GameTime.h"

namespace Library
{
    GameClock::GameClock()
        : mStartTime(), mCurrentTime(), mLastTime(),
        mFrequency()
    {
        mFrequency = GetFrequency();
        Reset();
    }
}

```

```
const LARGE_INTEGER& GameClock::StartTime() const
{
    return mStartTime;
}

const LARGE_INTEGER& GameClock::CurrentTime() const
{
    return mCurrentTime;
}

const LARGE_INTEGER& GameClock::LastTime() const
{
    return mLastTime;
}

void GameClock::Reset()
{
    GetTime(mStartTime);
    mCurrentTime = mStartTime;
    mLastTime = mCurrentTime;
}

double GameClock::GetFrequency() const
{
    LARGE_INTEGER frequency;

    if (QueryPerformanceFrequency(&frequency) == false)
    {
        throw std::exception("QueryPerformanceFrequency() failed.");
    }

    return (double)frequency.QuadPart;
}

void GameClock::GetTime(LARGE_INTEGER& time) const
{
    QueryPerformanceCounter(&time);
}

void GameClock::UpdateGameTime(GameTime& gameTime)
{
    GetTime(mCurrentTime);
    gameTime.SetTotalGameTime( (mCurrentTime.QuadPart - mStartTime.
QuadPart) / mFrequency );
    gameTime.SetElapsedGameTime( (mCurrentTime.QuadPart -
```

```
mLastTime.  
QuadPart) / mFrequency);  
mLastTime = mCurrentTime;  
}  
}
```

---

In this implementation, the `mStartTime` member stores the counter value for computing the total elapsed time and is assigned in the `Reset()` method. The members `mcurrentTime` and `mLastTime` are used for storing subsequent calls to `QueryPerformanceCounter()` to compute the elapsed time per frame. The `UpdateGameTime()` method encapsulates those elapsed-time calculations instead of forcing that code into the `Game::Run()` method. The frequency is queried on `GameClock` instantiation. The `GameTime` class has no special functionality; it simply contains accessors and mutators for its `mElapsedGameTime` and `mTotalElapsedGameTime` members. This and all source code is available on the book's companion website.

## Game Loop Initialization

The `Game` class is the heart of the rendering engine and encapsulates the game loop. The initialization stage of the game loop is executed with calls to three virtual methods of the `Game` class: `InitializeWindow()`, `InitializeDirectX()`, and `Initialize()`; respectively, they create a window for your application to draw to, set up DirectX, and perform generic component initialization (discussed shortly). The `Game` class is intended to be inherited (although it's not officially an abstract base class because it contains a full implementation), and because all three methods are virtual, they can be customized by the derived class. However, the base `Game` class provides general-purpose implementations of each of these methods.

## Window Initialization

Your DirectX game and graphics applications reside within windows, as do all Microsoft Windows programs, and you must create these containers with calls to the Windows API. [Listing 10.5](#) presents the header file for the `Game` class, containing just those members necessary for window initialization. In [Chapter 11, “Direct3D Initialization,”](#) the `Game` class will expand to include data and methods for initializing Direct3D.

### **Listing 10.5** The Game.h Header File

[Click here to view code image](#)

---

```
#pragma once  
  
#include <windows.h>  
#include <string>  
#include "GameClock.h"  
#include "GameTime.h"  
  
namespace Library  
{
```

```
class Game
{
public:
    Game(HINSTANCE instance, const std::wstring&
windowClass, const
std::wstring& windowTitle, int showCommand);
    virtual ~Game();

    HINSTANCE Instance() const;
    HWND WindowHandle() const;
    const WNDCLASSEX& Window() const;
    const std::wstring& WindowClass() const;
    const std::wstring& WindowTitle() const;
    int ScreenWidth() const;
    int ScreenHeight() const;

    virtual void Run();
    virtual void Exit();
    virtual void Initialize();
    virtual void Update(const GameTime& gameTime);
    virtual void Draw(const GameTime& gameTime);

protected:
    virtual void InitializeWindow();
    virtual void Shutdown();

    static const UINT DefaultScreenWidth;
    static const UINT DefaultScreenHeight;

    HINSTANCE mInstance;
    std::wstring mWindowClass;
    std::wstring mWindowTitle;
    int mShowCommand;

    HWND mWindowHandle;
    WNDCLASSEX mWindow;

    UINT mScreenWidth;
    UINT mScreenHeight;

    GameClock mGameClock;
    GameTime mGameTime;

private:
    Game(const Game& rhs);
    Game& operator=(const Game& rhs);
```

```

POINT CenterWindow(int windowHeight, int windowWidth);
static LRESULT WINAPI WndProc(HWND windowHandle, UINT
message,
WPARAM, LPARAM lParam);
}

```

---

As you can see, initializing a window requires a lot of declaration. First, the Game constructor accepts arguments for an instance handle and a window class name. The initial parameter is a handle to the program instance that contains the **windows procedure**, a call back for all messages passed from the operating system to your window. The second parameter (the window class name) is just a string. Together, these arguments uniquely identify your window among the sea of windows in your operating system. The instance handle is a pass-through parameter that comes from your program's entry point `WinMain()` as is the `showCommand` parameter, which determines how your window will be shown. The `windowTitle` parameter is the string used for the window's title bar. All parameters are saved to class members for use in the `InitializeWindow()` method. [Listing 10.6](#) presents the implementation of the `InitializeWindow()` method.

## **Listing 10.6** The Game::InitializeWindow() Method

[Click here to view code image](#)

---

```

void Game::InitializeWindow()
{
    ZeroMemory(&mWindow, sizeof(mWindow));
    mWindow.cbSize = sizeof(WNDCLASSEX);
    mWindow.style = CS_CLASSDC;
    mWindow.lpfnWndProc = WndProc;
    mWindow.hInstance = mInstance;
    mWindow.hIcon = LoadIcon(nullptr, IDI_APPLICATION);
    mWindow.hIconSm = LoadIcon(nullptr, IDI_APPLICATION);
    mWindow.hCursor = LoadCursor(nullptr, IDC_ARROW);
    mWindow.hbrBackground = GetSysColorBrush(COLOR_BTNFACE);
    mWindow.lpszClassName = mWindowClass.c_str();

    RECT windowRectangle = { 0, 0, mScreenWidth, mScreenHeight };
    AdjustWindowRect(&windowRectangle, WS_OVERLAPPEDWINDOW,
FALSE);

    RegisterClassEx(&mWindow);
    POINT center = CenterWindow(mScreenWidth, mScreenHeight);
    mWindowHandle = CreateWindow(mWindowClass.c_str(),
mWindowTitle.c_
str(), WS_OVERLAPPEDWINDOW, center.x, center.y,
windowRectangle.right

```

```

- windowRectangle.left, windowRectangle.bottom -
windowRectangle.top,
nullptr, nullptr, mInstance, nullptr);

ShowWindow(mWindowHandle, mShowCommand);
UpdateWindow(mWindowHandle);
}

```

---

A window is created with calls to `RegisterClassEx()` and `CreateWindow()` and subsequently is displayed with a call to `ShowWindow()`. The window is defined through the `WNDCLASSEX` type and has myriad options for customizing its look. The `AdjustWindowRect()` function creates a **client area** whose size is specified by the `mScreenWidth` and `mScreenHeight` class members. The client area is the inside the window, excluding elements such as the title bar, status bar, or scrollbars.

The `CenterWindow()` call centers the window within the screen and is not a Windows API function. [Listing 10.7](#) presents the implementation of this method and the `WndProc()` callback function.

## **Listing 10.7** Implementations for `Game::WndProc()` and `Game::CenterWindow()`

[Click here to view code image](#)

---

```

LRESULT WINAPI Game::WndProc(HWND windowHandle, UINT message,
WPARAM,
LPARAM lParam)
{
    switch(message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }

    return DefWindowProc(windowHandle, message, wParam, lParam);
}

POINT Game::CenterWindow(int windowWidth, int windowHeight)
{
    int screenWidth = GetSystemMetrics(SM_CXSCREEN);
    int screenHeight = GetSystemMetrics(SM_CYSCREEN);

    POINT center;
    center.x = (screenWidth - windowWidth) / 2;
    center.y = (screenHeight - windowHeight) / 2;

    return center;
}

```

}

---

The CenterWindow() method uses the Windows API GetSystemMetrics() function to determine the screen's width and height, and returns a point at the center of the screen. The windows procedure is required for all Windows applications. The WndProc() implementation simply passes all messages to the default windows procedure, except for the message to destroy the window after it has been removed from the screen (that is, the application should shut down). In response to this message, the PostQuitMessage() function is called and posts a WM\_QUIT message to the Windows message queue. When the application receives the WM\_QUIT message, the Windows message loop exits and the application ends.

With this code in place, you can update your Game::Run() method to [Listing 10.8](#).

### **Listing 10.8** The Game::Run() Method

[Click here to view code image](#)

---

```
void Game::Run()
{
    InitializeWindow();

    MSG message;
    ZeroMemory(&message, sizeof(message));

    mGameClock.Reset();

    while (message.message != WM_QUIT)
    {
        if (PeekMessage(&message, nullptr, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&message);
            DispatchMessage(&message);
        }
        else
        {
            mGameClock.UpdateGameTime(mGameTime);
            Update(mGameTime);
            Draw(mGameTime);
        }
    }

    Shut down();
}
```

---

This is the game loop, but augmented to accommodate window initialization and messages from the Windows message queue. The Windows function PeekMessage() looks at the queue for new messages and dispatches them with the calls to TranslateMessage() and

`DispatchMessage()`. The `WndProc()` method processes the resulting messages. You can find more details in the online documentation on creating windows and the Windows message loop, but this is enough to get a window on the screen. If you build and run your application, you should see the image in [Figure 10.10](#).



**Figure 10.10** A blank window created through the demo framework.

This might seem a bit anti-climactic, but you're much closer to having something rendering to the screen. The next step is to initialize Direct3D.

## Summary

In this chapter, you established the foundation of your rendering engine. You set up the appropriate Visual Studio projects, developed a game loop, and initialized a window through the Windows API. In the next chapter, you extend this work to build your first Direct3D demo.

## Exercise

1. From within the debugger, walk through the code used to initialize the game loop and create a blank window. Visit the book's companion website to find an associated demo application.

# Chapter 11. Direct3D Initialization

In this chapter, you finalize the foundation of your rendering engine. You take your first look at the Direct3D C++ API and write an initial Direct3D application. Along the way, you revisit some previously discussed topics, including the DirectX swap chain and depth/stencil buffering.

## Initializing Direct3D

You can perform Direct3D initialization through the following steps:

1. Create the Direct3D device and device context interfaces.
2. Check for multisampling support.
3. Create the swap chain.
4. Create a render target view.
5. Create a depth-stencil view.
6. Associate the render target view and depth-stencil view with the output-merger stage of the Direct3D pipeline.
7. Set the viewport.

The next few sections cover each of these steps.

## Creating the Device and Device Context

A Direct3D **device** represents a **display adapter**, an abstraction of the underlying graphics capability of your computer. Your PC likely contains several adapters, including the graphics card itself (fast) and adapters that are implemented entirely in software (slow). A device is represented with the `ID3D11Device` interface, which creates resources and enumerates the capabilities of the display adapter.

A **device context** also represents the display adapter, but within a particular setting. The device context is encapsulated in the `ID3D11DeviceContext` interface, which generates rendering commands using the resources of a device. You create a Direct3D device through the function `D3D11CreateDevice()` whose prototype and parameters are listed here:

[Click here to view code image](#)

```
HRESULT WINAPI D3D11CreateDevice(
    IDXGIAdapter* pAdapter,
    D3D_DRIVER_TYPE DriverType,
    HMODULE Software,
    UINT Flags,
    CONST D3D_FEATURE_LEVEL* pFeatureLevels,
    UINT FeatureLevels,
    UINT SDKVersion,
    ID3D11Device** ppDevice,
    D3D_FEATURE_LEVEL* pFeatureLevel,
    ID3D11DeviceContext** ppImmediateContext );
```

■ **pAdapter:** A pointer to the display adapter that this device will represent. Pass a value of

NULL to use the default adapter. You can enumerate the available adapters with a call to `IDXGIFactory::EnumAdapters()`.

- **DriverType:** The driver type to use with the device. Specify `D3D_DRIVER_TYPE_HARDWARE` to use the actual video card. This offers the best performance. However, you could specify `D3D_DRIVER_TYPE_REFERENCE` or `D3D_DRIVER_TYPE_WARP` for software implementations provided by Microsoft. See the online documentation for a complete list of driver types. All the work in this book uses the hardware driver.
- **Software:** A handle to a third-party DLL that implements Direct3D in software. This parameter is used in conjunction with a driver type of `D3D_DRIVER_TYPE_SOFTWARE`.
- **Flags:** Options to use when creating the Direct3D device. This parameter consists of bitwise OR'd values from the `D3D11_CREATE_DEVICE_FLAG` enumeration. A common option is `D3D11_CREATE_DEVICE_DEBUG` to create a device with debugging capability.
- **pFeatureLevels:** A pointer to an array of targeted **feature levels**, listed in order of preference from most to least preferred. Feature levels describe the capabilities of a device and roughly correspond to a version of Direct3D. If you specify a value of NULL, you get the highest feature level the device supports—*unless* that device supports Direct3D version 11.1. In that scenario, you see support for only version 11.0, not version 11.1. You must explicitly list `D3D_FEATURE_LEVEL_11_1` in the **pFeatureLevels** array if you want to target Direct3D 11.1. Furthermore, if you specify 11.1 as a targeted feature level and the device doesn't support it, the `D3D11CreateDevice()` function fails. [Table 11.1](#) lists the possible feature levels.

Feature Level	Description
<code>D3D_FEATURE_LEVEL_11_1</code>	Supports Direct3D 11.1, including shader model 5
<code>D3D_FEATURE_LEVEL_11_0</code>	Supports Direct3D 11.0, including shader model 5
<code>D3D_FEATURE_LEVEL_10_1</code>	Supports Direct3D 10.1, including shader model 4
<code>D3D_FEATURE_LEVEL_10_0</code>	Supports Direct3D 10.0, including shader model 4
<code>D3D_FEATURE_LEVEL_9_3</code>	Supports Direct3D 9.3, including shader model 2.0b
<code>D3D_FEATURE_LEVEL_9_2</code>	Supports Direct3D 9.2, including shader model 2
<code>D3D_FEATURE_LEVEL_9_1</code>	Supports Direct3D 9.1, including shader model 2

**Table 11.1** Direct3D Feature Levels

- **FeatureLevels:** Specifies the number of elements in the **pFeatureLevels** array.
- **SDKVersion:** States the version of the SDK. You will always specify `D3D11_SDK_VERSION`.
- **ppDevice:** Returns the created device.
- **pFeatureLevel:** Returns the selected feature level.
- **ppImmediateContext:** Returns the created device context.

[Listing 11.1](#) gives an example of the `D3D11CreateDevice()` call.

## **Listing 11.1 An Example Call to D3D11CreateDevice()**

[Click here to view code image](#)

---

```
HRESULT hr;
UINT createDeviceFlags = 0;

#if defined(DEBUG) || defined(_DEBUG)
    createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

D3D_FEATURE_LEVEL featureLevels[] = {
    D3D_FEATURE_LEVEL_11_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_10_0
};

ID3D11Device* direct3DDevice;
D3D_FEATURE_LEVEL selectedFeatureLevel;
ID3D11DeviceContext* direct3DDeviceContext;
if (FAILED(hr = D3D11CreateDevice(NULL,
D3D_DRIVER_TYPE_HARDWARE,
NULL, createDeviceFlags, featureLevels,
ARRAYSIZE(featureLevels),
D3D11_SDK_VERSION, &direct3DDevice, &selectedFeatureLevel,
&direct3DDeviceContext)))
{
    throw GameException("D3D11CreateDevice() failed", hr);
}
```

---

## **Checking for Multisampling Support**

**Multisample Anti-Aliasing (MSAA)** is a technique to improve image quality. **Aliasing** refers to the jagged look of a line or triangle edge when presented on a computer monitor. It occurs because monitors use discrete points of light (pixels) to display virtual objects. As small as a pixel might be, it is not infinitesimal. Anti-aliasing refers to techniques to remove these “jaggies.” [Figure 11.1](#) shows an example of a shape before and after anti-aliasing is applied.



**Figure 11.1** An example of a shape rendered before (top) and after (bottom) anti-aliasing is applied.

Reducing the size of each pixel can ameliorate aliasing, so increasing your monitor's resolution can help. But even at your monitor's maximum resolution, aliasing artifacts might still be unacceptable. **Supersampling** is a technique that reduces the effective size of a pixel by increasing the resolution of the render target. Thus, multiple *subpixels* are sampled and averaged together (in a process known as **downsampling**) to produce the final color of each pixel displayed. A render target four times (4x) larger than the display resolution is commonly used for super-sampling, although higher resolutions can be used at the cost of memory and performance. With supersampling, the pixel shader is executed for each subpixel. Thus, to produce a render target 4x larger than the display, the pixel shader must run 4x as often. Multisampling is an optimization of supersampling that executes the pixel shader just once per display resolution pixel (at the pixel's center) and uses that color for each of the subpixels. All Direct3D 11 devices are required to support 4x MSAA, although they can support even higher-quality levels. You can query the available quality levels through

`ID3D11Device::CheckMultisampleQualityLevels()`; the prototype and parameters are listed here:

[Click here to view code image](#)

```
HRESULT CheckMultisampleQualityLevels(
    DXGI_FORMAT Format,
    UINT SampleCount,
    UINT *pNumQualityLevels);
```

■ **Format:** The texture format. Commonly, you specify this parameter as

`DXGI_FORMAT_R8G8B8A8_UNORM`, a four-component format with 8 bits for each of the red, green, blue, and alpha channels. However, you can choose from many texture formats. Visit the online documentation for details.

■ **SampleCount:** The number of samples per pixel (for example, 4 for 4x multisampling).

- **pNumQualityLevels:** The number of quality levels supported for the given format and sample count. The actual meaning of “quality level” can differ among hardware manufacturers. Generally, higher quality levels mean better results at higher performance cost. A returned value of 0 indicates that this device does not support the combination of format and sample count.

The multisampling quality values are used during swap chain creation, so a code example is deferred to the next section.

## Creating the Swap Chain

A **swap chain** is a set of buffers used to display frames to the monitor. As discussed back in [Chapter 4, “Hello, Shaders!”](#), double buffering refers to a swap chain containing two buffers, a front buffer and a back buffer. The front buffer is what’s actually displayed while the back buffer is being filled with new data. When the application **presents** the newly completed back buffer, the two are swapped and the process starts over.

## Populating a Swap Chain Description

You create the swap chain by first populating an instance of the `DXGI_SWAP_CHAIN_DESC1` structure, whose members are presented and described here:

[Click here to view code image](#)

```
typedef struct DXGI_SWAP_CHAIN_DESC1
{
    UINT Width;
    UINT Height;
    DXGI_FORMAT Format;
    BOOL Stereo;
    DXGI_SAMPLE_DESC SampleDesc;
    DXGI_USAGE BufferUsage;
    UINT BufferCount;
    DXGI_SCALING Scaling;
    DXGI_SWAP_EFFECT SwapEffect;
    DXGI_ALPHA_MODE AlphaMode;
    UINT Flags;
} DXGI_SWAP_CHAIN_DESC1;
```

- **Width:** The resolution width.
- **Height:** The resolution height.
- **Format:** The display format of the back buffer, e.g. `DXGI_FORMAT_R8G8B8A8_UNORM`.
- **Stereo:** Whether the back buffer will be used for stereoscopic rendering.
- **SampleDesc:** Multisampling parameters. When multisampling is enabled, this member is populated with data validated through `ID3D11Device::CheckMultisampleQualityLevels()`.
- **BufferUsage:** Buffer usage and CPU access options. For example, the back buffer is commonly used for render target output (`DXGI_USAGE_RENDER_TARGET_OUTPUT`) but can also be

used for shader input (`DXGI_USAGE_SHADER_INPUT`).

■ **BufferCount:** The number of back buffers to include in the swap chain. A value of 1 indicates double buffering (a front buffer and *one* back buffer). A value of 2 indicates two back buffers (triple buffering).

■ **Scaling:** The resize behavior when the size of the back buffer is not equal to the target output (such as with a window). For example, a value of `DXGI_SCALING_STRETCH` scales the image to fit the output.

■ **SwapEffect:** How the contents of a buffer are handled after being presented.

`DXGI_SWAP_EFFECT_DISCARD` is the most efficient of the options and is the only one that supports multisampling.

■ **AlphaMode:** The transparency behavior of the back buffer.

■ **Flags:** Options to use when creating the swap chain. This value consists of bitwise OR'd values from the `D3D11_SWAP_CHAIN_FLAG` enumeration. Visit the online documentation for a list of available options.

[Listing 11.2](#) presents the code for populating a `DXGI_SWAP_CHAIN_DESC1` instance and the aforementioned `CheckMultisampleQualityLevels()` call.

## Listing 11.2 Populating a Swap Chain Description Structure

[Click here to view code image](#)

---

```
mDirect3DDevice->CheckMultisampleQualityLevels(DXGI_FORMAT_R8G8B8A8_UNORM,mMultiSamplingCount, &mMultiSamplingQualityLevels);  
if (mMultiSamplingQualityLevels == 0)  
{  
    throw GameException("Unsupported multi-sampling quality");  
}  
  
DXGI_SWAP_CHAIN_DESC1 swapChainDesc;  
ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));  
swapChainDesc.Width = mScreenWidth;  
swapChainDesc.Height = mScreenHeight;  
swapChainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;  
  
if (mMultiSamplingEnabled)  
{  
    swapChainDesc.SampleDesc.Count = mMultiSamplingCount;  
    swapChainDesc.SampleDesc.Quality =  
mMultiSamplingQualityLevels - 1;  
}  
else  
{  
    swapChainDesc.SampleDesc.Count = 1;
```

```
    swapChainDesc.SampleDesc.Quality = 0;  
}  
  
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;  
swapChainDesc.BufferCount = 1;  
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
```

---

Notice that the code in [Listing 11.2](#) omits explicit assignment of the fields `Stereo`, `Scaling`, `AlphaMode`, and `Flags`. This is possible because the “zero” values align with the default values for these members.

## Populating a Full-Screen Description

You can define your swap chain as either full screen or windowed. To create a full-screen swap chain, you must populate an instance of the `DXGI_SWAP_CHAIN_FULLSCREEN_DESC` structure; its members are presented and described next:

[Click here to view code image](#)

```
typedef struct DXGI_SWAP_CHAIN_FULLSCREEN_DESC  
{  
    DXGI_RATIONAL RefreshRate;  
    DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;  
    DXGI_MODE_SCALING Scaling;  
    BOOL Windowed;  
} DXGI_SWAP_CHAIN_FULLSCREEN_DESC;
```

- **RefreshRate:** Describes the refresh rate of the display, in hertz. The `DXGI_RATIONAL` structure has two fields, `Numerator` and `Denominator`, both unsigned integers. If your refresh rate were, for example, 60Hz, you would specify 60 for the numerator and 1 for the denominator.
- **ScanlineOrdering:** Indicates how the monitor should draw the image. For example, `DXGI_MODE_SCANLINE_ORDER_PROGRESSIVE` specifies that the image should be created one line after another without skipping any (noninterlaced scanning).
- **Scaling:** Indicates how the image is stretched to fit the monitor’s resolution. For example, `DXGI_MODE_SCALING_CENTERED` specifies that the image should be centered on the display and that no scaling should be performed.
- **Windowed:** Specifies whether the swap chain is in windowed mode.

[Listing 11.3](#) presents the code for populating a `DXGI_SWAP_CHAIN_FULLSCREEN_DESC` instance.

## Listing 11.3 Populating a Full-Screen Description Structure

[Click here to view code image](#)

---

```
DXGI_SWAP_CHAIN_FULLSCREEN_DESC fullScreenDesc;  
ZeroMemory(&fullScreenDesc, sizeof(fullScreenDesc));
```

```
fullScreenDesc.RefreshRate.Numerator = mFrameRate;
fullScreenDesc.RefreshRate.Denominator = 1;
fullScreenDesc.Windowed = !mIsFullScreen;
```

---

## DirectX Graphics Infrastructure Interfaces

With swap chain and full-screen descriptions ready, you can create the swap chain through a call to `IDXGIFactory2::CreateSwapChainForHwnd()`. The prototype and parameters are listed here:

[Click here to view code image](#)

```
HRESULT CreateSwapChainForHwnd(
    IUnknown *pDevice,
    HWND hWnd,
    const DXGI_SWAP_CHAIN_DESC1 *pDesc,
    const DXGI_SWAP_CHAIN_FULLSCREEN_DESC *pFullscreenDesc,
    IDXGIOutput *pRestrictToOutput,
    IDXGISwapChain1 **ppSwapChain);
```

- **pDevice:** The Direct3D device.
- **hWnd:** The window handle to associate with the swap chain. This is the return value of the `CreateWindow()` call.
- **pDesc:** The swap chain description structure.
- **pFullscreenDesc:** The full-screen description structure. You can set this to NULL to use a windowed swap chain, or you can set the `Windowed` member of the description structure to true.
- **pRestrictToOutput:** Enables you to restrict content to a specific output target. If this parameter is set to NULL, the output is not restricted.
- **ppSwapChain:** Returns the created swap chain.

Before examining a code sample, notice that the `CreateSwapChainForHwnd()` function is a member of the `IDXGIFactory2` interface, which no section thus far has explicitly defined. DXGI is the DirectX Graphics Infrastructure and manages such tasks as presenting rendered frames to the screen and transitioning from windowed to full-screen display modes. These tasks are common across many APIs and are therefore independent of Direct3D. An `IDXGIFactory2` object was instantiated when you created the Direct3D device, but you need to query the device to extract the required interface. You do this through the `IUnknown::QueryInterface()` method. `IUnknown` is a Component Object Model (COM) interface, and COM interfaces have different acquisition and release patterns than typical C++ classes. After you query the interface, you call `IUnknown::Release()` when you are finished with the object. [Listing 11.4](#) presents the code required to acquire the `IDXGIFactory2` instance, along with the call to `CreateSwapChainForHwnd()`.

### Listing 11.4 Creating a Swap Chain

[Click here to view code image](#)

```
#define ReleaseObject(object) if((object) != NULL) { object-  
>Release();  
object = NULL; }  
  
IDXGIDevice* dxgiDevice = nullptr;  
if (FAILED(hr = mDirect3DDevice-  
>QueryInterface(__uuidof(IDXGIDevice),  
reinterpret_cast<void**>(&dxgiDevice))))  
{  
    throw GameException("ID3D11Device::QueryInterface() failed",  
hr);  
}  
  
IDXGIAdapter *dxgiAdapter = nullptr;  
if (FAILED(hr = dxgiDevice->GetParent(__uuidof(IDXGIAdapter),  
reinterpret_cast<void**>(&dxgiAdapter))))  
{  
    ReleaseObject(dxgiDevice);  
    throw GameException("IDXGIDevice::GetParent() failed  
retrieving  
adapter.", hr);  
}  
  
IDXGIFactory2* dxgiFactory = nullptr;  
if (FAILED(hr = dxgiAdapter->GetParent(__uuidof(IDXGIFactory2),  
reinterpret_cast<void**>(&dxgiFactory))))  
{  
    ReleaseObject(dxgiDevice);  
    ReleaseObject(dxgiAdapter);  
    throw GameException("IDXGIAdapter::GetParent() failed  
retrieving  
factory.", hr);  
}  
  
IDXGISwapChain1* mSwapChain;  
if (FAILED(hr = dxgiFactory->CreateSwapChainForHwnd(dxgiDevice,  
mWindowHandle, &swapChainDesc, &fullScreenDesc, nullptr,  
&mSwapChain)))  
{  
    ReleaseObject(dxgiDevice);  
    ReleaseObject(dxgiAdapter);  
    ReleaseObject(dxgiFactory);  
    throw GameException("IDXGIDevice::CreateSwapChainForHwnd()  
failed.", hr);  
}
```

```
ReleaseObject(dxgiDevice);
ReleaseObject(dxgiAdapter);
ReleaseObject(dxgiFactory);
```

---

## Creating a Render Target View

When you created the swap chain, you established a back buffer, a texture to render to. The intent is to bind that texture to the output-merger stage of the Direct3D pipeline. However, such resources aren't bound directly to a pipeline stage; instead, you create a **resource view** of the texture and bind the view to the pipeline. You create a render target view with a call to `ID3D11Device::CreateRenderTargetView()`; its prototype and parameters are listed next:

[Click here to view code image](#)

```
HRESULT CreateRenderTargetView(
    ID3D11Resource *pResource,
    const D3D11_RENDER_TARGET_VIEW_DESC *pDesc,
    ID3D11RenderTargetView **ppRTView);
```

- **pResource:** The render target resource.
- **pDesc:** A render target view description structure that allows explicit definition of the render target view. Set this parameter to `NULL` to create a view that accesses all subresources at mip-level 0.
- **ppRTView:** The created render target view.

You can query the texture resource for the `CreateRenderTargetView()` function from the swap chain with a call to `IDXGISwapChain::GetBuffer()`. [Listing 11.5](#) presents the code for querying the swap chain and creating a render target view.

### **Listing 11.5** Creating a Render Target View

[Click here to view code image](#)

```
ID3D11Texture2D* backBuffer;
if (FAILED(hr = mSwapChain->GetBuffer(0,
    __uuidof(ID3D11Texture2D),
    reinterpret_cast<void**>(&backBuffer)))
{
    throw GameException("IDXGISwapChain::GetBuffer() failed.",
    hr);
}

if (FAILED(hr = mDirect3DDevice-
>CreateRenderTargetView(backBuffer,
nullptr, &mRenderTargetView)))
{
    ReleaseObject(backBuffer);
```

```
    throw GameException("IDXGIDevice::CreateRenderTargetView()  
failed.", hr);  
}  
  
ReleaseObject(backBuffer);
```

---

## Creating a Depth-Stencil View

Objects in a scene can overlap. An object nearer the camera can partially or completely occlude an object that is farther away. When a pixel is written to the render target, its depth (its distance from the camera) can be written to a **depth buffer**. The depth buffer is just a 2D texture, but instead of storing colors, it stores depths. The output-merger stage can use these depths to determine whether new pixels should overwrite existing colors already present in the render target. This process is known as depth testing; we discussed it briefly back in [Chapter 1, “Introducing DirectX”](#).

Stencil testing uses a mask to determine which pixels to update. This is conceptually similar to a cardboard or plastic stencil you might use to paint or print a design on a physical surface. The depth and stencil buffers are attached, hence the terms *depth-stencil buffer* and *depth-stencil view*.

## Populating a 2D Texture Description

When you created the swap chain, you implicitly created the associated 2D texture for the back buffer, but a depth buffer was not created. You build a 2D texture for the depth buffer through a call to `ID3D11Device::CreateTexture2D()`. This method follows the description structure pattern you used when creating the swap chain and requires you to populate an instance of the `D3D11_TEXTURE2D_DESC` structure. The members of this structure are presented and described next:

[Click here to view code image](#)

```
typedef struct D3D11_TEXTURE2D_DESC  
{  
    UINT Width;  
    UINT Height;  
    UINT MipLevels;  
    UINT ArraySize;  
    DXGI_FORMAT Format;  
    DXGI_SAMPLE_DESC SampleDesc;  
    D3D11_USAGE Usage;  
    UINT BindFlags;  
    UINT CPUAccessFlags;  
    UINT MiscFlags;  
} D3D11_TEXTURE2D_DESC;
```

- **Width:** The texture width.
- **Height:** The texture height.
- **MipLevels:** The number of mip-levels in the texture.
- **ArraySize:** The number of textures in a texture array.

- **Format:** The texture format. Unlike the back buffer format, which stores color, the depth buffer stores depth and stencil data. Common formats include `DXGI_FORMAT_D24_UNORM_S8_UINT` (a 24-bit depth buffer and an 8-bit stencil buffer) and `DXGI_FORMAT_D32_FLOAT` (all 32 bits are for the depth buffer).
- **SampleDesc:** Multisampling parameters.
- **Usage:** How the texture will be read from and written to. Most commonly set to `D3D11_USAGE_DEFAULT`.
- **BindFlags:** `D3D11_BIND_DEPTH_STENCIL` for a depth-stencil buffer.
- **CPUAccessFlags:** Bitwise OR'd values that specify whether the CPU is allowed to read or write to the texture. Use 0 when no CPU access is required.
- **MiscFlags:** Flags for less commonly used resource options. Not used for the depth-stencil buffer.

[Listing 11.6](#) shows an example of populating an instance of the `D3D11_TEXTURE2D_DESC` structure.

## Listing 11.6 Populating a 2D Texture Description Structure

[Click here to view code image](#)

```
D3D11_TEXTURE2D_DESC depthStencilDesc;
ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));
depthStencilDesc.Width = mScreenWidth;
depthStencilDesc.Height = mScreenHeight;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.ArraySize = 1;
depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthStencilDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;

if (mMultiSamplingEnabled)
{
    depthStencilDesc.SampleDesc.Count = mMultiSamplingCount;
    depthStencilDesc.SampleDesc.Quality =
mMultiSamplingQualityLevels - 1;
}
else
{
    depthStencilDesc.SampleDesc.Count = 1;
    depthStencilDesc.SampleDesc.Quality = 0;
}
```

## Creating a 2D Texture and Depth-Stencil View

You use the `D3D11_TEXTURE2D_DESC` structure instance with a call to `ID3D11Device::CreateTexture2D()`, whose prototype and parameters are listed next:

[Click here to view code image](#)

```
HRESULT CreateTexture2D(
    const D3D11_TEXTURE2D_DESC *pDesc,
    const D3D11_SUBRESOURCE_DATA *pInitialData,
    ID3D11Texture2D **ppTexture2D);
```

- **pDesc:** The 2D texture description structure.
- **pInitialData:** Initial data used to populate the texture. No initial data is required for the depth buffer, so this parameter is set to NULL.
- **ppTexture2D:** Returns the created 2D texture.

Finally, the depth-stencil view is created with a call to

`ID3D11Device::CreateDepthStencilView()`. This method has a set of parameters analogous to those of `CreateRenderTargetView()`. [Listing 11.7](#) presents example code for creating a depth buffer and depth-stencil view.

### **Listing 11.7 Creating a Depth Buffer and Depth-Stencil View**

[Click here to view code image](#)

```
ID3D11Texture2D* mDepthStencilBuffer;
ID3D11DepthStencilView* mDepthStencilView;

if (FAILED(hr = mDirect3DDevice-
>CreateTexture2D(&depthStencilDesc,
nullptr, &mDepthStencilBuffer)))
{
    throw GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
}

if (FAILED(hr = mDirect3DDevice-
>CreateDepthStencilView(mDepthStencil
Buffer, nullptr, &mDepthStencilView)))
{
    throw GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
}
```

### **Associating the Views to the Output-Merger Stage**

With the render target view and the depth-stencil view created, you can now bind them to the output-merger stage of the Direct3D pipeline with a call to `ID3D11DeviceContext::OMSetRenderTargets()`. The prototype and parameters of this method are listed next:

[Click here to view code image](#)

```
void OMSetRenderTargets(
```

```
UINT NumViews,  
ID3D11RenderTargetView *const *ppRenderTargetViews,  
ID3D11DepthStencilView *pDepthStencilView);
```

- **NumViews:** The number of render targets to bind to the pipeline. [Part IV, “Intermediate-Level Rendering Topics,”](#) covers using multiple render targets. For now, you specify 1 to bind just a single render target.
- **ppRenderTargetViews:** The (input) array of render targets to bind to the pipeline (must have the same number of elements specified in **NumViews**).
- **pDepthStencilView:** The depth-stencil view to bind to the pipeline.

[Listing 11.8](#) presents the code for binding the views to the output-merger stage.

## Listing 11.8 Binding the Views to the Output-Merger Stage

[Click here to view code image](#)

---

```
mDirect3DDeviceContext->OMSetRenderTargets(1,  
&mRenderTargetView,  
mDepthStencilView);
```

---

## Setting the Viewport

The final step of Direct3D initialization is to set a **viewport**. A viewport is a rectangular area that can encompass the entire back buffer, or some portion of it; your scenes are rendered to this area. Viewports are commonly used for split-screen multiplayer games, where different views of the game world are displayed to separate areas of the screen. You set the viewport by populating an instance of the `D3D11_VIEWPORT` structure and calling `ID3D11DeviceContext::RSSetViewports()`. The members of the `D3D11_VIEWPORT` structure are described here:

[Click here to view code image](#)

```
typedef struct D3D11_VIEWPORT  
{  
    FLOAT TopLeftX;  
    FLOAT TopLeftY;  
    FLOAT Width;  
    FLOAT Height;  
    FLOAT MinDepth;  
    FLOAT MaxDepth;  
} D3D11_VIEWPORT;
```

- **TopLeftX, TopLeftY, Width, Height:** Members that define the area of the viewport.
- **MinDepth, MaxDepth:** The minimum and maximum values for the depth buffer. These values are typically set to 0.0, and 1.0, respectively.

The `RSSetViewports()` method has just two parameters: the number of viewports to bind and an array of viewports. [Listing 11.9](#) shows an example of creating a viewport structure and binding it to

the pipeline.

## Listing 11.9 Setting the Viewport

[Click here to view code image](#)

---

```
D3D11_VIEWPORT mViewport;
mViewport.TopLeftX = 0.0f;
mViewport.TopLeftY = 0.0f;
mViewport.Width = static_cast<float>(mScreenWidth);
mViewport.Height = static_cast<float>(mScreenHeight);
mViewport.MinDepth = 0.0f;
mViewport.MaxDepth = 1.0f;

mDirect3DDeviceContext->RSSetViewports(1, &mViewport);
```

---

## Putting It All Together

With the specifics of Direct3D initialization in hand, you are ready to create your first Direct3D application. This entails augmenting the general-purpose Game class to incorporate the Direct3D initialization phases, creating a specialized Game class, and modifying the `WinMain()` function to use the specialized game class. The next few sections describe these steps.

## Updated General-Purpose Game Class

[Listing 11.10](#) presents an abbreviated version of the `Game.h` file. This code builds upon the window initialization code from [Chapter 10, “Project Setup and Window Initialization.”](#) Visit the book’s companion website for the full source code.

## Listing 11.10 An Abbreviated Game.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"
#include "GameClock.h"
#include "GameTime.h"

namespace Library
{
    class Game
    {
        public:
            /* ... Previously presented members removed for brevity
... */
```

ID3D11Device1\* Direct3DDevice() const;

```

ID3D11DeviceContext1* Direct3DDeviceContext() const;
bool DepthBufferEnabled() const;
bool IsFullScreen() const;
const D3D11_TEXTURE2D_DESC& BackBufferDesc() const;
const D3D11_VIEWPORT& Viewport() const;

protected:
    /* ... Previously presented members removed for brevity
...
}

virtual void InitializeDirectX();

static const UINT DefaultFrameRate;
static const UINT DefaultMultiSamplingCount;

D3D_FEATURE_LEVEL mFeatureLevel;
ID3D11Device1* mDirect3DDevice;
ID3D11DeviceContext1* mDirect3DDeviceContext;
IDXGISwapChain1* mSwapChain;

UINT mFrameRate;
bool mIsFullScreen;
bool mDepthStencilBufferEnabled;
bool mMultiSamplingEnabled;
UINT mMultiSamplingCount;
UINT mMultiSamplingQualityLevels;

ID3D11Texture2D* mDepthStencilBuffer;
D3D11_TEXTURE2D_DESC mBackBufferDesc;
ID3D11RenderTargetView* mRenderTargetView;
ID3D11DepthStencilView* mDepthStencilView;
D3D11_VIEWPORT mViewport;
};

}

```

As this code demonstrates, the Game class now stores members for the Direct3D device and device context, multisampling settings, the depth-stencil buffer and view, the render target view, and the viewport. A few of these members are exposed through public accessors, and Direct3D initialization is performed through the `InitializeDirectX()` method. Before presenting the implementation of these new Game members, notice the include directive for `Common.h`. This is a header file for commonly needed declarations (see [Listing 11.11](#)).

## **Listing 11.11** The Common.h Header File

[Click here to view code image](#)

---

```
#pragma once
```

```

#include <windows.h>
#include <exception>
#include <cassert>
#include <string>
#include <vector>
#include <map>
#include <memory>

#include <d3d11_1.h>
#include <DirectXMath.h>
#include <DirectXPackedVector.h>

#define DeleteObject(object) if((object) != NULL) { delete object;
object = NULL; }
#define DeleteObjects(objects) if((objects) != NULL) { delete [] objects; objects = NULL; }
#define ReleaseObject(object) if((object) != NULL) { object->Release(); object = NULL; }

namespace Library
{
    typedef unsigned char byte;
}

using namespace DirectX;
using namespace DirectX::PackedVector;

```

---

The d3d11\_1.h, DirectXMath.h and DirectXPackedVector.h files include declarations for Direct3D 11.1 and DirectXMath. The DirectXMath types are part of the DirectX and DirectX::PackedVector namespaces, hence their inclusion.

[Listing 11.12](#) presents the updated Game class implementation. Again, this is an abbreviated listing of the file that omits the previously presented material and implementations of pass-through accessors.

### Listing 11.12 An Abbreviated Game.cpp File

[Click here to view code image](#)

---

```

#include "Game.h"
#include "GameException.h"

namespace Library
{

```

```
const UINT Game::DefaultScreenWidth = 1024;
const UINT Game::DefaultScreenHeight = 768;
const UINT Game::DefaultFrameRate = 60;
const UINT Game::DefaultMultiSamplingCount = 4;

Game::Game(HINSTANCE instance, const std::wstring& windowClass,
           const std::wstring& windowTitle, int showCommand)
    : mInstance(instance), mWindowClass(windowClass),
      mWindowTitle(windowTitle), mShowCommand(showCommand),
      mWindowHandle(), mWindow(),
      mScreenWidth(DefaultScreenWidth),
      mScreenHeight(DefaultScreenHeight),
      mGameClock(), mGameTime(),
      mFeatureLevel(D3D_FEATURE_LEVEL_9_1),
      mDirect3DDevice(nullptr),
      mDirect3DDeviceContext(nullptr), mSwapChain(nullptr),
      mFrameRate(DefaultFrameRate), mIsFullScreen(false),
      mDepthStencilBufferEnabled(false),
      mMultiSamplingEnabled(false),
      mMultiSamplingCount(DefaultMultiSamplingCount),
      mMultiSamplingQualityLevels(0),
      mDepthStencilBuffer(nullptr),
      mRenderTargetView(nullptr),
      mDepthStencilView(nullptr), mViewPort()
{
}

void Game::Run()
{
    InitializeWindow();
    InitializeDirectX();
    Initialize();

    MSG message;
    ZeroMemory(&message, sizeof(message));

    mGameClock.Reset();

    while (message.message != WM_QUIT)
    {
        if (PeekMessage(&message, nullptr, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&message);
            DispatchMessage(&message);
        }
    }
}
```

```
        else
    {
        mGameClock.UpdateGameTime(mGameTime);
        Update(mGameTime);
        Draw(mGameTime);
    }

    Shutdown();
}

void Game::Shutdown()
{
    ReleaseObject(mRenderTargetView);
    ReleaseObject(mDepthStencilView);
    ReleaseObject(mSwapChain);
    ReleaseObject(mDepthStencilBuffer);

    if (mDirect3DDeviceContext != nullptr)
    {
        mDirect3DDeviceContext->ClearState();
    }

    ReleaseObject(mDirect3DDeviceContext);
    ReleaseObject(mDirect3DDevice);

    UnregisterClass(mWindowClass.c_str(),
mWindow.hInstance);
}

void Game::InitializeDirectX()
{
    HRESULT hr;
    UINT createDeviceFlags = 0;

#if defined(DEBUG) || defined(_DEBUG)
    createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

    D3D_FEATURE_LEVEL featureLevels[] = {
        D3D_FEATURE_LEVEL_11_0,
        D3D_FEATURE_LEVEL_10_1,
        D3D_FEATURE_LEVEL_10_0
    };

    // Step 1: Create the Direct3D device and device context
    interfaces
```

```

ID3D11Device* direct3DDevice = nullptr;
ID3D11DeviceContext* direct3DDeviceContext = nullptr;
if (FAILED(hr = D3D11CreateDevice(NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, createDeviceFlags, featureLevels,
ARRAYSIZE(featureLevels), D3D11_SDK_VERSION, &direct3DDevice,
&mFeatureLevel, &direct3DDeviceContext)))
{
    throw GameException("D3D11CreateDevice() failed",
hr);
}

if (FAILED(hr = direct3DDevice->QueryInterface(__uuidof(ID3D11Device1), reinterpret_cast<void**>(&mDirect3DDevice))))
{
    throw GameException("ID3D11Device::QueryInterface() failed",
hr);
}

if (FAILED(hr = direct3DDeviceContext->QueryInterface(__uuidof(ID3D11DeviceContext1), reinterpret_cast<void**>(&mDirect3DDeviceContext))))
{
    throw GameException("ID3D11Device::QueryInterface() failed",
hr);
}

ReleaseObject(direct3DDevice);
ReleaseObject(direct3DDeviceContext);

// Step 2: Check for multisampling support
mDirect3DDevice-
>CheckMultisampleQualityLevels(DXGI_FORMAT_R8G8B8A8_UNORM, mMultiSamplingCount,
&mMultiSamplingQualityLevels);
if (mMultiSamplingQualityLevels == 0)
{
    throw GameException("Unsupported multi-sampling
quality");
}

DXGI_SWAP_CHAIN_DESC1 swapChainDesc;
ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));
swapChainDesc.Width = mScreenWidth;
swapChainDesc.Height = mScreenHeight;
swapChainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;

```

```

if (mMultiSamplingEnabled)
{
    swapChainDesc.SampleDesc.Count =
mMultiSamplingCount;
    swapChainDesc.SampleDesc.Quality =
mMultiSamplingQualityLevels - 1;
}
else
{
    swapChainDesc.SampleDesc.Count = 1;
    swapChainDesc.SampleDesc.Quality = 0;
}

swapChainDesc.BufferUsage =
DXGI_USAGE_RENDER_TARGET_OUTPUT;
swapChainDesc.BufferCount = 1;
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;

// Step 3: Create the swap chain
IDXGIDevice* dxgiDevice = nullptr;
if (FAILED(hr = mDirect3DDevice->QueryInterface(
__uuidof(IDXGIDevice), reinterpret_cast<void**>(&dxgiDevice))))
{
    throw GameException("ID3D11Device::QueryInterface() failed",
hr);
}

IDXGIAdapter *dxgiAdapter = nullptr;
if (FAILED(hr = dxgiDevice-
>GetParent(__uuidof(IDXGIAdapter),
reinterpret_cast<void**>(&dxgiAdapter))))
{
    ReleaseObject(dxgiDevice);
    throw GameException("IDXGIDevice::GetParent() failed
retrieving adapter.", hr);
}

IDXGIFactory2* dxgiFactory = nullptr;
if (FAILED(hr = dxgiAdapter-
>GetParent(__uuidof(IDXGIFactory2),
reinterpret_cast<void**>(&dxgiFactory))))
{
    ReleaseObject(dxgiDevice);
    ReleaseObject(dxgiAdapter);
    throw GameException("IDXGIAdapter::GetParent() failed
retrieving factory.", hr);
}

```

```
}

DXGI_SWAP_CHAIN_FULLSCREEN_DESC fullScreenDesc;
ZeroMemory(&fullScreenDesc, sizeof(fullScreenDesc));
fullScreenDesc.RefreshRate.Numerator = mFrameRate;
fullScreenDesc.RefreshRate.Denominator = 1;
fullScreenDesc.Windowed = !mIsFullScreen;

    if (FAILED(hr = dxgiFactory-
>CreateSwapChainForHwnd(dxgiDevice,
mWindowHandle, &swapChainDesc, &fullScreenDesc, nullptr,
&mSwapChain)))
{
    ReleaseObject(dxgiDevice);
    ReleaseObject(dxgiAdapter);
    ReleaseObject(dxgiFactory);
    throw
GameException("IDXGIDevice::CreateSwapChainForHwnd()
failed.", hr);
}

ReleaseObject(dxgiDevice);
ReleaseObject(dxgiAdapter);
ReleaseObject(dxgiFactory);

// Step 4: Create the render target view
ID3D11Texture2D* backBuffer;
if (FAILED(hr = mSwapChain->GetBuffer(0, __
uuidof(ID3D11Texture2D), reinterpret_cast<void**>(
&backBuffer))))
{
    throw GameException("IDXGISwapChain::GetBuffer()
failed.", hr);
}

backBuffer->GetDesc(&mBackBufferDesc);

    if (FAILED(hr = mDirect3DDevice->CreateRenderTargetView(
backBuffer, nullptr, &mRenderTargetView) ))
{
    ReleaseObject(backBuffer);
    throw
GameException("IDXGIDevice::CreateRenderTargetView()
failed.", hr);
}

ReleaseObject(backBuffer);
```

```

// Step 5: Create the depth-stencil view
if (mDepthStencilBufferEnabled)
{
    D3D11_TEXTURE2D_DESC depthStencilDesc;
    ZeroMemory(&depthStencilDesc,
sizeof(depthStencilDesc));
    depthStencilDesc.Width = mScreenWidth;
    depthStencilDesc.Height = mScreenHeight;
    depthStencilDesc.MipLevels = 1;
    depthStencilDesc.ArraySize = 1;
    depthStencilDesc.Format =
DXGI_FORMAT_D24_UNORM_S8_UINT;
    depthStencilDesc.BindFlags =
D3D11_BIND_DEPTH_STENCIL;
    depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;

    if (mMultiSamplingEnabled)
    {
        depthStencilDesc.SampleDesc.Count =
mMultiSamplingCount;
        depthStencilDesc.SampleDesc.Quality =
mMultiSamplingQualityLevels - 1;
    }
    else
    {
        depthStencilDesc.SampleDesc.Count = 1;
        depthStencilDesc.SampleDesc.Quality = 0;
    }

    if (FAILED(hr = mDirect3DDevice->CreateTexture2D
(&depthStencilDesc, nullptr, &mDepthStencilBuffer)))
    {
        throw
GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
    }

    if (FAILED(hr = mDirect3DDevice-
>CreateDepthStencilView
(mDepthStencilBuffer, nullptr, &mDepthStencilView)))
    {
        throw
GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
    }
}

```

```

    // Step 6: Bind the render target and depth-stencil
views to OM
stage
    mDirect3DDeviceContext->OMSetRenderTargets(1,
&mRenderTargetView, mDepthStencilView);

    mViewport.TopLeftX = 0.0f;
    mViewport.TopLeftY = 0.0f;
    mViewport.Width = static_cast<float>(mScreenWidth);
    mViewport.Height = static_cast<float>(mScreenHeight);
    mViewport.MinDepth = 0.0f;
    mViewport.MaxDepth = 1.0f;

    // Step 7: Set the viewport
    mDirect3DDeviceContext->RSSetViewports(1, &mViewport);
}

```

---

The new code in [Listing 11.12](#) includes the Game class constructor and the Shutdown () and InitializeDirectX () methods. The constructor simply initializes all the class members; the Shutdown () method releases Direct3D objects instantiated within InitializeDirectX (). The Run () method has been updated to call the DirectX initialization method. Recognize that the InitializeDirectX (), Shutdown (), and Run () methods are marked virtual and can therefore be overridden in a derived class. The presented implementations cover general-purpose uses of the Game class.

## Specialized Game Class

For all the demo projects, you'll create a specialized version of the Game class that resides within the executable project of your Visual Studio solution. Generally, this derived class overrides the general-purpose Initialize () method and the Update () and Draw () methods invoked by the base class Run () implementation. The base Game class implementations of these three methods are currently empty. In the next chapter, we discuss the concept of **game components**, a modular approach for adding functionality to your applications. These methods will initialize, update, and draw the collection of active components.

[Listing 11.13](#) presents the header file for the RenderingGame class, which derives from the Library::Game class. Create this class within your Game project, not the Library project, which houses the majority of your code thus far.

### Listing 11.13 The RenderingGame.h File

[Click here to view code image](#)

---

```

#pragma once

#include "Common.h"
#include "Game.h"

```

```
using namespace Library;

namespace Rendering
{
    class RenderingGame : public Game
    {
        public:
            RenderingGame(HINSTANCE instance, const std::wstring& windowClass, const std::wstring& windowTitle, int showCommand);
            ~RenderingGame();

            virtual void Initialize() override;
            virtual void Update(const GameTime& gameTime) override;
            virtual void Draw(const GameTime& gameTime) override;

        private:
            static const XMVECTORF32 BackgroundColor;
    };
}
```

---

The only item to point out from this header file is the `BackgroundColor` member of type `XMVECTORF32`. This is a SIMD DirectXMath type that allows initializer syntax. [Listing 11.14](#) shows the implementation of the `RenderingGame` class.

### **Listing 11.14** The `RenderingGame.cpp` File

[Click here to view code image](#)

---

```
#include "RenderingGame.h"
#include "GameException.h"

namespace Rendering
{
    const XMVECTORF32 RenderingGame::BackgroundColor = { 0.392f,
0.584f, 0.929f, 1.0f };

    RenderingGame::RenderingGame(HINSTANCE instance, const std::wstring& windowClass, const std::wstring& windowTitle, int showCommand)
        : Game(instance, windowClass, windowTitle, showCommand)
    {
        mDepthStencilBufferEnabled = true;
        mMutiSamplingEnabled = true;
    }

    RenderingGame::~RenderingGame()
```

```

{
}

void RenderingGame::Initialize()
{
    Game::Initialize();
}

void RenderingGame::Update(const GameTime &gameTime)
{
    Game::Update(gameTime);
}

void RenderingGame::Draw(const GameTime &gameTime)
{
    mDirect3DDeviceContext-
>ClearRenderTargetView(mRenderTarget
View, reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext-
>ClearDepthStencilView(mDepthStencil
View, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    Game::Draw(gameTime);

    HRESULT hr = mSwapChain->Present(0, 0);
    if (FAILED(hr))
    {
        throw GameException("IDXGISwapChain::Present()
failed.", hr);
    }
}

```

This class renders a blank screen in cornflower blue (or whatever color values you specify for the `BackgroundColor` member). The code shows the general structure of the `Game`-derived class, although you'll actually extend the `Initialize()` and `Update()` methods in more interesting applications.

In the constructor of the `RenderingGame` class, you enable the use of a depth-stencil buffer and multisampling (although you aren't specifically using them in this example). In the `Draw()` method, you clear the render target with a call to `ID3D11DeviceContext1::ClearRenderTargetView()`. This method sets the entire render target to the specified color. Next, you clear the depth-stencil buffer with a call to `ID3D11DeviceContext1::ClearDepthStencilView()`. The second parameter of this method is one or more `D3D11_CLEAR_FLAG` options, bitwise OR'd together to specify which part of the depth-stencil buffer to clear. The third parameter specifies the clear value for the depth buffer (typically 1.0), and the fourth parameter is the clear value for the stencil buffer (typically 0).

After you clear the render and depth-stencil views, you perform any game-specific rendering calls. In [Listing 11.14](#), the call to `Game::Draw()` is intended to render any visible game components. Again, the next chapter covers the topic of game components. When all game-specific rendering has been completed, you flip the buffers in the swap chain with a call to `IDXGISwapChain1::Present()`. The first parameter in the `Present()` method specifies how the frame is presented with respect to the monitor's vertical refresh. A value of 0 indicates that presentation should occur immediately, with no vertical refresh synchronization. The second parameter is a set of bitwise OR'd `DXGI_PRESENT` flags. A value of 0 simply presents the frame to the output without any special options. You can find details on more advanced vertical synchronization and presentation settings in the online documentation.

## Updated WinMain()

The last step before running your application is to modify the `WinMain()` function to use the new `RenderingGame` class. The changes are simple; you include `RenderingGame.h` instead of `Game.h` and update the `game` variable instantiation to the corresponding class. [Listing 11.15](#) presents the code for the updated `WinMain()` from `Program.cpp`.

### **Listing 11.15** The `Program.cpp` File

[Click here to view code image](#)

```
#include <memory>
#include "GameException.h"
#include "RenderingGame.h"

#if defined(DEBUG) || defined(_DEBUG)
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtDBG.h>
#endif

using namespace Library;
using namespace Rendering;

int WINAPI WinMain(HINSTANCE instance, HINSTANCE
previousInstance,
LPSTR commandLine, int showCommand)
{
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF |
    _CRTDBG_LEAK_CHECK_DF);
#endif

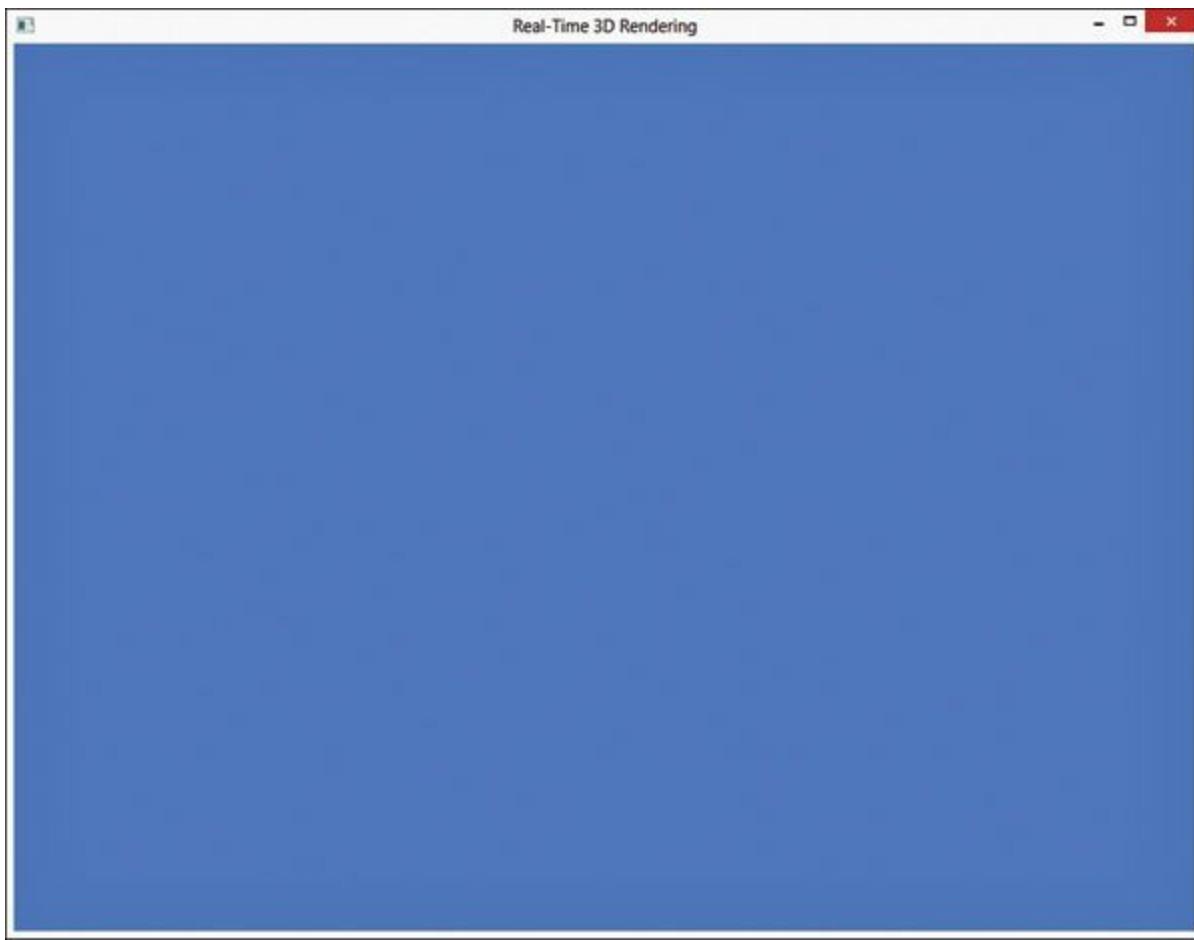
    std::unique_ptr<RenderingGame> game(new
RenderingGame(instance,
L"RenderingClass", L"Real-Time 3D Rendering", showCommand));
}
```

```
try
{
    game->Run();
}
catch (GameException ex)
{
    MessageBox(game->WindowHandle(), ex.whatw().c_str(),
game->WindowTitle().c_str(), MB_ABORTRETRYIGNORE);
}

return 0;
}
```

---

Build and run your application; you should see the image in [Figure 11.2](#).



**Figure 11.2** A solid-color window rendered with Direct3D using the demo framework.

Again, the results of your efforts are a bit anticlimactic, but most of the preliminary work is now complete and you've set the stage for rendering with Direct3D.

## Summary

In this chapter, you learned how to initialize Direct3D. You took a first look at the Direct3D C++ API and finalized the foundation of your rendering engine. Additional supporting systems are left to develop, but you have successfully completed your first Direct3D application.

In the next chapter, you create interfaces for game components, develop mouse and keyboard input systems, build a dynamic 3D camera, and learn how to render 2D text to the screen.

## Exercise

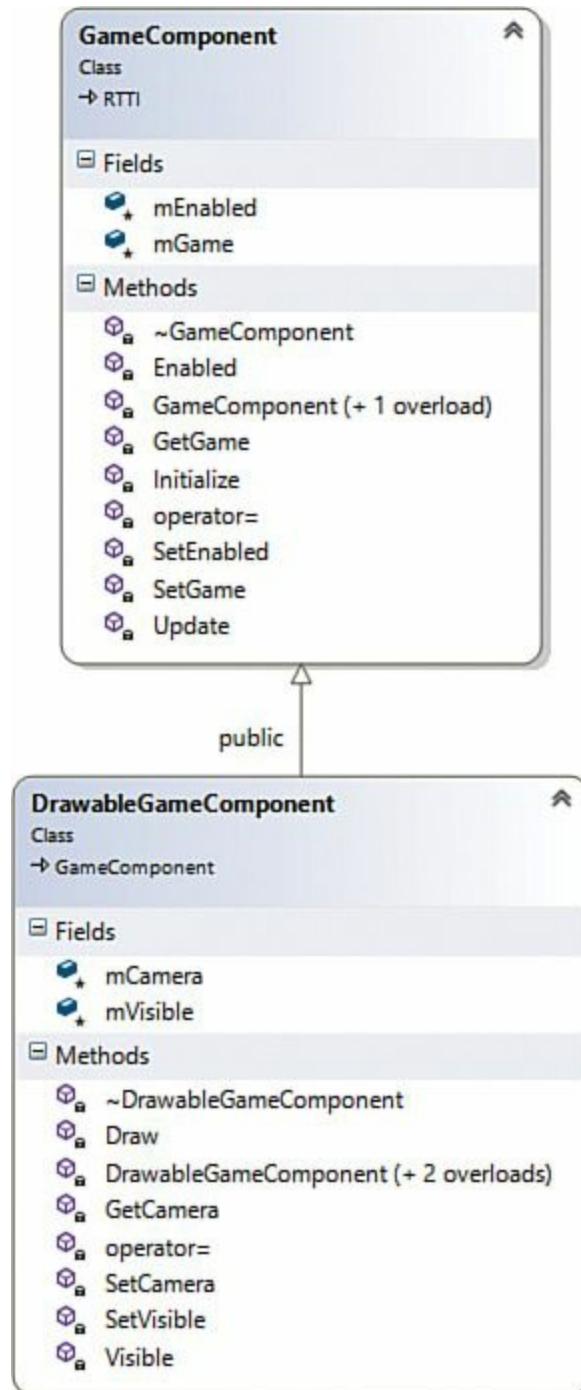
1. From within the debugger, walk through the code used to initialize Direct3D, to better understand the initialization process. You'll find an associated demo application on the book's companion website.

# Chapter 12. Supporting Systems

This chapter takes a brief detour from 3D rendering to create some scaffolding to support the rendering engine. You implement classes for keyboard and mouse input, and you establish a reusable component system for adding functionality to your applications. You also learn about text rendering and create a service container for housing commonly accessed software modules.

## Game Components

Game components provide a modular approach to adding functionality to your applications and are supported through two classes, `GameComponent` and `DrawableGameComponent`. [Figure 12.1](#) show their class diagrams.



**Figure 12.1** Class diagrams for the `GameComponent` and `DrawableGameComponent` classes.

You create a game component by deriving from one of these classes—from `GameComponent` if your functionality requires no rendering, and from `DrawableGameComponent` otherwise. All game components have `Initialize()` and `Update()` methods, while drawable game components include a `Draw()` method. These methods are typically invoked by the general-purpose `Game` class methods of the same names. The `Game` class stores a `std::vector` of game components and you register new components by pushing them onto this vector.

[Listings 12.1](#) and [12.2](#) present the header file and implementation for the `GameComponent` class.

### **Listing 12.1** The `GameComponent.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;
    class GameTime;

    class GameComponent : public RTTI
    {
        RTTI_DECLARATIONS(GameComponent, RTTI)

    public:
        GameComponent();
        GameComponent(Game& game);
        virtual ~GameComponent();

        Game* GetGame();
        void SetGame(Game& game);
        bool Enabled() const;
        void SetEnabled(bool enabled);

        virtual void Initialize();
        virtual void Update(const GameTime& gameTime);

    protected:
        Game* mGame;
        bool mEnabled;

    private:
        GameComponent(const GameComponent& rhs);
        GameComponent& operator=(const GameComponent& rhs);
    };
}
```

}

---

## Listing 12.2 The GameComponent.cpp File

[Click here to view code image](#)

---

```
#include "GameComponent.h"
#include "GameTime.h"

namespace Library
{
    RTTI_DEFINITIONS(GameComponent)

    GameComponent::GameComponent()
        : mGame(nullptr), mEnabled(true)
    {

    }

    GameComponent::GameComponent(Game& game)
        : mGame(&game), mEnabled(true)
    {
    }

    GameComponent::~GameComponent()
    {

    }

    Game* GameComponent::GetGame()
    {
        return mGame;
    }

    void GameComponent::SetGame(Game& game)
    {
        mGame = &game;
    }

    bool GameComponent::Enabled() const
    {
        return mEnabled;
    }

    void GameComponent::SetEnabled(bool enabled)
    {
        mEnabled = enabled;
    }
}
```

```
void GameComponent::Initialize()
{
}

void GameComponent::Update(const GameTime& gameTime)
{
}

}
```

---

As you can see, this is a fairly simple class. It stores a pointer to the associated Game instance and a flag denoting the enabled/disabled status of the component. And although the `Initialize()` and `Update()` methods aren't purely virtual, their implementations are empty. Your derived classes are intended to override these methods but aren't explicitly required to do so. This is useful for drawable game components, which are specialized game components that need to draw but not update, or for components that require no initialization.

## Custom Runtime Type Information

From the two code listings and the class diagrams, you might have noticed the references to the type RTTI. This is a custom implementation of **Runtime Type Information** (RTTI). RTTI refers to type introspection, the program's capability to examine a type at runtime. At a minimum, this allows a program to identify the specific data type of an object, although the object can be referenced through a generic pointer. Some languages also include the capability to query properties of an interface. Do not confuse RTTI with **reflection**, which provides the capability to query *and* manipulate members of an object.

C++ supports RTTI through the `typeid` operator, the `type_info` class, and the `dynamic_cast` operator. However, when facilitated at the language level, RTTI applies either to all polymorphic classes or to none; you cannot select which classes to generate type information for. Furthermore, the expense of RTTI is implementation specific. Its cost might be acceptable, or even negligible, on one platform and not on another. You can remove any doubt by disabling language-level RTTI support and writing your own RTTI implementation. That's the approach taken here.

To disable RTTI within Visual Studio, open the Property Pages of your projects and navigate to **Configuration Properties, C/C++, Language**. Set the **Enable Run-Time Type Information** field to **No (/GR-)**. [Listing 12.3](#) presents the code for a custom RTTI implementation.

### Listing 12.3 The RTTI.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include <string>

namespace Library
{
```

```
class RTTI
{
public:
    virtual const unsigned int& TypeIdInstance() const = 0;

    virtual RTTI* QueryInterface(const unsigned id) const
    {
        return nullptr;
    }

    virtual bool Is(const unsigned int id) const
    {
        return false;
    }

    virtual bool Is(const std::string& name) const
    {
        return false;
    }

    template <typename T>
    T* As() const
    {
        if (Is(T::TypeIdClass()))
        {
            return (T*)this;
        }

        return nullptr;
    }
};

#define RTTI_DECLARATIONS(Type,
ParentType) \
public: \
    typedef ParentType \
Parent; \
    static std::string TypeName() { return std::string \
(#Type); \
} \
    virtual const unsigned int& TypeIdInstance() const { \
return \
Type::TypeIdClass(); \
} \
    static const unsigned int& TypeIdClass() { return \
sRunTimeTypeId;
}
```

```

        virtual Library::RTTI* QueryInterface( const unsigned
int
id ) const\

    {
        if (id ==
sRuntimeTypeId)
            \
                { return (RTTI*)this;
}
            \
        else
            { return Parent::QueryInterface(id);
}
            \
    }

virtual bool Is(const unsigned int id)
const
{
    if (id ==
sRuntimeTypeId)
        \
            { return true;
}
            \
    else
            { return Parent::Is(id);
}
            \
}

virtual bool Is(const std::string& name)
const
{
    if (name ==
TypeName())
        \
            { return true;
}
            \
    else
            { return Parent::Is(name);
}
            \
}

private:
    static unsigned int sRuntimeTypeId;

#define RTTI_DEFINITIONS(Type) unsigned int
Type::sRuntimeTypeId =
(unsigned int)& Type::sRuntimeTypeId;
}

```

As you can see, the entire implementation of the RTTI class is contained within the header file. Be sure to include this header file within Common.h. To use the interface, derive a class from the RTTI type and include the RTTI\_DECLARATIONS macro somewhere within the class declaration. The first argument of the RTTI\_DECLARATIONS macro is the derived type, and the second argument is its parent (this implementation provides no explicit support for multiple inheritance). Use the

RTTI\_DEFINITIONS macro in the class implementation. That macro's only argument is the associated type. [Listings 12.1](#) and [12.2](#) demonstrate this usage.

When a type is enabled with this RTTI system, it can be queried through the Is(), As(), and QueryInterface() methods. A type's identification is stored as an unsigned integer and is guaranteed to be unique because it's assigned as the address of the static sRunTimeTypeId member. This identifier is exposed through the TypeIdClass() and TypeIdInstance() methods, which can be used as arguments to the Is() and QueryInterface() methods. The name of the class (stored as a std::string) can also be used in an Is() query. If the queried instance is not the specified type, the Is() methods walk the class hierarchy looking for the type identifier. The QueryInterface() method has the same behavior but returns an RTTI pointer instead of a Boolean. The templated As() method returns a pointer of the specified type or NULL if the interface is not of the queried type. You see an application of this system shortly.

## Drawable Game Components

Drawable game components are extensions of regular game components and add Draw() and Visible() methods. They also include the concept of a camera, but we defer that topic for a few sections. Instead of listing the entire implementation, [Listing 12.4](#) presents only the DrawableGameComponent.h header file. You can deduce the simple implementation or download it from the companion website.

### **Listing 12.4** The DrawableGameComponent.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class Camera;

    class DrawableGameComponent : public GameComponent
    {
        RTTI_DECLARATIONS(DrawableGameComponent, GameComponent)

    public:
        DrawableGameComponent();
        DrawableGameComponent(Game& game);
        DrawableGameComponent(Game& game, Camera& camera);
        virtual ~DrawableGameComponent();

        bool Visible() const;
        void SetVisible(bool visible);

        Camera* GetCamera();
    };
}
```

```

void SetCamera(Camera* camera);

virtual void Draw(const GameTime& gameTime);

protected:
    bool mVisible;
    Camera* mCamera;

private:
    DrawableGameComponent(const DrawableGameComponent& rhs);
    DrawableGameComponent& operator=(const
DrawableGameComponent&
rhs);
    };
}

```

---

## An Updated Game Class

You must update the `Library::Game` class to support game components. The class declaration will now include a member for the list of components, and the implementations of `Game::Initialize()`, `Game::Update()`, and `Game::Draw()` will act on these components. [Listing 12.5](#) presents the minor updates to the `Game.h` header file. [Listing 12.6](#) shows the implementations for the `Initialize()`, `Update()`, and `Draw()` methods.

### **Listing 12.5** Updated `Game.h` File to Support Game Components (Abbreviated)

[Click here to view code image](#)

---

```

class Game
{
public:
    /* ... Previously presented members removed for brevity
...
    const std::vector<GameComponent*>& Components() const;

protected:
    /* ... Previously presented members removed for brevity
...
    std::vector<GameComponent*> mComponents;
};

```

---

### **Listing 12.6** Updated `Game.cpp` File to Support Game Components (Abbreviated)

[Click here to view code image](#)

---

```
/* ... Previously presented members removed for brevity ... */
```

```

void Game::Initialize()
{
    for (GameComponent* component : mComponents)
    {
        component->Initialize();
    }
}

void Game::Update(const GameTime& gameTime)
{
    for (GameComponent* component : mComponents)
    {
        if (component->Enabled())
        {
            component->Update(gameTime);
        }
    }
}

void Game::Draw(const GameTime& gameTime)
{
    for (GameComponent* component : mComponents)
    {
        DrawableGameComponent* drawableGameComponent =
component->As<DrawableGameComponent>();
        if (drawableGameComponent != nullptr &&
drawableGameComponent->Visible())
        {
            drawableGameComponent->Draw(gameTime);
        }
    }
}

```

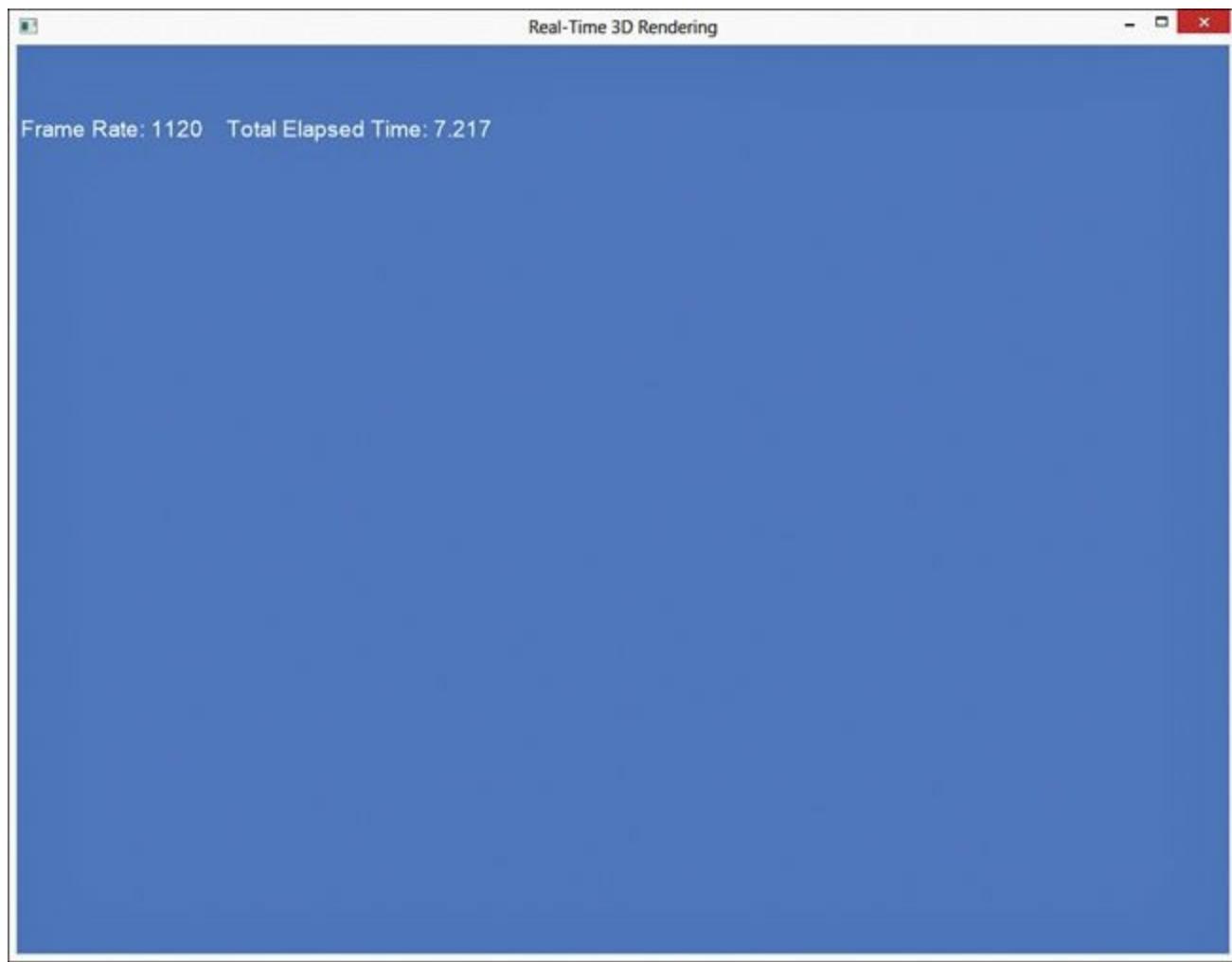
---

Note the use of C++ 11 *range-based for loops* in [Listing 12.5](#). These statements are analogous to traditional STL iterator usage, just with happier syntax. Also note the `Enabled()` and `Visible()` checks for game and drawable game components within the `Update()` and `Draw()` methods, and the `RTTI::As()` call to verify that the component is, in fact, a `DrawableGameComponent`. Alternately, you could consider storing separate vectors for `GameComponent` and `DrawableGameComponent` objects to eliminate this particular need for runtime type checking.

## A Frame Rate Component

To demonstrate the component system, you next create a component to display the frame rate of the application in frames per second (FPS). This example also introduces a bitmapped font system for rendering text to the screen. [Figure 12.2](#) shows the desired output of the component. It displays the frame rate and the total elapsed time of the application in a white font toward the top of the screen.

[Listing 12.7](#) presents the header file for the FpsComponent class.



**Figure 12.2** Output of the frame rate component.

### **Listing 12.7** The FpsComponent.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "DrawableGameComponent.h"

namespace DirectX
{
    class SpriteBatch;
    class SpriteFont;
}

namespace Library
{
    class FpsComponent : public DrawableGameComponent
    {
        RTTI_DECLARATIONS(FpsComponent, DrawableGameComponent)
```

```

public:
    FpsComponent(Game& game);
    ~FpsComponent();

    XMFLOAT2& TextPosition();
    int FrameRate() const;

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    FpsComponent();
    FpsComponent(const FpsComponent& rhs);
    FpsComponent& operator=(const FpsComponent& rhs);

    SpriteBatch* mSpriteBatch;
    SpriteFont* mSpriteFont;
    XMFLOAT2 mTextPosition;

    int mFrameCount;
    int mFrameRate;
    double mLastTotalElapsedTime;
};

}

```

---

The `FpsComponent` derives from the `DrawableGameComponent` class; provides implementations for the `Initialize()`, `Update()`, and `Draw()` methods; and stores class members specific to its task. This is the typical pattern for all drawable game components.

## SpriteBatch and SpriteFont

Note the `SpriteBatch` and `SpriteFont` members in the `FpsComponent` class. These types come from the DirectX Tool Kit (DirectXTK), discussed in [Chapter 3](#), “[Tools of the Trade](#).[“](#) If you haven’t already linked this library, review the related material in [Chapter 3](#) and [Chapter 10](#), or visit the companion website for references to the library.

The `SpriteBatch` class is used to render **sprites**. A sprite is just a texture rendered to the 2D screen surface instead of being mapped to an object in 3D space. As such, sprites do not require a 3D camera. Sprites are commonly used for user interfaces (scores, buttons, or mini-maps) and **2D platformers** (games such as *Super Mario Bros.*). Drawing sprites with the `SpriteBatch` class follows this pattern:

[Click here to view code image](#)

```

mSpriteBatch->Begin();
mSpriteBatch->Draw(texture, position);
mSpriteBatch->End();

```

Note that more than one sprite draw call can be executed between the calls to `SpriteBatch::Begin()` and `SpriteBatch::End()`.

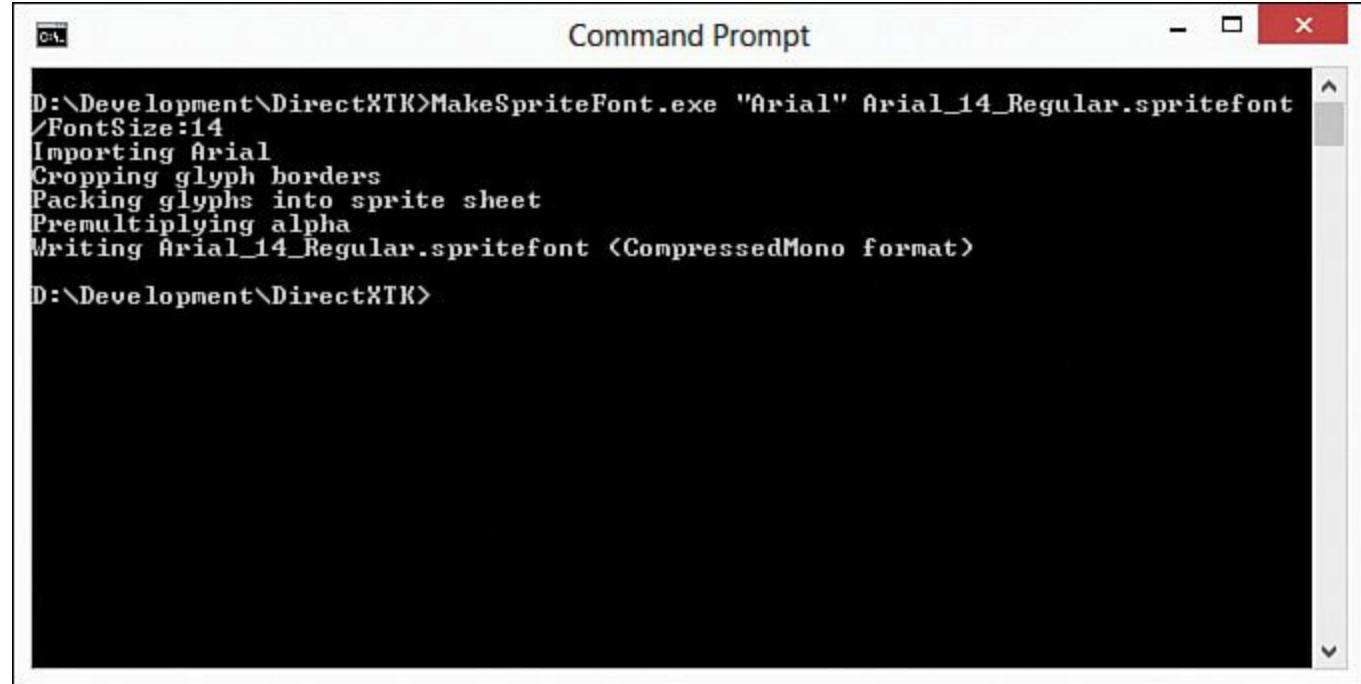
For the frame rate component, you use the `SpriteBatch` class to render strings representing the frame rate and total elapsed time. Drawing text in Direct3D isn't done like it is with a Windows or Windows Console application. In Direct3D, text is rendered as a sprite, and the output characters come from a texture that contains the desired character set (such as ASCII) in a particular font.

The `SpriteFont` class represents a *TrueType* font that has been converted to a bitmap using the `MakeSpriteFont` tool (included with the DirectXTK package). This tool outputs a binary file with an extension of `.spritefont`, that is used to instantiate a `SpriteFont` object. The following command creates a file named `Arial_14-Regular.spritefont` using the Arial font family and a point size of 14:

[Click here to view code image](#)

```
MakeSpriteFont.exe "Arial" Arial_14-Regular.spritefont  
/FontSize:14
```

[Figure 12.3](#) shows the invocation of this command on the DOS command prompt. Visit the DirectXTK website for more options available with the `MakeSpriteFont` tool.



**Figure 12.3** Invocation of the `MakeSpriteFont` tool.

To use the resulting `.spritefont` file within your application, it should reside in a directory that's relative to your executable. One approach for such data is to create a folder named `content` under the `source\Library` directory. Its location denotes its association with functionality that resides within the Library project. You then augment the Library project's post-build event to copy any files within the `content` directory to `$(SolutionDir)..\content`. You create the opposite command for the Game project's prebuild event to copy the files from `$(SolutionDir)..\content` to the output directory of the game. [Listing 12.8](#) presents the updated post-build event for the Library project, and [Listing 12.9](#) presents the prebuild event for the Game project. Be sure to apply these changes to both debug and release configurations.

## **Listing 12.8** The Library Project's Post-build Event

[Click here to view code image](#)

```
mkdir "$(SolutionDir)..\lib\"  
copy "$(TargetPath)" "$(SolutionDir)..\lib\"  
  
mkdir "$(SolutionDir)..\content\"  
IF EXIST "$(ProjectDir)Content" xcopy /E /Y  
"$(ProjectDir)Content"  
"$(SolutionDir)..\content\"  
IF EXIST "$(TargetDir)Content" xcopy /E /Y "$(TargetDir)Content"  
"$(SolutionDir)..\content\"
```

## **Listing 12.9** The Game Project's Prebuild Event

[Click here to view code image](#)

```
mkdir "$(OutDir)Content"  
IF EXIST "$(SolutionDir)..\content" xcopy /E /Y  
"$(SolutionDir)..\content" "$(OutDir)Content\"  
IF EXIST "$(ProjectDir)Content" xcopy /E /Y  
"$(ProjectDir)Content"  
"$(OutDir)Content\"
```

Notice that the Game project's prebuild event adopts the same \$(ProjectDir)Content directory structure for game-specific content. Furthermore, the Library project also copies content from a \$(TargetDir)Content directory. That directory will be used for shaders that are compiled as part of the Visual Studio build process. [Chapter 14](#) discusses shader compilation.

With a .spritefont file created and copied to a folder accessible by the executable, it can be used to instantiate a SpriteFont object with a call such as this:

[Click here to view code image](#)

```
mSpriteFont = new SpriteFont(mGame->Direct3DDevice(),  
L"Content\\Fonts\\Arial_14-Regular.spritefont");
```

The SpriteFont class has a DrawString() method with parameters for the sprite batch, the string to output, and the 2D screen location. [Listing 12.10](#) shows the full listing of the FpsComponent.cpp file, including the SpriteBatch and SpriteFont usage.

## **Listing 12.10** The FpsComponent.cpp File

[Click here to view code image](#)

```
#include "FpsComponent.h"
```

```
#include <sstream>
#include <iomanip>
#include <SpriteBatch.h>
#include <SpriteFont.h>
#include "Game.h"
#include "Utility.h"

namespace Library
{
    RTTI_DEFINITIONS(FpsComponent)

    FpsComponent::FpsComponent(Game& game)
        : DrawableGameComponent(game), mSpriteBatch(nullptr),
          mSpriteFont(nullptr), mTextPosition(0.0f, 60.0f),
          mFrameCount(0), mFrameRate(0),
    mLastTotalElapsedTime(0.0)
    {
    }

    FpsComponent::~FpsComponent()
    {
        DeleteObject(mSpriteFont);
        DeleteObject(mSpriteBatch);
    }

    XMFLOAT2& FpsComponent::TextPosition()
    {
        return mTextPosition;
    }

    int FpsComponent::FrameRate() const
    {
        return mFrameCount;
    }

    void FpsComponent::Initialize()
    {
        SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

        mSpriteBatch = new SpriteBatch(mGame-
>Direct3DDeviceContext());
        mSpriteFont = new SpriteFont(mGame->Direct3DDevice(),
L"Content\\Fonts\\Arial_14-Regular.spritefont");
    }

    void FpsComponent::Update(const GameTime& gameTime)
    {
```

```

1) if (gameTime.TotalGameTime() - mLastTotalElapsedTIme >=
{
    mLastTotalElapsedTIme = gameTime.TotalGameTime();
    mFrameRate = mFrameCount;
    mFrameCount = 0;
}

mFrameCount++;
}

void FpsComponent::Draw(const GameTime& gameTime)
{
    mSpriteBatch->Begin();

    std::wostringstream fpsLabel;
    fpsLabel << std::setprecision(4) << L"Frame Rate: " <<
mFrameRate << " Total Elapsed Time: " <<
gameTime.TotalGameTime();
    mSpriteFont->DrawString(mSpriteBatch,
fpsLabel.str().c_str(),
mTextPosition);

    mSpriteBatch->End();
}
}

```

---

The `FpsComponent::Initialize()` method first sets the current working directory to the executable directory. This is so that access to the `Content\Fonts` directory is performed from the correct location. The associated `Utility` class contains a variety of useful methods; you can find it on the book's companion website. Next, the `Initialize()` method instantiates the `SpriteBatch` and `SpriteFont` objects. These objects are released in the component's destructor.

The `FpsComponent::Update()` method increments the `mFrameCount` member with each call and resets it after a second of time has passed. The `FpsComponent::Draw()` method builds a string and renders it through the `mSpriteBatch` and `mSpriteFont` members.

## Integrating the Component

The `FpsComponent` should reside in the `Library` project, but you'll integrate the component in the `RenderingContext` class of your `Game` project. To do this, add an `FpsComponent` member within the `RenderingContext` class and update the `RenderingContext::Initialize()` and `RenderingContext::Shutdown()` methods to match [Listing 12.11](#).

### **Listing 12.11 Integration of the FpsComponent Within the RenderingGame Class**

[Click here to view code image](#)

```

void RenderingGame::Initialize()
{
    mFpsComponent = new FpsComponent(*this);
    mComponents.push_back(mFpsComponent);

    Game::Initialize();
}

void RenderingGame::Shutdown()
{
    DeleteObject(mFpsComponent);

    Game::Shutdown();
}

```

---

The component is registered by adding it to the `Game::mComponents` vector. Note that you do not need to explicitly call the component's `Initialize()`, `Update()`, or `Draw()` methods because the base `Game` class invokes them. Run the application, and you should see output similar to [Figure 12.2](#). The patterns set in this example are common for all the components you create with this framework.

## Device Input

Another supporting system for interactive rendering is the capability to accept device input. On the PC, this is commonly mouse, keyboard, and gamepad input. This section discusses mouse and keyboard input and you'll develop corresponding game components.

Two general approaches are useful in collecting device input: You can either poll the device periodically or wait for the device to inform you that its state has changed. The approach you choose should take into account the frequency of input changes, the cost to poll the device, and the cost to process an event. If the device is expected to change every frame, polling the device might make more sense than processing a flood of events. Conversely, if the device is mostly idle and changes state only periodically, you might choose an event-based input system. You need not employ a one-size-fits-all approach; you can poll one device and accept events for another.

Multiple APIs can be used for device input. The Windows API, for example, supports keyboard and mouse events through the windows procedure (the `WndProc()` method you wrote for the `Game` class) or with polling methods such as `GetAsyncKeyState()`. DirectX 11 includes the **XInput** system for querying Xbox 360 game controllers, although it lacks support for mouse and keyboard input. The DirectX 11 installation also includes the older **DirectInput** library. DirectInput supports mouse, keyboard, gamepad, and joystick input, but it hasn't been updated since DirectX 8. Microsoft recommends the use of XInput for "next-generation" game controllers, but XInput also hasn't seen a major update since DirectX 9.

For the components in this book, you use the DirectInput library for polling the keyboard and mouse.

## Keyboard Input

Using DirectInput to query the keyboard translates into the following steps:

1. Create the DirectInput object.
2. Create the DirectInput keyboard device.
3. Set the device data format and cooperative level.
4. Acquire the device.
5. Query the device state.

The next few sections cover each of these steps.

## Create the DirectInput Object

The `IDirectInput8` interface enumerates, creates, and queries DirectInput devices. You must create an object of this type before performing any of these actions. You accomplish this through the `DirectInput8Create()` method; the prototype and parameters are listed next. The `dinput.h` header file declares this method and all DirectInput-related functionality.

[Click here to view code image](#)

```
HRESULT DirectInput8Create(
    HINSTANCE hinst, DWORD dwVersion, REFIID riidltf,
    LPVOID *ppvOut, LPUNKNOWN punkOuter);
```

- **hinst:** Specifies the handle to the application that is creating the DirectInput object. This is the same handle you used to instantiate a window, and it's stored in the `mInstance` member of the `Game` class.
- **dwVersion:** The version number of DirectInput. This is always `DIRECTINPUT_VERSION`.
- **riidltf:** The unique identifier of the requested interface. This is always `IID_IDirectInput8`.
- **ppvOut:** Returns the created DirectInput object.
- **punkOuter:** Used for COM aggregation. This is always `NULL`.

The keyboard and mouse components use the created DirectInput object and release it after those components have been freed. Thus, you should store this object as a member in your specialized `RenderingGame` class, to be instantiated in the `Initialize()` method and released in `Shutdown()`. [Listing 12.12](#) presents an example call to `DirectInput8Create()` and its cleanup within the `RenderingGame` implementation.

## Listing 12.12 DirectInput Object Creation and Release

[Click here to view code image](#)

---

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(mInstance,
        DIRECTINPUT_VERSION,
        IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }
}
```

```

/* ... Previously presented statements removed for brevity ... */
}

void RenderingGame::Shutdown()
{
    /* ... Previously presented statements removed for brevity
... */

    ReleaseObject(mDirectInput);

    Game::Shutdown();
}

```

---

## Create the DirectX Keyboard Device

With the DirectInput object instantiated, you can now create the specific device. To do this, add a class called `Keyboard` to the Library project. This class derives from `GameComponent` and encapsulates the functionality for creating, configuring, acquiring, releasing, and querying the keyboard. Add a class member with the `IDirectInputDevice8` interface, which stores the created device. You create the device with a call to `IDirectInput8::CreateDevice()`; the prototype and parameters are listed next:

[Click here to view code image](#)

```

HRESULT CreateDevice(
    REFGUID rguid,
    LPDIRECTINPUTDEVICE * lplpDirectInputDevice,
    LPUNKNOWN pUnkOuter);

```

- **rguid:** The unique identifier of the requested input device. For a keyboard, this is `GUID_SysKeyboard`.
- **lplpDirectInputDevice:** Returns the created DirectInput device.
- **pUnkOuter:** Used for COM aggregation. This is always `NULL`.

Creating, configuring, and acquiring an input device is typically done in the same code block. Thus, we defer an example call until we cover these topics.

## Set the Device Data Format and Cooperative Level

After the input device is created, you must specify the format of the data the device should return. This is done through the `IDirectInputDevice8::SetDataFormat()` method, which has only one parameter: a `DIDATAFORMAT` structure. You can configure a custom `DIDATAFORMAT` structure or use one of the following predefined instances:

- `c_dfDIKeyboard`
- `c_dfDIMouse`
- `c_dfDIMouse2`

- c\_dfDIJoystick
- c\_dfDIJoystick2

For the keyboard component, you specify the `c_dfdIKeyboard` structure.

Next, you must set the **cooperative level**, which determines how the device interacts with other instances of the same device and with the rest of the operating system. This is accomplished with a call to `IDirectInputDevice8::SetCooperativeLevel()`, which has the following prototype:

[Click here to view code image](#)

```
HRESULT SetCooperativeLevel (
    HWND hwnd,
    DWORD dwFlags);
```

- **hwnd:** The top-level window handle that belongs to the application. This is created within `Game::InitializeWindow()` and stored in the `Game::mWindowHandle` member.
- **dwFlags:** Bitwise OR'd flags that describe the cooperative level. [Table 12.1](#) lists the possible flags.

Flag	Description
<code>DISCL_BACKGROUND</code>	The device can be used even when the associated window is not active.
<code>DISCL_EXCLUSIVE</code>	The application requires exclusive access to the device.
<code>DISCL_FOREGROUND</code>	The device is automatically unacquired when the associated window is moved to the background (becomes the nonactive window).
<code>DISCL_NONEXCLUSIVE</code>	The application requires nonexclusive access to the device and does not interfere with other applications using the same device.
<code>DISCL_NOWINKEY</code>	This disables the Windows key.

**Table 12.1** DirectInput Device Cooperative-Level Flags

## Acquire the Device

After setting the data format and cooperative level, you are ready to **acquire** the device. A successful device acquisition means that you have access to the device and can query its state. Device acquisition is done through the `IDirectInputDevice8::Acquire()` method, which has no input parameters. [Listing 12.13](#) presents the implementation of the `Keyboard::Initialize()` method, which demonstrates this call and the ones described in the last few sections.

## Listing 12.13 The `Keyboard::Initialize()` Method

[Click here to view code image](#)

```
void Keyboard::Initialize()
{
    if (FAILED(mDirectInput->CreateDevice(GUID_SysKeyboard,
```

```

&mDevice,
nullptr) ) )
{
    throw GameException( "IDIRECTINPUT8::CreateDevice() failed" );
}

if (FAILED(mDevice->SetDataFormat (&c_dfDIKeyboard) ))
{
    throw
GameException( "IDIRECTINPUTDEVICE8::SetDataFormat() failed" );
}

if (FAILED(mDevice->SetCooperativeLevel (mGame-
>WindowHandle(),
DISCL_FOREGROUND | DISCL_NONEXCLUSIVE) ))
{
    throw
GameException( "IDIRECTINPUTDEVICE8::SetCooperativeLevel() failed" );
}

if (FAILED(mDevice->Acquire( )))
{
    throw GameException( "IDIRECTINPUTDEVICE8::Acquire() failed" );
}
}

```

---

## Query the Device State

Now that the device has been initialized and acquired, you can query its state. This is done with a call to `IDirectInputDevice8::GetDeviceState()`, which has the following prototype and parameters:

[Click here to view code image](#)

```

HRESULT GetDeviceState(
    DWORD cbData,
    LPVOID lpvData);

```

- **cbData:** The size (in bytes) of the buffer specified in **lpvData**.
- **lpvData:** The buffer to write the device state to. The structure of this buffer is defined by the format specified through `IDirectInputDevice8::SetDataFormat()`. When using the predefined `c_dfDIKeyboard` data format structure, this buffer is simply an array of 256 bytes.

## A Keyboard Component

Instead of just presenting a call to `IDirectInputDevice8::GetDeviceState()`, the next two listings present the complete declaration ([Listing 12.14](#)) and implementation ([Listing 12.15](#)) of the `Keyboard` class. You can find the `IDirectInputDevice8::GetDeviceState()` call within the `Keyboard::Update()` method.

## **Listing 12.14** The `Keyboard.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class Keyboard : public GameComponent
    {
        RTTI_DECLARATIONS(Keyboard, GameComponent)

    public:
        Keyboard(Game& game, LPDIRECTINPUT8 directInput);
        ~Keyboard();

        const byte* const CurrentState() const;
        const byte* const LastState() const;

        virtual void Initialize() override;
        virtual void Update(const GameTime& gameTime) override;

        bool IsKeyUp(byte key) const;
        bool IsKeyDown(byte key) const;
        bool WasKeyUp(byte key) const;
        bool WasKeyDown(byte key) const;
        bool WasKeyPressedThisFrame(byte key) const;
        bool WasKeyReleasedThisFrame(byte key) const;
        bool IsKeyHeldDown(byte key) const;

    private:
        Keyboard();
        static const int KeyCount = 256;

        Keyboard(const Keyboard& rhs);

        LPDIRECTINPUT8 mDirectInput;
        LPDIRECTINPUTDEVICE8 mDevice;
```

```
    byte mcurrentState[KeyCount];
    byte mLastState[KeyCount];
};

}
```

---

## Listing 12.15 The Keyboard.cpp File

[Click here to view code image](#)

---

```
#include "Keyboard.h"
#include "Game.h"
#include "GameTime.h"
#include "GameException.h"

namespace Library
{
    RTTI_DEFINITIONS(Keyboard)

    Keyboard::Keyboard(Game& game, LPDIRECTINPUT8 directInput)
        : GameComponent(game), mDirectInput(directInput),
mDevice(nullptr)
    {
        assert(mDirectInput != nullptr);
        ZeroMemory(mcurrentState, sizeof(mcurrentState));
        ZeroMemory(mLastState, sizeof(mLastState));
    }

    Keyboard::~Keyboard()
    {
        if (mDevice != nullptr)
        {
            mDevice->Unacquire();
            mDevice->Release();
            mDevice = nullptr;
        }
    }

    const byte* const Keyboard::CurrentState() const
    {
        return mcurrentState;
    }

    const byte* const Keyboard::LastState() const
    {
        return mLastState;
    }
}
```

```
void Keyboard::Initialize()
{
    if (FAILED(mDirectInput->CreateDevice(GUID_SysKeyboard,
&mDevice, nullptr)))
    {
        throw GameException("IDIRECTINPUT8::CreateDevice() failed");
    }

    if (FAILED(mDevice->SetDataFormat(&c_dfDIKeyboard)))
    {
        throw
GameException("IDIRECTINPUTDEVICE8::SetDataFormat() failed");
    }
}

if (FAILED(mDevice->SetCooperativeLevel(mGame-
>WindowHandle(),
DISCL_FOREGROUND | DISCL_NONEXCLUSIVE)))
{
    throw GameException("IDIRECTINPUTDEVICE8::SetCooperativeLevel() failed");
}

if (FAILED(mDevice->Acquire()))
{
    throw GameException("IDIRECTINPUTDEVICE8::Acquire() failed");
}
}

void Keyboard::Update(const GameTime& gameTime)
{
    if (mDevice != nullptr)
    {
        memcpy(mLastState, mCurrentState,
sizeof(mCurrentState));

        if (FAILED(mDevice-
>GetDeviceState(sizeof(mCurrentState),
(LPVOID)mCurrentState)))
        {
            // Try to reacquire the device
            if (SUCCEEDED(mDevice->Acquire()))
            {
                mDevice-
```

```

>GetDeviceState(sizeof(mCurrentState),
(LPVOID)mCurrentState);
}
}

bool Keyboard::IsKeyUp(byte key) const
{
    return ((mCurrentState[key] & 0x80) == 0);
}

bool Keyboard::IsKeyDown(byte key) const
{
    return ((mCurrentState[key] & 0x80) != 0);
}

bool Keyboard::WasKeyUp(byte key) const
{
    return ((mLastState[key] & 0x80) == 0);
}

bool Keyboard::WasKeyDown(byte key) const
{
    return ((mLastState[key] & 0x80) != 0);
}

bool Keyboard::WasKeyPressedThisFrame(byte key) const
{
    return (IsKeyDown(key) && WasKeyUp(key));
}

bool Keyboard::WasKeyReleasedThisFrame(byte key) const
{
    return (IsKeyUp(key) && WasKeyDown(key));
}

bool Keyboard::IsKeyHeldDown(byte key) const
{
    return (IsKeyDown(key) && WasKeyDown(key));
}

```

---

This implementation follows the previously established game component pattern. It creates, configures, and acquires the input device within the `Initialize()` method and then queries the device state with each call to `Update()`. Notice the two class members `mCurrentState` and `mLastState`, which store the current state of the device and the state from the previous frame. This

allows you to ask questions such as `WasKeyPressedThisFrame()`, `WasKeyReleasedThisFrame()`, or `IsKeyHeldDown()`. The parameter to such methods is the index to the state byte array. The DirectInput library includes a set of definitions to use with these queries. You can find these in the `dinput.h` header file; they all begin with the `DIK_` prefix. [Table 12.2](#) lists a few of these scan codes.

Scan Code	Description
<code>DIK_ESCAPE</code>	The escape key
<code>DIK_W, DIK_A, DIK_S, DIK_D</code>	The W, A, S, and D keys (commonly used for camera movement)
<code>DIK_SPACE</code>	The spacebar

**Table 12.2** Common DirectInput Keyboard Scan Codes

## Reacquiring the Device

Notice the additional `IDirectInputDevice8::Acquire()` call within the `Keyboard::Update()` method in [Listing 12.15](#). Throughout the execution of your application, you might lose access to the device; if so, you must reacquire access before the device can be read. This can happen, for example, if another application requests exclusive access to the device. For the previous implementation, if the state retrieval fails, a single attempt is made (per frame) to reacquire the device. No attempt is made to error out or warn the user if the device can't be reacquired. You might want to add such a feature to your own implementation.

## Integrating the Keyboard Component

As with the frame rate component, you integrate your keyboard component in the `RenderingGame` class of your Game project. Add a `Keyboard` class member, and update your `RenderingGame::Initialize()` and `RenderingGame::Shutdown()` methods to match [Listing 12.16](#). This listing also includes a modification to the `RenderingGame::Update()` method that exits the application when the Escape key is pressed.

### Listing 12.16 Integration of the Keyboard Component Within the `RenderingGame` Class

[Click here to view code image](#)

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(mInstance,
DIRECTINPUT_VERSION,
IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    mKeyboard = new Keyboard(*this, mDirectInput);
    mComponents.push_back(mKeyboard);
```

```

mFpsComponent = new FpsComponent(*this);
mComponents.push_back(mFpsComponent);

Game::Initialize();

}

void RenderingGame::Shutdown()
{
    DeleteObject(mKeyboard);
    DeleteObject(mFpsComponent);

    ReleaseObject(mDirectInput);

    Game::Shutdown();
}

void RenderingGame::Update(const GameTime &gameTime)
{
    if (mKeyboard->WasKeyPressedThisFrame(DIK_ESCAPE))
    {
        Exit();
    }

    Game::Update(gameTime);
}

```

---

## Mouse Input

To access the mouse with DirectInput, you use the very same interfaces and methods you did for the keyboard. The chief difference is the data you query. Instead of byte arrays, you store mcurrentState and mLastState members whose types are of the DIMOUSESTATE structure. This structure has the following definition:

[Click here to view code image](#)

```

typedef struct _DIMOUSESTATE {
    LONG     lX;
    LONG     lY;
    LONG     lZ;
    BYTE    rgButtons[4];
} DIMOUSESTATE;

```

The lX and lY members of this structure store the X and Y positions of the mouse. The lZ member stores the position of the mouse wheel. These positions are relative to their previous values. A negative X value indicates that the mouse has moved to the left, and a positive Y value indicates that the mouse has moved down (toward the bottom of the screen). The higher the magnitude of the value, the more the mouse has moved from its previously polled position.

The rgButtons array supports a four-button mouse, and the elements of the array are identified

by an enumeration such as this:

[Click here to view code image](#)

```
enum MouseButtons
{
    MouseButtonsLeft = 0,
    MouseButtonsRight = 1,
    MouseButtonsMiddle = 2,
    MouseButtonsX1 = 3
};
```

To configure the proper data format, you specify `c_dfDIMouse` in the call to `IDirectInputDevice8::SetDataFormat()`. You also specify `GUID_SysMouse` instead of `GUID_SysKeyboard` as the first argument to `IDirectInputDevice8::CreateDevice()`.

[Listings 12.17](#) and [12.18](#) present the declaration and implementation of the `Mouse` class.

### **Listing 12.17** The `Mouse.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class GameTime;

    enum MouseButtons
    {
        MouseButtonsLeft = 0,
        MouseButtonsRight = 1,
        MouseButtonsMiddle = 2,
        MouseButtonsX1 = 3
    };

    class Mouse : public GameComponent
    {
        RTTI_DECLARATIONS(Mouse, GameComponent)

    public:
        Mouse(Game& game, LPDIRECTINPUT8 directInput);
        ~Mouse();

        LPDIMOUSESTATE CurrentState();
        LPDIMOUSESTATE LastState();
    };
}
```

```

virtual void Initialize() override;
virtual void Update(const GameTime& gameTime) override;

long X() const;
long Y() const;
long Wheel() const;

bool IsButtonUp(MouseButtons button) const;
bool IsButtonDown(MouseButtons button) const;
bool WasButtonUp(MouseButtons button) const;
bool WasButtonDown(MouseButtons button) const;
bool WasButtonPressedThisFrame(MouseButtons button)
const;
bool WasButtonReleasedThisFrame(MouseButtons button)
const;
bool IsButtonHeldDown(MouseButtons button) const;

private:
Mouse();

LPDIRECTINPUT8 mDirectInput;
LPDIRECTINPUTDEVICE8 mDevice;
DIMOUSESTATE mCurrentState;
DIMOUSESTATE mLastState;

long mX;
long mY;
long mWheel;
};

}

```

---

## Listing 12.18 The Mouse.cpp File

[Click here to view code image](#)

---

```

#include "Mouse.h"
#include "Game.h"
#include "GameTime.h"
#include "GameException.h"

namespace Library
{
RTTI_DEFINITIONS(Mouse)

Mouse::Mouse(Game& game, LPDIRECTINPUT8 directInput)

```

```
: GameComponent(game), mDirectInput(directInput),
mDevice(nullptr), mX(0), mY(0), mWheel(0)
{
    assert(mDirectInput != nullptr);
    ZeroMemory(&mCurrentState, sizeof(mCurrentState));
    ZeroMemory(&mLastState, sizeof(mLastState));
}

Mouse::~Mouse()
{
    if (mDevice != nullptr)
    {
        mDevice->Unacquire();
        mDevice->Release();
        mDevice = nullptr;
    }
}

LPDIMOUSESTATE Mouse::GetCurrentState()
{
    return &mCurrentState;
}

LPDIMOUSESTATE Mouse::GetLastState()
{
    return &mLastState;
}

long Mouse::X() const
{
    return mX;
}

long Mouse::Y() const
{
    return mY;
}

long Mouse::Wheel() const
{
    return mWheel;
}

void Mouse::Initialize()
{
    if (FAILED(mDirectInput->CreateDevice(GUID_SysMouse,
&mDevice,
```

```
nullptr) ) )
{
    throw GameException( "IDIRECTINPUT8::CreateDevice() failed" );
}

if (FAILED(mDevice->SetDataFormat(&c_dfDIMouse)))
{
    throw
GameException( "IDIRECTINPUTDEVICE8::SetDataFormat() failed" );
}

if (FAILED(mDevice->SetCooperativeLevel(mGame->windowHandle(),
DISCL_FOREGROUND | DISCL_NONEXCLUSIVE)))
{
    throw GameException( "IDIRECTINPUTDEVICE8::SetCooperativeLevel() failed" );
}

if (FAILED(mDevice->Acquire()))
{
    throw GameException( "IDIRECTINPUTDEVICE8::Acquire() failed" );
}

void Mouse::Update(const GameTime& gameTime)
{
    if (mDevice != nullptr)
    {
        memcpy(&mLastState, &mCurrentState,
sizeof(mCurrentState));

        if (FAILED(mDevice->GetDeviceState(sizeof(mCurrentState),
&mCurrentState)))
        {
            // Try to reaqcuire the device
            if (SUCCEEDED(mDevice->Acquire()))
            {
                if (FAILED(mDevice->GetDeviceState(sizeof(mCurrentState),
&mCurrentState)))
                {
                    return;
                }
            }
        }
    }
}
```

```
        }

        // Accumulate positions
        mX += mCurrentState.lX;
        mY += mCurrentState.lY;
        mWheel += mCurrentState.lZ;
    }

}

bool Mouse::IsButtonUp(MouseButtons button) const
{
    return ((mCurrentState.rgbButtons[button] & 0x80) == 0);
}

bool Mouse::IsButtonDown(MouseButtons button) const
{
    return ((mCurrentState.rgbButtons[button] & 0x80) != 0);
}

bool Mouse::WasButtonUp(MouseButtons button) const
{
    return ((mLastState.rgbButtons[button] & 0x80) == 0);
}

bool Mouse::WasButtonDown(MouseButtons button) const
{
    return ((mLastState.rgbButtons[button] & 0x80) != 0);
}

bool Mouse::WasButtonPressedThisFrame(MouseButtons button)
const
{
    return (IsButtonDown(button) && WasButtonUp(button));
}

bool Mouse::WasButtonReleasedThisFrame(MouseButtons button)
const
{
    return (IsButtonUp(button) && WasButtonDown(button));
}

bool Mouse::IsButtonHeldDown(MouseButtons button) const
{
    return (IsButtonDown(button) && WasButtonDown(button));
}
```

## Integrating the Mouse Component

[Listing 12.19](#) presents the code to integrate the Mouse component into your `RenderingGame` class. This listing includes a modification to the `RenderingGame::Draw()` method that outputs the mouse position and wheel value through the `SpriteBatch/SpriteFont` system. Note that the mouse position values are accumulated from the relative values returned by `IDirectInputDevice8::GetDeviceState()`, and no facility has been created to establish a position origin or to limit the values to the application window or screen; it's left as an exercise for you to add this functionality.

### **Listing 12.19** Integration of the Mouse Component Within the `RenderingGame` Class

[Click here to view code image](#)

---

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(mInstance,
DIRECTINPUT_VERSION,
IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

mKeyboard = new Keyboard(*this, mDirectInput);
mComponents.push_back(mKeyboard);

mMouse = new Mouse(*this, mDirectInput);
mComponents.push_back(mMouse);

mFpsComponent = new FpsComponent(*this);
mComponents.push_back(mFpsComponent);

SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

mSpriteBatch = new SpriteBatch(mDirect3DDeviceContext);
mSpriteFont = new SpriteFont(mDirect3DDevice,
L"Content\\Fonts\\Arial_14-Regular.spritefont");

Game::Initialize();
}

void RenderingGame::Shutdown()
{
    DeleteObject(mKeyboard);
    DeleteObject(mMouse);
    DeleteObject(mFpsComponent);
    DeleteObject(mSpriteFont);
```

```

DeleteObject(mSpriteBatch);

ReleaseObject(mDirectInput);

Game::Shutdown();

}

void RenderingGame::Draw(const GameTime &gameTime)
{
    mDirect3DDeviceContext-
>ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext-
>ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

Game::Draw(gameTime);

mSpriteBatch->Begin();

std::wostringstream mouseLabel;
mouseLabel << L"Mouse Position: " << mMoue->X() << ", " <<
mMoue->Y() << " Mouse Wheel: " << mMoue->Wheel();
    mSpriteFont->DrawString(mSpriteBatch,
mouseLabel.str().c_str(),
mMouseTextPosition);

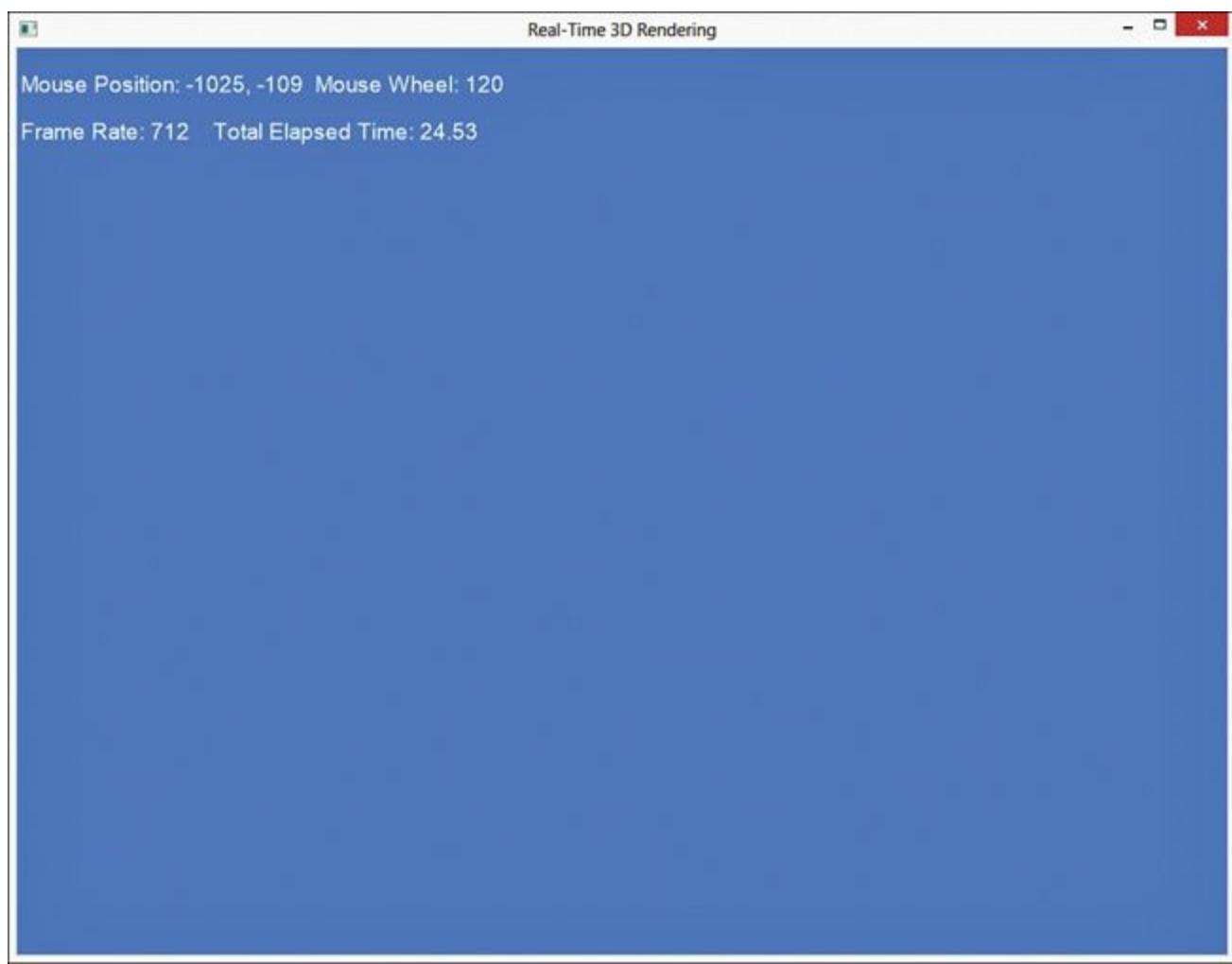
mSpriteBatch->End();

HRESULT hr = mSwapChain->Present(0, 0);
if (FAILED(hr))
{
    throw GameException("IDXGISwapChain::Present() failed.",
hr);
}
}

```

---

[Figure 12.4](#) shows the output of the integrated mouse component.



**Figure 12.4** Output of the integrated mouse component.

## Software Services

One final piece of scaffolding deserves discussion before you refocus on 3D rendering: **software services**. A service is just an object (any kind of object) that is useful to other software systems. For example, you might use the keyboard and mouse components throughout your application, and they would be good candidates as services. You could pass such objects as arguments to constructors or through public mutators, but that increases **coupling** between classes (the degree to which software modules depend on other modules). Good programming practices encourage low coupling, and software services can help. Instead of publicly prescribing necessary software systems, a component can internally query a service container to find the modules it needs. Furthermore, a component's implementation can change, requiring more or fewer services, without impacting its public interface. The demos in the coming chapters make extensive use of services.

## A Service Container Class

[Listing 12.20](#) presents the declaration of the `ServiceContainer` class, a thin wrapper around an `std::map` that associates objects with unsigned integers. Any unsigned integer can work as the object's key, but the system is intended for use with the RTTI system's type identifier. Thus, you register an object into the service container and use its type for subsequent lookup.

### **Listing 12.20** The `ServiceContainer.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class ServiceContainer
    {
public:
    ServiceContainer();

    void AddService(UINT typeID, void* service);
    void RemoveService(UINT typeID);
    void* GetService(UINT typeID) const;

private:
    ServiceContainer(const ServiceContainer& rhs);
    ServiceContainer& operator=(const ServiceContainer&
rhs);

    std::map<UINT, void*> mServices;
    };
}
```

---

[Listing 12.21](#) shows the simple implementation of the ServiceContainer class.

### **Listing 12.21** The ServiceContainer.cpp File

[Click here to view code image](#)

---

```
#include "ServiceContainer.h"

namespace Library
{
    ServiceContainer::ServiceContainer()
        : mServices()
    {

    void ServiceContainer::AddService(UINT typeID, void*
service)
    {
        mServices.insert(std::pair<UINT, void*>(typeID,
service));
    }
```

```
void ServiceContainer::RemoveService(UINT typeID)
{
    mServices.erase(typeID);
}

void* ServiceContainer::GetService(UINT typeID) const
{
    std::map<UINT, void*>::const_iterator it = mServices.
find(typeID);

    return (it != mServices.end() ? it->second : nullptr);
}
```

---

## An Updated Game Class

Because the Game class is central to the rendering engine and accessible by all game components, it works well as the host for the service container. Add a ServiceContainer member to the Library::Game class, and expose it through a public accessor. The companion website has a full implementation.

[Listing 12.22](#) presents a version of the RenderingGame::Initialize() method that registers the mouse and keyboard components with the service container.

### **Listing 12.22** Registering Objects with the Service Container

[Click here to view code image](#)

---

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(mInstance,
DIRECTINPUT_VERSION, IID_
IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    mKeyboard = new Keyboard(*this, mDirectInput);
    mComponents.push_back(mKeyboard);
    mServices.AddService(Keyboard::TypeIdClass(), mKeyboard);

    mMouse = new Mouse(*this, mDirectInput);
    mComponents.push_back(mMouse);
    mServices.AddService(Mouse::TypeIdClass(), mMouse);

    Game::Initialize();
}
```

You can retrieve these services with a call to `ServiceContainer::GetService()`. For example, a component might look up the `Keyboard` service and store it in a class member with a call such as this:

[Click here to view code image](#)

```
mKeyboard = (Keyboard*) mGame->Services().GetService(Keyboard::TypeIdClass());
```

Of course, the service container isn't guaranteed to contain the queried service, and your components should take this into consideration. You can always assert if you don't find a service that is truly required. Furthermore, you can register only a single object with a type identifier. But nothing prevents you from storing a vector of objects with a given ID. You simply need to know how an object was put into the container so that you can correctly take it out.

## Summary

In this chapter, you built a lot of infrastructure for your rendering engine. You learned about game components, text rendering, device input, and software services. You implemented components for collecting keyboard and mouse input, as well as a component for displaying the application's frame rate. You will use these systems in practically all the upcoming demos.

In the next chapter, you write a reusable camera component for viewing objects in your 3D scenes.

## Exercises

1. Experiment with the `SpriteBatch/SpriteFont` system. Create a variety of `.spritefont` objects, and render them in different colors. *Hint:* The `SpriteFont::Draw()` method has an overload that accepts a color.
2. Extend the mouse component to limit the range of XY positions. The mouse positions should not continue to increase or decrease after the mouse has reached the edges of the screen. If the application is in windowed mode, the `(0, 0)` position should represent the upper-left corner of the client area, and the values should not be allowed to go negative, stopping at the upper-left corner of the screen. Likewise, the values should not be allowed to extend past the application resolution to the lower-right corner of the screen. If the application is in full-screen mode, `(0, 0)` marks the minimum position and `(width - 1, height - 1)` marks the maximum.

# Chapter 13. Cameras

In this chapter, you create a reusable camera system to visualize your 3D scenes. You first develop a base camera class to support common functionality, and then you extend the class to create a first-person camera controlled by the mouse and keyboard.

## A Base Camera Component

The topic of a virtual camera is peppered throughout this text, and you can't visualize your 3D scenes without one. But there's no one-size-fits-all camera to meet the needs of all your applications. For example, you might want an orbit camera, similar to what you used in NVIDIA FX Composer. Or perhaps you're building an application that needs a first-person camera, in which you use the mouse to control the pitch and yaw, and you use the W, A, S, and D keys to control movement. Maybe you're building a 2.5D platformer (a side-scroller that's rendered in 3D but viewed with a fixed-axis camera) or an action game with an over-the-shoulder third-person camera that "chases" the avatar with a motion that suggests it's attached with springs. You can implement many types of cameras, but all of them share a base set of functionality that's necessary without respect to the specific behavior of the camera. In this section, you develop a general-purpose camera component that's intended for use as a base class. Then you extend this class to create a first-person camera you can use for future demos.

## Camera Theory Revisited

Recall the discussion of cameras from [Chapter 2, “A 3D/Math Primer.”](#) There you learned that the properties of a camera create a view frustum—sort of a pyramid with the top chopped off—and that only objects within the frustum are visible. A view frustum is created through the camera's position (in world space), a vector describing where the camera is looking, and a vector describing which way is up. Additional properties include the camera's vertical field of view, the aspect ratio (width over height), and the distances of the near and far planes. You can consider these properties as the inputs to your camera. Also recall that these properties define the transformation matrices for moving objects into view space and projection space. Thus, the outputs of a camera can be considered the view and projection matrices, or a combined view-projection matrix.

[Listing 13.1](#) declares the Camera class, which includes members for these inputs and outputs. This class has a full implementation and can be used as is within your applications. Indeed, you'll use this camera in the next chapter to render your first 3D scene. However, it's really intended as a base class and has no functionality for the specific motion of the camera. That type of behavior is delegated to the derived classes.

### **Listing 13.1** The Camera.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "GameComponent.h"
```

```
namespace Library
{
    class GameTime;

    class Camera : public GameComponent
    {
        RTTI_DECLARATIONS(Camera, GameComponent)

public:
    Camera(Game& game);
    Camera(Game& game, float fieldOfView, float aspectRatio,
float
nearPlaneDistance, float farPlaneDistance);

    virtual ~Camera();

    const XMFLOAT3& Position() const;
    const XMFLOAT3& Direction() const;
    const XMFLOAT3& Up() const;
    const XMFLOAT3& Right() const;

    XMVECTOR PositionVector() const;
    XMVECTOR DirectionVector() const;
    XMVECTOR UpVector() const;
    XMVECTOR RightVector() const;

    float AspectRatio() const;
    float FieldOfView() const;
    float NearPlaneDistance() const;
    float FarPlaneDistance() const;

    XMMATRIX ViewMatrix() const;
    XMMATRIX ProjectionMatrix() const;
    XMMATRIX ViewProjectionMatrix() const;

    virtual void SetPosition(FLOAT x, FLOAT y, FLOAT z);
    virtual void SetPosition(FXMVECTOR position);
    virtual void SetPosition(const XMFLOAT3& position);

    virtual void Reset();
    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;
    virtual void UpdateViewMatrix();
    virtual void UpdateProjectionMatrix();
    void ApplyRotation(CXMMATRIX transform);
    void ApplyRotation(const XMFLOAT4X4& transform);
}
```

```

static const float DefaultFieldOfView;
static const float DefaultAspectRatio;
static const float DefaultNearPlaneDistance;
static const float DefaultFarPlaneDistance;

protected:
    float mFieldOfView;
    float mAspectRatio;
    float mNearPlaneDistance;
    float mFarPlaneDistance;

    XMFLOAT3 mPosition;
    XMFLOAT3 mDirection;
    XMFLOAT3 mUp;
    XMFLOAT3 mRight;

    XMFLOAT4X4 mViewMatrix;
    XMFLOAT4X4 mProjectionMatrix;

private:
    Camera(const Camera& rhs);
    Camera& operator=(const Camera& rhs);
};

}

```

Before diving into the implementation, let's examine the structure of the `Camera` class. As you can see, a `Camera` is a game component and can therefore be initialized and updated by the base `Game` class without explicit calls to the associated methods. Instead, you simply add a `Camera` instance to the `Game::mComponents` member. This class is also a good candidate for inclusion in the service container because it is useful to any component that needs to draw 3D objects to the screen.

The `Camera` class has members for the field of view, aspect ratio, and near and far plane distances, and a constructor accepts arguments for these values. A default constructor also uses the quantities stored in the static members: `DefaultFieldOfView`, `DefaultAspectRatio`, `DefaultNearPlaneDistance`, and `DefaultFarPlaneDistance`.

Three members define the orientation of the camera in three-dimensional space: `mDirection`, `mUp`, and `mRight`. These vectors are orthogonal to each other and are rotated in concert. (More specifically, the direction and up vectors are rotated, and the right vector is derived through a cross product of the other two vectors.) The `mPosition` member stores the location of the camera.

The two *outputs* of the camera are the `mViewMatrix` and `mProjectionMatrix` members. We describe the Direct3D methods, used to update these values, after the code presentation. [Listing 13.2](#) presents an abbreviated implementation of `Camera` class. For brevity, the listing omits most of the single-line accessors and mutators. You can find the full source code on the book's companion website.

## **Listing 13.2** The Camera Class Implementation (Abbreviated)

```
#include "Camera.h"
#include "Game.h"
#include "GameTime.h"
#include "VectorHelper.h"
#include "MatrixHelper.h"

namespace Library
{
    RTTI_DEFINITIONS(Camera)

    const float Camera::DefaultFieldOfView = XM_PIDIV4;
    const float Camera::DefaultNearPlaneDistance = 0.01f;
    const float Camera::DefaultFarPlaneDistance = 1000.0f;

    Camera::Camera(Game& game)
        : GameComponent(game),
          mFieldOfView(DefaultFieldOfView),
          mAspectRatio(game.AspectRatio()),
          mNearPlaneDistance(DefaultNearPlaneDistance),
          mFarPlaneDistance(DefaultFarPlaneDistance),
          mPosition(), mDirection(), mUp(), mRight(),
          mViewMatrix(), mProjectionMatrix()
    {
    }

    XMMATRIX Camera::ViewProjectionMatrix() const
    {
        XMMATRIX viewMatrix = XMLoadFloat4x4(&mViewMatrix);
        XMMATRIX projectionMatrix =
        XMLoadFloat4x4(&mProjectionMatrix);

        return XMMatrixMultiply(viewMatrix, projectionMatrix);
    }

    void Camera::SetPosition(FLOAT x, FLOAT y, FLOAT z)
    {
        XMVECTOR position = XMVectorSet(x, y, z, 1.0f);
        SetPosition(position);
    }

    void Camera::SetPosition(FXMVECTOR position)
    {
        XMStoreFloat3(&mPosition, position);
    }
```

```
void Camera::SetPosition(const XMFLOAT3& position)
{
    mPosition = position;
}

void Camera::Reset()
{
    mPosition = Vector3Helper::Zero;
    mDirection = Vector3Helper::Forward;
    mUp = Vector3Helper::Up;
    mRight = Vector3Helper::Right;

    UpdateViewMatrix();
}

void Camera::Initialize()
{
    UpdateProjectionMatrix();
    Reset();
}

void Camera::Update(const GameTime& gameTime)
{
    UpdateViewMatrix();
}

void Camera::UpdateViewMatrix()
{
    XMVECTOR eyePosition = XMLoadFloat3(&mPosition);
    XMVECTOR direction = XMLoadFloat3(&mDirection);
    XMVECTOR upDirection = XMLoadFloat3(&mUp);

    XMMATRIX viewMatrix = XMMatrixLookToRH(eyePosition,
direction,
upDirection);
    XMStoreFloat4x4(&mViewMatrix, viewMatrix);
}

void Camera::UpdateProjectionMatrix()
{
    XMMATRIX projectionMatrix = XMMatrixPerspectiveFovRH
(mFieldOfView, mAspectRatio, mNearPlaneDistance,
mFarPlaneDistance);
    XMStoreFloat4x4(&mProjectionMatrix, projectionMatrix);
}

void Camera::ApplyRotation(CXMMATRIX transform)
```

```

    XMVECTOR direction = XMLoadFloat3(&mDirection);
    XMVECTOR up = XMLoadFloat3(&mUp);

    direction = XMVector3TransformNormal(direction,
transform);
    direction = XMVector3Normalize(direction);

    up = XMVector3TransformNormal(up, transform);
    up = XMVector3Normalize(up);

    XMVECTOR right = XMVector3Cross(direction, up);
    up = XMVector3Cross(right, direction);

    XMStoreFloat3(&mDirection, direction);
    XMStoreFloat3(&mUp, up);
    XMStoreFloat3(&mRight, right);
}

void Camera::ApplyRotation(const XMFLOAT4X4& transform)
{
    XMMATRIX transformMatrix = XMLoadFloat4x4(&transform);
    ApplyRotation(transformMatrix);
}

```

---

## DirectXMath Usage

The first area to examine is the DirectXMath usage found throughout the `Camera` class implementation. For example, the `Camera::ViewProjectionMatrix()` accessor first loads the `mViewMatrix` and `mProjectionMatrix` members into `XMMATRIX` variables and then multiplies them with the `XMMatrixMultiply()` function. Recall the discussion of DirectXMath from [Chapter 2](#) and the performance benefits of SIMD instructions. Matrix and vector operations should use SIMD variables, but SIMD types are 16-byte aligned and are therefore not good candidates for class member storage. Thus, you store the view and projection matrices as `XMFLOAT4X4` class members and *load* them into `XMMATRIX` variables to use them. Likewise, you store three-component vectors with `XMFLOAT3` class members and load them into `XMVECTOR` objects for calculations.

Conversely, if you need to save computed SIMD variables back to associated class members, you do so with *store* calls, such as `XMStoreFloat3()` and `XMStoreFloat4×4()`. You can see this usage in the `Camera::UpdateViewMatrix()` and `Camera::ApplyRotation()` methods.

### Note

An alternative to using DirectX Math load and store methods is to align your classes and structs along 16-byte boundaries with `__declspec(align(16))`. You can find more information on data alignment at <http://msdn.microsoft.com/en-us/library/2e754c56.aspx>

## The **Reset()** Method

The `Camera::Reset()` method sets the position and orientation members to reasonable values and then invokes the `Camera::UpdateViewMatrix()` methods. Updating the `mViewMatrix` member is necessary whenever the position or orientation of the camera changes. As with most of the `Camera` methods, the `Reset()` method is virtual and can be customized by a derived class.

The `Reset()` method uses a `Vector3Helper` class that has a number of useful static `XMFLOAT3` members. Analogous `Vector2Helper` and `Vector4Helper` classes exist for 2D and 4D vectors. You can find these helper classes in the `VectorHelper.h` header file on the book's companion website.

## The **UpdateViewMatrix()** Method

The `Camera::UpdateViewMatrix()` method uses the `XMMatrixLookToRH()` method from the DirectXMath library. This method creates a view matrix for a right-handed coordinate system. An `XMMatrixLookToLH()` method creates a view matrix for a left-handed coordinate system. Both methods accept three `XMFLOAT3` arguments for the camera's position, direction, and up vector. These values are loaded from the associated class members.

Note that the `Camera::Update()` method invokes the `UpdateViewMatrix()` method each frame. You could consider an optimization that checked a "dirty" status to opt out of the view matrix calculation if the camera hasn't moved.

## The **UpdateProjectionMatrix()** Method

Along with helper methods for computing the view matrix, the DirectXMath library includes methods for computing the projection matrix. For perspective projection, the camera employs `XMMatrixPerspectiveFovRH()`, which accepts arguments for the field of view, aspect ratio, and near and far plane distances. This method also has a left-handed version.

Whereas the view matrix is expected to update regularly, the projection matrix is not. Indeed, it's not uncommon to set the projection matrix once (at camera initialization) and not change it again throughout the program's execution. That's the approach taken here. Note that the associated class members are protected and have no public mutators. However, a derived class could expose these members or otherwise allow modification of the projection matrix post-initialization.

## The **ApplyRotation()** Method

Although the base `Camera` class doesn't provide implicit behavior for the camera's movement, it does include an `ApplyRotation()` method. This method applies a rotation matrix to the orientation of the camera. This is accomplished by transforming the direction and up vectors by the rotation matrix and then computing the right vector as the cross product of the other two vectors. A second cross product exists, which might seem rather strange. Immediately after the right vector is computed, the up vector is recalculated as a cross product of the right and direction vectors. This step is intended to eliminate any computation error and guarantee that the three vectors are orthogonal.

Note that any translation passed into the `ApplyRotation()` method does not modify the camera's position. Instead, you can change the camera's position through the overloaded `SetPosition()`

methods.

## A First-Person Camera

Now let's extend the base `Camera` class to implement a first-person camera. This camera is controlled through a combination of the mouse and keyboard. The W and S keys move the camera forward and backward along its direction vector. The A and D keys “strafe” (move the camera horizontally) along the right vector. The mouse controls the yaw and pitch of the camera: You move the mouse vertically to pitch, and horizontally to yaw. Note that pitch rotation is performed around the camera’s right vector, and yaw is around the y-axis. This is the traditional behavior of a first-person camera, but it can’t accommodate, for example, a game set in space in which the camera would require roll (longitudinal rotation) as well.

[Listing 13.3](#) presents the header file for the `FirstPersonCamera` class.

### **Listing 13.3** The `FirstPersonCamera.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Camera.h"

namespace Library
{
    class Keyboard;
    class Mouse;

    class FirstPersonCamera : public Camera
    {
        RTTI_DECLARATIONS(FirstPersonCamera, Camera)

    public:
        FirstPersonCamera(Game& game);
        FirstPersonCamera(Game& game, float fieldOfView, float
aspectRatio, float nearPlaneDistance, float farPlaneDistance);

        virtual ~FirstPersonCamera();

        const Keyboard& GetKeyboard() const;
        void SetKeyboard(Keyboard& keyboard);

        const Mouse& GetMouse() const;
        void SetMouse(Mouse& mouse);

        float& MouseSensitivity();
        float& RotationRate();
        float& MovementRate();
    };
}
```

```

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;

    static const float DefaultMouseSensitivity;
    static const float DefaultRotationRate;
    static const float DefaultMovementRate;

protected:
    float mMouseSensitivity;
    float mRotationRate;
    float mMovementRate;

    Keyboard* mKeyboard;
    Mouse* mMouse;

private:
    FirstPersonCamera(const FirstPersonCamera& rhs);
    FirstPersonCamera& operator=(const FirstPersonCamera&
rhs);
};

}

```

---

The `FirstPersonCamera` class has members for the mouse and keyboard, and it exposes them through public accessors and mutators. However, as you see in the implementation, the `Initialize()` method attempts to find the mouse and keyboard within the service container. The mutators simply enable you to override these settings and even disable mouse or keyboard control by nullifying these values.

The class also contains members for movement and rotation rates and mouse sensitivity. The “rate” variables specify how many units the camera should translate or rotate in 1 second. The mouse sensitivity enables you to amplify or dampen the input of the mouse.

[Listing 13.4](#) presents the more interesting aspects of the camera implementation. Complete source code is available on the companion website.

#### Listing 13.4 The `FirstPersonCamera` Class Implementation (Abbreviated)

[Click here to view code image](#)

---

```

const float FirstPersonCamera::DefaultRotationRate =
XMConvertToRadians(1.0f);
const float FirstPersonCamera::DefaultMovementRate = 10.0f;
const float FirstPersonCamera::DefaultMouseSensitivity = 100.0f;

void FirstPersonCamera::Initialize()
{
    mKeyboard = (Keyboard*)mGame-

```

```

>Services().GetService<Keyboard::Type>
IdClass());
    mMouse = (Mouse*)mGame->Services().GetService
(Mouse::TypeIdClass());
}

    Camera::Initialize();
}

void FirstPersonCamera::Update(const GameTime& gameTime)
{
    XMFLOAT2 movementAmount = Vector2Helper::Zero;
    if (mKeyboard != nullptr)
    {
        if (mKeyboard->IsKeyDown(DIK_W))
        {
            movementAmount.y = 1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_S))
        {
            movementAmount.y = -1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_A))
        {
            movementAmount.x = -1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_D))
        {
            movementAmount.x = 1.0f;
        }
    }

    XMFLOAT2 rotationAmount = Vector2Helper::Zero;
    if ((mMouse != nullptr) && (mMouse->IsButtonHeldDown
(MouseButtonsLeft)))
    {
        LPDIMOUSESTATE mouseState = mMouse->CurrentState();
        rotationAmount.x = -mouseState->lX * mMouseSensitivity;
        rotationAmount.y = -mouseState->lY * mMouseSensitivity;
    }

    float elapsedTime = (float)gameTime.ElapsedGameTime();
    XMVECTOR rotationVector = XMLoadFloat2(&rotationAmount) *
mRotationRate * elapsedTime;
    XMVECTOR right = XMLoadFloat3(&mRight);
}

```

```

XMMATRIX pitchMatrix = XMMatrixRotationAxis(right,
XMVectorGetY(rotationVector));
XMMATRIX yawMatrix = XMMatrixRotationY(XMVectorGetX
(rotationVector));

ApplyRotation(XMMatrixMultiply(pitchMatrix, yawMatrix));

XMVECTOR position = XMLoadFloat3(&mPosition);
XMVECTOR movement = XMLoadFloat2(&movementAmount) *
mMovementRate *
elapsedTime;

XMVECTOR strafe = right * XMVectorGetX(movement);
position += strafe;

XMVECTOR forward = XMLoadFloat3(&mDirection) *
XMVectorGetY(movement);
position += forward;

XMStoreFloat3(&mPosition, position);

Camera::Update(gameTime);
}

```

---

The default values for the rotation rate, movement rate, and mouse sensitivity are set at the beginning of [Listing 13.4](#). Note the `XMConvertToRadians()` function: This is a DirectXMath helper function that converts degrees to radians. `XMConvertToDegrees()` is also present to go the other way.

The `Initialize()` method demonstrates the retrieval of the mouse and keyboard services from the service container. Note that these services potentially might not exist. In that case, the returned value is `NULL`. This component is written so that it doesn't require the keyboard or mouse to function. Otherwise, you want to assert that the objects are present in the service container.

The `Update()` method contains the bulk of the implementation. It first tests whether the W, S, A, and D keys are pressed; if so, it saves off the value of 1 or -1 to denote the direction of movement. A keyboard is digital—its keys are either pressed or not—and therefore cannot express a variable quantity such as an analog trigger or thumbstick on a gamepad. Thus, the `mMovementRate` class member defines the magnitude of the movement (although it would be reasonable to scale the magnitude by the length of time the key was held down).

Next, the rotation amounts are collected from the mouse (if the left mouse button is held down). These amounts are multiplied by the `mRotationRate` member and the elapsed time of the frame. Notice that the `rotationAmount` variable is loaded into an `XMVECTOR` object, before the multiplication, to take advantage of SIMD operations. The elapsed time element is included to make the computation independent of the frame rate.

With the rotation amounts in hand, you construct the yaw and pitch rotation matrices you'll apply to

the camera. The pitch matrix is created with a call to `XMMatrixRotationAxis()`, which builds a matrix that rotates around an arbitrary axis. In this case, the axis is the camera's right vector. Note the `XMVectorGetY()` function used in the second argument to the `XMMatrixRotationAxis()` call. This retrieves the Y component from the opaque `XMVECTOR` object. The yaw matrix is created through `XMMatrixRotationY()`, which builds a rotation matrix around the y-axis. The DirectXMath library has analogous calls for building rotation matrices around the x- and z-axes. The yaw and pitch matrices are concatenated through `XMMatrixMultiply()` before being passed to the `Camera::ApplyRotation()` method.

Next, the movement amount is modulated by the `mMovementRate` member and the elapsed time and is stored in a vector. Then the strafe vector is calculated by multiplying the camera's right vector by the X component of the movement vector. This quantity is added to the camera's position. The process is repeated for forward movement, and the final position is written back to the `Camera::mPosition` member.

## Summary

In this chapter, you developed a base component to support common aspects of a virtual camera. Then you extended this class to create a first-person camera controlled by the mouse and keyboard. Along the way, you exercised a bit more of the component and services scaffolding that you created in the last chapter, and you revisited some 3D camera theory and DirectX math usage.

All this work is about to pay off. In the next chapter, you begin rendering 3D content to the screen.

## Exercise

1. Create an orbit camera similar to the one found in NVIDIA FX Composer. *Note:* Visualizing the output of your camera will be difficult without any 3D rendering (which the next chapter covers). You can use the `SpriteBatch/SpriteFont` system to output your camera's data or employ the `Grid` component on the companion website. This component renders a customizable reference grid at the world origin.

# Chapter 14. Hello, Rendering!

You've made it! You've completed the application framework, and you have enough code to start rendering. In this chapter, you implement your first full 3D applications. You examine numerous Direct3D and Effects 11 API calls. And when you're finished, you'll have a deeper understanding of the Direct3D graphics pipeline.

## Your First Full Rendering Application

It's now time to render your first 3D object to the screen: a triangle. It seems only fitting to call this application "Hello, Rendering" because it's your first full Direct3D program. But the name belies its complexity because, as you've discovered over the last few chapters, a lot of infrastructure is required just to render a single triangle. Furthermore, you encounter a lot of new code in this section. This is far from a single-line "Hello, World!". Let's get started.

## An Updated Game Project

You are welcome to continue using the Game project you've been developing or to create a new one. You'll make no modifications to the structure of the Game project; you'll have the same Program.cpp file that instantiates a RenderingGame object and invokes its Run() method. The implementation of the RenderingGame class is the same, with two exceptions: the inclusion of the FirstPersonCamera and a TriangleDemo class member. The TriangleDemo class will derive from DrawableGameComponent and will contain all the code for rendering a triangle. With these new members, the implementation of your RenderingGame class should resemble [Listing 14.1](#).

### **Listing 14.1** The RenderingGame.cpp File

[Click here to view code image](#)

```
#include "RenderingGame.h"
#include "GameException.h"
#include "Keyboard.h"
#include "Mouse.h"
#include "FpsComponent.h"
#include "ColorHelper.h"
#include "FirstPersonCamera.h"
#include "TriangleDemo.h"

namespace Rendering
{
    const XMVECTORF32 RenderingGame::BackgroundColor =
        ColorHelper::CornflowerBlue;

    RenderingGame::RenderingGame(HINSTANCE instance, const
        std::wstring& windowClass, const std::wstring& windowTitle, int
        showCommand)
```

```
: Game(instance, windowClass, windowTitle,
showCommand),
    mFpsComponent(nullptr),
    mDirectInput(nullptr), mKeyboard(nullptr),
mMouse(nullptr),
    mDemo(nullptr)
{
    mDepthStencilBufferEnabled = true;
    mMutiSamplingEnabled = true;
}

RenderingGame::~RenderingGame()
{
}

void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(mInstance,
DIRECTINPUT_VERSION, IID_IDirectInput8, (LPVOID*)&mDirectInput,
nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    mKeyboard = new Keyboard(*this, mDirectInput);
    mComponents.push_back(mKeyboard);
    mServices.AddService(Keyboard::TypeIdClass(),
mKeyboard);

    mMouse = new Mouse(*this, mDirectInput);
    mComponents.push_back(mMouse);
    mServices.AddService(Mouse::TypeIdClass(), mMouse);

    mCamera = new FirstPersonCamera(*this);
    mComponents.push_back(mCamera);
    mServices.AddService(Camera::TypeIdClass(), mCamera);

    mFpsComponent = new FpsComponent(*this);
    mComponents.push_back(mFpsComponent);

    mDemo = new TriangleDemo(*this, *mCamera);
    mComponents.push_back(mDemo);

    Game::Initialize();

    mCamera->SetPosition(0.0f, 0.0f, 5.0f);
}
```

```

void RenderingGame::Shutdown()
{
    DeleteObject(mDemo);
    DeleteObject(mKeyboard);
    DeleteObject(mMouse);
    DeleteObject(mFpsComponent);
    DeleteObject(mCamera);

    ReleaseObject(mDirectInput);

    Game::Shutdown();
}

void RenderingGame::Update(const GameTime &gameTime)
{
    if (mKeyboard->WasKeyPressedThisFrame(DIK_ESCAPE))
    {
        Exit();
    }

    Game::Update(gameTime);
}

void RenderingGame::Draw(const GameTime &gameTime)
{
    mDirect3DDeviceContext-
>ClearRenderTargetView(mRenderTarget
View, reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext-
>ClearDepthStencilView(mDepthStencil
View, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    Game::Draw(gameTime);

    HRESULT hr = mSwapChain->Present(0, 0);
    if (FAILED(hr))
    {
        throw GameException("IDXGISwapChain::Present()
failed.", hr);
    }
}

```

The `RenderingGame::Initialize()` method now instantiates the `mCamera` and `mDemo` class members and adds them to the components container. Additionally, a `ColorHelper` class is introduced that initializes the background color. This class contains a few static members for common

colors (of type XMVECTORF32) and isn't listed here, but the companion website does include it. A final note is for the FirstPersonCamera::SetPosition() call at the end of the Initialize() method. This moves the camera five units along the positive z-axis. You'll be creating a triangle at the origin, and this call moves the camera so that the triangle is immediately visible upon program execution. This demo uses a right-handed coordinate system, so positive z is headed out of the screen.

## A TriangleDemo Component

[Listing 14.2](#) presents the declaration of the TriangleDemo class. Notice the BasicEffectVertex structure and the class members that begin with ID3DX11Effect. The former defines the structure of the triangle's vertices and what data will be passed as shader input. The latter refers to types from the Effects 11 library, which [Chapter 3, “Tools of the Trade,”](#) introduced. Extract this library to the external directory if you haven't already done so. Refer to the “[Linking Libraries](#)” section of [Chapter 10, “Project Setup and Window Initialization,”](#) if you have difficulty referencing this package. Shaders are required for all rendering in Direct3D. Thus, such class members are necessary regardless of the complexity of your scene.

A class member of type ID3DInputLayout also stores the definition of vertex data for the input-assembler pipeline stage, and a class member of type ID3D11Buffer exists for the vertex buffer. We discuss all these items after the code listing.

### **Listing 14.2** The TriangleDemo.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "DrawableGameComponent.h"

using namespace Library;

namespace Rendering
{
    class TriangleDemo : public DrawableGameComponent
    {
        RTTI_DECLARATIONS(TriangleDemo, DrawableGameComponent)

    public:
        TriangleDemo(Game& game, Camera& camera);
        ~TriangleDemo();

        virtual void Initialize() override;
        virtual void Draw(const GameTime& gameTime) override;

    private:
        typedef struct _BasicEffectVertex
        {
```

```

XMFLOAT4 Position;
XMFLOAT4 Color;

_BasicEffectVertex() { }

_BasicEffectVertex(XMFLOAT4 position, XMFLOAT4
color)
    : Position(position), Color(color) { }
} BasicEffectVertex;

TriangleDemo();
TriangleDemo(const TriangleDemo& rhs);
TriangleDemo& operator=(const TriangleDemo& rhs);

ID3DX11Effect* mEffect;
ID3DX11EffectTechnique* mTechnique;
ID3DX11EffectPass* mPass;
ID3DX11EffectMatrixVariable* mWvpVariable;

ID3D11InputLayout* mInputLayout;
ID3D11Buffer* mVertexBuffer;

XMFLOAT4X4 mWorldMatrix;
};

}

```

## The Effects 11 Library

The Effects 11 library provides a set of C++ interfaces for the effect file format (the .fx format you've been using for all your shaders). The root interface is `ID3DX11Effect`, which encapsulates (among other things) the effect's variables and techniques. A technique is exposed through the `ID3DX11EffectTechnique` interface, and it contains one or more passes, represented by the `ID3DX11EffectPass` interface. An effect's variables are represented by the `ID3DX11EffectVariable` interface, but a set of derived classes also exists for specific types of shader constant. For example, the `TriangleDemo::mWvpVariable` class member is of type `ID3DEffectMatrixVariable` and is used to access the WorldViewProjection matrix within the effect. All types within the Effects 11 library are found in the `d3dx11Effect.h` header file.

## Compiling an Effect

An effect must be compiled before it can be used. You can do this when the Visual Studio project is built, or you can compile the effect at runtime. In the next chapter, you learn how to compile effects at build time, but this section covers runtime effect compilation. You'll compile an effect using the `D3DCompileFromFile()` function, found in the `D3DCompiler.h` header file. This function's prototype and parameters are listed here:

[Click here to view code image](#)

```

HRESULT D3DCompileFromFile(
    LPCWSTR pFileName,
    D3D_SHADER_MACRO* pDefines,
    ID3DInclude* pInclude,
    LPCSTR pEntryPoint,
    LPCSTR pTarget,
    UINT Flags1,
    UINT Flags2,
    ID3DBlob** ppCode,
    ID3DBlob** ppErrorMsgs);

```

- **pFileName:** The name of the file containing the effect.
- **pDefines:** An optional array of shader macros defined by the `D3D_SHADER_MACRO` structure. Your effects aren't using any shader macros, so you can set this parameter to `NULL`.
- **pInclude:** An optional `ID3DInclude` pointer that directs the application to call user-defined methods for opening and closing `#include` files. If you set this parameter to `NULL`, the compilation will fail on effects with `#include` directives. You can pass the `D3D_COMPILE_STANDARD_FILE_INCLUDE` macro to specify the default include handler. The default handler includes files relative to the current directory, so it's important to set the current working directory to the correct location before compiling effects with this option.
- **pEntryPoint:** The name of the shader's entry point function. This always is set to `NULL` when compiling effect files.
- **pTarget:** The compiler's shader target. You'll specify `fx_5_0` for all your effects.
- **Flags1:** Bitwise OR'd flags from the `D3DCOMPILE` enumeration that specify how your shader code is compiled. Common options include `D3DCOMPILE_DEBUG` and `D3DCOMPILE_SKIP_OPTIMIZATIONS`, which are useful for debugging shaders. Visit the online documentation for a full list of options.
- **Flags2:** Bitwise OR'd flags from the `D3DCOMPILE_EFFECT` enumeration that specify how your effect file is compiled. These are for advanced features and aren't commonly employed.
- **ppCode:** The compiled code returned as a pointer to an `ID3DBlob` object. The `ID3DBlob` interface describes a buffer of arbitrary length. It has only two methods: `ID3DBlob::GetBufferPointer()` and `ID3DBlob::GetBufferSize()`. They return a pointer to the data and the size (in bytes) of the data, respectively.
- **ppErrorMsgs:** An optional parameter for storing compiler error messages.

[Listing 14.3](#) demonstrates the `D3DCompileFromFile()` function.

### Listing 14.3 Compiling an Effect

[Click here to view code image](#)

---

```

SetCurrentDirectory(Utility::ExecutableDirectory().c_str());
UINT shaderFlags = 0;

```

```

#ifndef DEBUG
#define _DEBUG
#endif

ID3D10Blob* compiledShader = nullptr;
ID3D10Blob* errorMessages = nullptr;
HRESULT hr =
D3DCompileFromFile(L"Content\\Effects\\BasicEffect.fx",
nullptr, nullptr, nullptr, "fx_5_0", shaderFlags, 0,
&compiledShader,
&errorMessages);
if (errorMessages != nullptr)
{
    GameException ex((char*)errorMessages->GetBufferPointer(),
hr);
    ReleaseObject(errorMessages);

    throw ex;
}

if (FAILED(hr))
{
    throw GameException("D3DX11CompileFromFile() failed.", hr);
}

```

This code compiles the BasicEffect.fx file from the Content\Effects\ folder relative to the directory housing the application's executable. The compiled effect is returned through the compiledShader variable, and any error messages are returned through the errorMessages variable.

The BasicEffect.fx file is a derivation of the HelloStructs.fx effect you wrote back in [Chapter 4](#). It accepts a vertex position and color for shader inputs, transforms the position to homogenous clip space, and outputs the vertex color, interpolated per pixel. [Listing 14.4](#) presents the complete effect. Note the use of the technique11 keyword to specify a DirectX 11 technique.

#### **Listing 14.4** The BasicEffect.fx File

[Click here to view code image](#)

```

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

struct VS_INPUT

```

```

{
    float4 ObjectPosition: POSITION;
    float4 Color : COLOR;
};

struct VS_OUTPUT
{
    float4 Position: SV_Position;
    float4 Color : COLOR;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.Color = IN.Color;

    return OUT;
}

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return IN.Color;
}

technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}

```

---

You should store this file under the Content\Effects\ directory of either your Game or your Library project. Be certain that you have the proper pre- and post-build events set up to copy this content to the executable directory. See [Chapter 12](#) for details on these build events.

## Creating an Effect Object

With the effect compiled, you can use the resulting `ID3DBlob` to create an `ID3DX11Effect` object. You do so through the `D3DX11CreateEffectFromMemory()` function. The prototype and parameters are listed next:

[Click here to view code image](#)

```
HRESULT D3DX11CreateEffectFromMemory(  
    LPCVOID pData,  
    SIZE_T DataLength,  
    UINT FXFlags,  
    ID3D11Device *pDevice,  
    ID3DX11Effect **ppEffect );
```

- **pData:** The compiled effect.
- **DataLength:** The size (in bytes) of the compiled effect.
- **FXFlags:** Currently, no effect flags exist. This parameter always is set to 0.
- **pDevice:** The Direct3D device.
- **ppEffect:** The created effect instance.

[Listing 14.5](#) presents an example of the `D3DX11CreateEffectFromMemory()` call. Note the release of the compiled shader blob. After the effect object has been created, there is little use for the compiled shader, so its reference is released.

### **Listing 14.5** Creating an Effect Object

[Click here to view code image](#)

---

```
hr = D3DX11CreateEffectFromMemory(compiledShader-  
    >GetBufferPointer(), compiledShader->GetBufferSize(), 0, mGame-  
    >Direct3DDevice(), &mEffect);  
if (FAILED(hr))  
{  
    throw GameException("D3DX11CreateEffectFromMemory()  
failed.", hr);  
}  
  
ReleaseObject(compiledShader);
```

---

## Technique, Pass, and Variable Lookup

With an effect object created, you can find the technique, pass, and variables to use. The methods for accomplishing these tasks share a consistent naming convention:

`ID3DX11Effect::GetTechniqueByName()`,

`ID3DX11EffectTechnique::GetPassByName()`, and

`ID3DX11Effect::GetVariableByName()`. All three methods have exactly one parameter, the string representing the value to find; they also each return the corresponding interface, or `NULL` if

the identifier is not found.

The `ID3DX11EffectVariable` interface has a set of “As” methods that cast the variable to a more specific interface. For example, the `ID3DX11EffectVariable::AsMatrix()` method attempts to cast the object to the `ID3DX11EffectMatrixVariable` interface. You must test the success of this method with a call to `ID3DX11EffectVariable::IsValid()` against the cast instance.

[Listing 14.6](#) demonstrates the calls to find the `main11` technique, the `p0` pass, and the `WorldViewProjection` variable from the `BasicEffect.fx` file. It also includes an example call for casting the `WorldViewProjection` variable to the `ID3DX11EffectMatrixVariable` interface.

## Listing 14.6 Retrieving Techniques, Passes, and Variables

[Click here to view code image](#)

```
mTechnique = mEffect->GetTechniqueByName("main11");
if (mTechnique == nullptr)
{
    throw GameException("ID3DX11Effect::GetTechniqueByName()
could not
find the specified technique.", hr);
}

mPass = mTechnique->GetPassByName("p0");
if (mPass == nullptr)
{
    throw GameException("ID3DX11EffectTechnique::GetPassByName()
could
not find the specified pass.", hr);
}

ID3DX11EffectVariable* variable = mEffect->GetVariableByName
("WorldViewProjection");
if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName()
could not
find the specified variable.", hr);
}

mWvpVariable = variable->AsMatrix();
if (mWvpVariable->IsValid() == false)
{
    throw GameException("Invalid effect variable cast.");
}
```

# Creating an Input Layout

Following is the VS\_INPUT structure of the BasicEffect.fx file and the BasicEffectVertex structure from the TriangleDemo class:

[Click here to view code image](#)

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

typedef struct _BasicEffectVertex
{
    XMFLOAT4 Position;
    XMFLOAT4 Color;

    _BasicEffectVertex() { }

    _BasicEffectVertex(XMFLOAT4 position, XMFLOAT4 color)
        : Position(position), Color(color) { }
} BasicEffectVertex;
```

These structures both represent the same vertex data (a position and a color), the BasicEffectVertex structure from the C++/CPU side, and the VS\_INPUT structure from the HLSL/GPU side. Direct3D requires an **input layout** to map the vertex data from the CPU to the GPU. You create an input layout by first configuring an array of D3D11\_INPUT\_ELEMENT\_DESC instances. This structure has the following definition:

[Click here to view code image](#)

```
typedef struct D3D11_INPUT_ELEMENT_DESC
{
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D11_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D11_INPUT_ELEMENT_DESC;
```

- **SemanticName:** The HLSL semantic associated with the input element (for example, POSITION or COLOR).
- **SemanticIndex:** The semantic index for the element (for example, 1 for TEXCOORD1 and 2 for COLOR2). Indices are not required unless the same semantic is assigned to more than one input element. For example, the semantic POSITION is equivalent to POSITION0.
- **Format:** The type of data in the input element. For example,

`DXGI_FORMAT_R32G32B32A32_FLOAT` describes a format with four 32-bit floating-point channels.

- **InputSlot:** Depending on the feature level, there are 16 or 32 available input slots through which your vertex data can be sent. These slots accommodate multiple vertex buffers, where each vertex buffer is assigned a different slot.
- **AlignedByteOffset:** The offset (in bytes) between each input element. Use `D3D11_APPEND_ALIGNED_ELEMENT` to autoalign the elements.
- **InputSlotClass:** Input data specification as either per-vertex or per-instance. [Part IV, “Intermediate-Level Rendering Topics,”](#) discusses hardware instancing. For now, you’ll use `D3D_INPUT_PER_VERTEX_DATA`.
- **InstanceDataStepRate:** This parameter is used in conjunction with per-instance data (an **InputSlotClass** specified as `D3D_INPUT_PER_INSTANCE_DATA`). For per-vertex data, this parameter always is set to 0.

With an array of input element descriptions, you invoke the `ID3D11Device::CreateInputLayout()` method; the prototype and parameters are listed here:

[Click here to view code image](#)

```
HRESULT CreateInputLayout(
    const D3D11_INPUT_ELEMENT_DESC *pInputElementDescs,
    UINT NumElements,
    const void *pShaderBytecodeWithInputSignature,
    SIZE_T BytecodeLength,
    ID3D11InputLayout **ppInputLayout);
```

- **pInputElementDescs:** An array of input element descriptions.
- **NumElements:** The number of elements in **pInputElementDescs**.
- **pShaderBytecodeWithInputSignature:** The compiled shader (contains an input signature that is validated against the input element descriptions).
- **BytecodeLength:** The size (in bytes) of the compiled shader.
- **ppInputLayout:** The created input layout.

[Listing 14.7](#) demonstrates the creation of the input layout for the triangle demo.

### **Listing 14.7** Input Layout Creation

[Click here to view code image](#)

---

```
D3DX11_PASS_DESC passDesc;
mPass->GetDesc(&passDesc);

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

```

    { "COLOR",      0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
D3D11_APPEND_
ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

if (FAILED(hr = mGame->Direct3DDevice()->CreateInputLayout
(inputElementDescriptions, ARRAYSIZE(inputElementDescriptions),
passDesc.pIAInputSignature,
passDesc.IAInputSignatureSize, &mInputLayout)))
{
    throw GameException("ID3D11Device::CreateInputLayout()
failed.",
hr);
}

```

---

The first two lines of [Listing 14.7](#) get a pass description structure. This data type contains the input signature that the `ID3D11Device::CreateInputLayout()` method requires. The input signature is associated with a pass because a pass specifies which vertex shader to use and, hence, what shader inputs to use. Next, an array of input element descriptions is created for the POSITION and COLOR inputs. Finally, the input layout object is created.

## Creating a Vertex Buffer

The last step in the initialization of the `TriangleDemo` class is the creation of a vertex buffer. A vertex buffer stores the vertices processed by the Direct3D graphics pipeline and is represented by the `ID3D11Buffer` interface. To create a vertex buffer, you first configure a `D3D11_BUFFER_DESC` structure which is defined as follows:

[Click here to view code image](#)

```

typedef struct D3D11_BUFFER_DESC
{
    UINT ByteWidth;
    D3D11_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
    UINT StructureByteStride;
} D3D11_BUFFER_DESC;

```

■ **ByteWidth:** The size of the buffer.

■ **Usage:** How the buffer will be read from and written to. Set this parameter to `D3D11_USAGE_IMMUTABLE` if the vertex buffer won't be modified after creation. Only the GPU can access immutable buffers (read, not write); the CPU can't access them at all.

■ **BindFlags:** Set to `D3D11_BIND_VERTEX_BUFFER` for vertex buffers.

■ **CPUAccessFlags:** Bitwise OR'd values that specify whether the CPU is allowed to read or write to the texture. Use 0 when no CPU access is required.

■ **MiscFlags:** Flags for less commonly used buffer options.

■ **StructureByteStride:** The size of an element when the buffer is a **structured buffer**. A structured buffer contains elements of equal sizes. This is set to 0 for all the demos in this text.

Next, you populate a D3D11\_SUBRESOURCE\_DATA structure to provide initial data to the vertex buffer. For immutable buffers (buffers with a usage of D3D11\_USAGE\_IMMUTABLE), data must be provided when the buffer is created. Otherwise, this is an optional step. The D3D11\_SUBRESOURCE\_DATA structure has the following definition:

[Click here to view code image](#)

```
typedef struct D3D11_SUBRESOURCE_DATA
{
    const void *pSysMem;
    UINT SysMemPitch;
    UINT SysMemSlicePitch;
} D3D11_SUBRESOURCE_DATA;
```

■ **pSysMem:** A pointer to the initialization data.

■ **SysMemPitch:** The length (in bytes) of one line of a texture. This parameter applies only to 2D and 3D textures and has no bearing on vertex buffers.

■ **SysMemSlicePitch:** The distance (in bytes) from one depth level to the next. This parameter is used only for 3D textures.

The final step for creating a vertex buffer is to invoke the method ID3D11Device::CreateBuffer().

[Click here to view code image](#)

```
HRESULT CreateBuffer(
    const D3D11_BUFFER_DESC *pDesc,
    const D3D11_SUBRESOURCE_DATA *pInitialData,
    ID3D11Buffer **ppBuffer);
```

■ **pDesc:** The buffer description structure

■ **pInitialData:** The (optional) initial data for the buffer

■ **ppBuffer:** The created buffer

[Listing 14.8](#) presents the code for creating the vertex buffer for the triangle demo.

## Listing 14.8 Vertex Buffer Creation

[Click here to view code image](#)

---

```
BasicEffectVertex vertices[] =
{
    BasicEffectVertex(XMFLOAT4(-1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Red))),
    BasicEffectVertex(XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Green))),
```

```

BasicEffectVertex(XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Blue)))
};

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) * 
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData,
sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()-
>CreateBuffer(&vertexBufferDesc, &vertexSubResourceData,
&mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}

```

---

The first statements of [Listing 14.8](#) initialize an array of three vertices at positions  $(-1, 0, 0)$ ,  $(0, 1, 0)$ , and  $(1, 0, 0)$  with red, green, and blue vertex colors, respectively. Next a buffer description structure is configured for an immutable vertex buffer, and a `D3D11_SUBRESOURCE_DATA` structure is populated with the array of vertices. Finally, the vertex buffer is created and stored in the `mVertexBuffer` class member.

## Rendering a Triangle

You are now ready to draw the triangle. You render a set of nonindexed primitives (for example, a triangle without an index buffer) in these steps:

1. Set the primitive topology for the input-assembler stage.
2. Bind the input layout to the input-assembler stage.
3. Bind the vertex buffer to the input-assembler stage.
4. Set any shader constants.
5. Apply the effect pass to the device.
6. Execute the nonindexed draw call.

The next sections cover each of these steps.

### Setting the Primitive Topology

The first step for rendering a triangle is to tell the input-assembler stage what topology to use for the incoming vertices. You accomplish this with the

`ID3D11DeviceContext::IASetPrimitiveTopology()` method, which has no return value and accepts a single argument of type `D3D11_PRIMITIVE_TOPOLOGY`. [Table 14.1](#) lists the

most common topologies.

## Enumeration

D3D11_PRIMITIVE_TOPOLOGY_POINTLIST	Interprets the vertex data as a list of points
D3D11_PRIMITIVE_TOPOLOGY_LINELIST	Interprets the vertex data as a list of lines
D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP	Interprets the vertex data as a line strip
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST	Interprets the vertex data as a list of triangles
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP	Interprets the vertex data as a triangle strip

**Table 14.1** Common Primitive Topologies

## Binding the Input Layout

The next step is to bind the input layout to the input-assembler stage with a call to `ID3D11DeviceContext::IASetInputLayout()`. This method has no return value and only a single input parameter: the `ID3D11InputLayout` object you created.

## Binding the Vertex Buffer

Next, you must bind the vertex buffer to the input-assembler stage with a call to `ID3D11DeviceContext::IASetVertexBuffers()`. This method establishes the CPU-to-GPU vertex input stream and has the following prototype:

[Click here to view code image](#)

```
void IASetVertexBuffers(
    UINT StartSlot,
    UINT NumBuffers,
    ID3D11Buffer *const *ppVertexBuffers,
    const UINT *pStrides,
    const UINT *pOffsets);
```

- **StartSlot:** The first input slot to use for the list of vertex buffers. Recall that 16 or 32 input slots are available (depending on the feature level) for binding multiple vertex buffers.
- **NumBuffers:** The number of vertex buffers in `ppVertexBuffers`.
- **ppVertexBuffers:** An array of vertex buffers to bind.
- **pStrides:** An array of stride values, one for each of the vertex buffers. A stride value is the size (in bytes) of the elements in the vertex buffer (for example, the size of the `BasicEffectVertex` structure for the triangle demo).
- **pOffsets:** An array of offsets, one for each of the vertex buffers. An offset is the number of bytes in the vertex buffer to skip before processing vertices. This is useful if a single buffer contains vertices for multiple objects and you intend to draw only a subset of the complete vertex buffer.

The `ID3D11DeviceContext::IASetVertexBuffers()` method might feel a bit

cumbersome at this stage because the triangle demo won't be using multiple vertex buffers or offsets. But this is a one-size-fits-all sort of method that accommodates more advanced usage. [Listing 14.9](#) demonstrates the calls for setting the primitive topology, binding the input layout, and binding the triangle demo's vertex buffer.

## **Listing 14.9** Setting the Primitive Topology and Binding the Input Layout and Vertex Buffer

[Click here to view code image](#)

```
ID3D11DeviceContext* direct3DDeviceContext =  
mGame->Direct3DDeviceContext();  
direct3DDeviceContext->IASetPrimitiveTopology(  
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);  
direct3DDeviceContext->IASetInputLayout(&mInputLayout);  
  
UINT stride = sizeof(BasicEffectVertex);  
UINT offset = 0;  
direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,  
&stride, &offset);
```

## Setting Shader Constants and Applying the Effect Pass

You must update the variables in your shaders (for example, the `WorldViewProjection` matrix) before executing a draw call. The `ID3DX11EffectPass` and `ID3DX11EffectVariable` types from the Effects 11 library provide this functionality. For example, the `ID3DX11EffectVariable::SetRawValue()` method enables you to send arbitrary data to a shader variable. It has the following declaration:

```
HRESULT SetRawValue(  
    void *pData,  
    UINT Offset,  
    UINT ByteCount);
```

- **pData:** The data to set
- **Offset:** An offset (in bytes) from the beginning of the data
- **ByteCount:** The number of bytes to set

You can use this method to set variables of any type. However, the derived effect variable types are a bit more user friendly. For example, the `ID3DX11EffectMatrixVariable` type has a `SetMatrix()` method that accepts just a single input parameter: an array of floating-point values to set. Regardless of which interface you use, all "Set" methods are cached, and their updates are not sent to the GPU until you invoke the `ID3DX11EffectPass::Apply()` method. This is a performance-saving mechanism because you typically set multiple shader variables when drawing an object. By caching these updates, the GPU state can be modified just once per batch instead of once per "Set" call. The `ID3DX11EffectPass::Apply()` method has two parameters, an unsigned integer for flags (which is unused) and the `ID3D11DeviceContext` to apply the pass to. [Listing 14.10](#) demonstrates the calls for updating the `WorldViewProjection` variable and applying the

effect pass.

## Listing 14.10 Setting Shader Constants and Applying the Effect Pass

[Click here to view code image](#)

```
XMMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
XMMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
mWvpVariable->SetMatrix(reinterpret_cast<const float*>(&wvp));

mPass->Apply(0, direct3DDeviceContext);
```

The first two statements in [Listing 14.10](#) load the `mWorldMatrix` class member into a SIMD `XMMATRIX` object and then construct the world view projection matrix. The world matrix must be initialized to the identity matrix or must contain a valid combination of translation, rotation, and scaling transformations.

## Executing the Draw Call

The final step is to execute the nonindexed draw call against the Direct3D device context. This is done through the `ID3D11DeviceContext::Draw()` method, which has the following prototype and parameters:

[Click here to view code image](#)

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

- **VertexCount:** The number of vertices to render.
- **StartVertexLocation:** The index of the first vertex you'd like to render. This supplies an offset into the vertex buffer; it is useful if multiple objects are contained within a shared vertex buffer and you want to render only a subset.

[Listing 14.11](#) demonstrates the draw call for the triangle demo. It specifies three vertices to draw, with no vertex buffer offset.

## Listing 14.11 Executing the Nonindexed Draw Call

[Click here to view code image](#)

```
direct3DDeviceContext->Draw(3, 0);
```

## Putting It All Together

You now have all the pieces for rendering a triangle. [Listing 14.12](#) presents the complete implementation of the `TriangleDemo` class. All the steps in this chapter are split between the `TriangleDemo::Initialize()` and `TriangleDemo::Draw()` methods. Note that the

ID3D11DeviceContext::IASetPrimitiveTopology(), IASetInputLayout(), and IASetVertexBuffers() methods could have been invoked within the TriangleDemo::Initialize() method but are instead invoked (each frame) from the TriangleDemo::Draw() method. This is done to emphasize that components have no knowledge of changes to the Direct3D pipeline state that happen between frames. Another component could have modified any of these settings, which would break the TriangleDemo's rendering if the states weren't reset each frame.

## Listing 14.12 The TriangleDemo.cpp File

[Click here to view code image](#)

```
#include "TriangleDemo.h"
#include "Game.h"
#include "GameException.h"
#include "MatrixHelper.h"
#include "ColorHelper.h"
#include "Camera.h"
#include "Utility.h"
#include "D3DCompiler.h"

namespace Rendering
{
    RTTI_DEFINITIONS(TriangleDemo)

    TriangleDemo::TriangleDemo(Game& game, Camera& camera)
        : DrawableGameComponent(game, camera),
          mEffect(nullptr), mTechnique(nullptr), mPass(nullptr),
          mWvpVariable(nullptr),
          mInputLayout(nullptr),
          mWorldMatrix(MatrixHelper::Identity),
          mVertexBuffer(nullptr)
    {
    }

    TriangleDemo::~TriangleDemo()
    {
        ReleaseObject(mWvpVariable);
        ReleaseObject(mPass);
        ReleaseObject(mTechnique);
        ReleaseObject(mEffect);
        ReleaseObject(mInputLayout);
        ReleaseObject(mVertexBuffer);
    }

    void TriangleDemo::Initialize()
    {
```

```

SetCurrentDirectory(Utility::ExecutableDirectory()).c_str()

// Compile the shader
UINT shaderFlags = 0;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3DCOMPILE_DEBUG;
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

ID3D10Blob* compiledShader = nullptr;
ID3D10Blob* errorMessages = nullptr;
HRESULT hr = D3DCompileFromFile(L"Content\\Effects\\
BasicEffect.fx",nullptr, nullptr, nullptr, "fx_5_0",
shaderFlags, 0,
&compiledShader,
&errorMessages);
if (errorMessages != nullptr)
{
    GameException ex((char*)errorMessages-
>GetBufferPointer(),
hr);
    ReleaseObject(errorMessages);

    throw ex;
}

if (FAILED(hr))
{
    throw GameException("D3DX11CompileFromFile() failed.", hr);
}

// Create an effect object from the compiled shader
hr = D3DX11CreateEffectFromMemory(compiledShader->
GetBufferPointer(),
compiledShader->GetBufferSize(), 0, mGame->Direct3DDevice(),
&mEffect);
if (FAILED(hr))
{
    throw GameException("D3DX11CreateEffectFromMemory() failed.", hr);
}

ReleaseObject(compiledShader);

// Look up the technique, pass, and WVP variable from

```

the  
effect

```
mTechnique = mEffect->GetTechniqueByName ("main11");
if (mTechnique == nullptr)
{
    throw
GameException ("ID3DX11Effect::GetTechniqueByName ()
could not find the specified technique.", hr);
}

mPass = mTechnique->GetPassByName ("p0");
if (mPass == nullptr)
{
    throw
GameException ("ID3DX11EffectTechnique::GetPassBy
Name () could not find the specified pass.", hr);
}

ID3DX11EffectVariable* variable = mEffect-
>GetVariableByName
("WorldViewProjection");
if (variable == nullptr)
{
    throw
GameException ("ID3DX11Effect::GetVariableByName ()
could not find the specified variable.", hr);
}

mWvpVariable = variable->AsMatrix();
if (mWvpVariable->IsValid() == false)
{
    throw GameException ("Invalid effect variable
cast.");
}

// Create the input layout
D3DX11_PASS_DESC passDesc;
mPass->GetDesc (&passDesc);

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};
```

```

    if (FAILED(hr = mGame->Direct3DDevice()->
CreateInputLayout(inputElementDescriptions,
ARRAYSIZE(inputElementDescriptions), passDesc.pIAInputSignature,
passDesc.IAInputSignatureSize, &mInputLayout)))
{
    throw
GameException("ID3D11Device::CreateInputLayout()
failed.", hr);
}

// Create the vertex buffer
BasicEffectVertex vertices[] =
{
    BasicEffectVertex(XMFLOAT4(-1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Red))),
    BasicEffectVertex(XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Green))),
    BasicEffectVertex(XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Blue)))
};

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) *
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof
(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()->CreateBuffer
(&vertexBufferDesc, &vertexSubResourceData, &mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer()
failed.");
}
}

void TriangleDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
}

```

```

    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(BasicEffectVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);

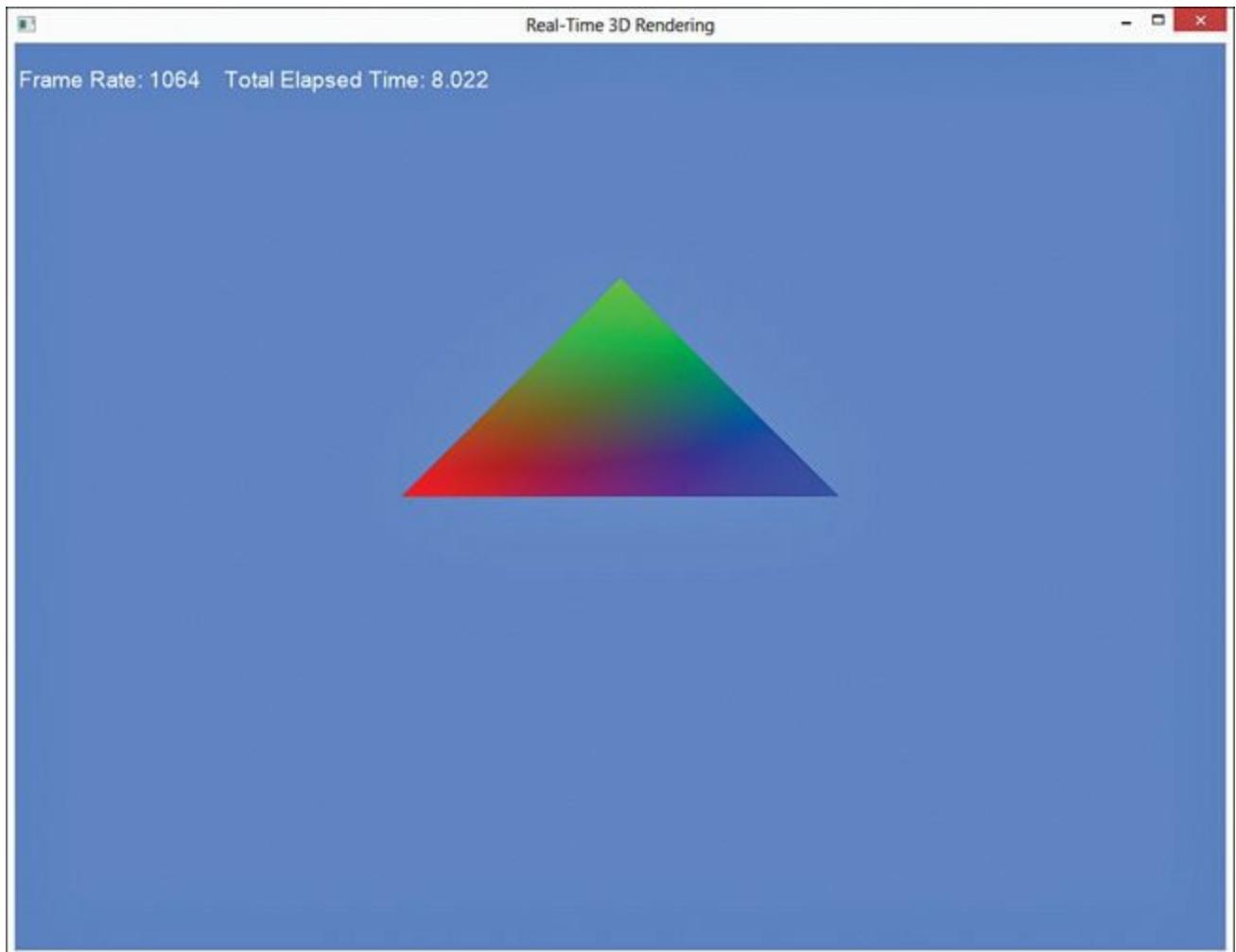
    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mWvpVariable->SetMatrix(reinterpret_cast<const float*>
(&wvp));
}

mPass->Apply(0, direct3DDeviceContext);

direct3DDeviceContext->Draw(3, 0);
}
}

```

[Figure 14.1](#) shows the output of the TriangleDemo component. Note that you can maneuver your camera with the mouse and keyboard and even view the back of the triangle (because the BasicEffect.fx file disabled backface culling).



**Figure 14.1** The output of the triangle demo.

## Spicing Things Up

Let's add some interest to this demo by spinning the triangle around the z-axis. With all the infrastructure you've built, this is incredibly simple to do. First, add a class member called `mAngle` (of type `float`) to your `TriangleDemo` class, and initialize it to zero in the constructor. Then add an override for the `DrawableGameComponent::Update()` method to your class declaration and drop in the method's implementation from [Listing 14.13](#).

### **Listing 14.13** The `TriangleDemo::Update()` Method

[Click here to view code image](#)

```
void TriangleDemo::Update(const GameTime& gameTime)
{
    mAngle += XM_PI *
    static_cast<float>(gameTime.ElapsedGameTime());
    XMStoreFloat4x4(&mWorldMatrix, XMMatrixRotationZ(mAngle));
}
```

This code increments the `mAngle` member each frame (at a rotation rate of 180 degrees per second). This angle is used to construct a rotation matrix around the z-axis, which is stored in the world matrix member. Remember that the world matrix is concatenated with the camera's view and projection matrices and is sent to the GPU in the `TriangleDemo::Draw()` method. That's all there is to it! If you run your application now, you'll see a rotating triangle.

## An Indexed Cube

Let's finish off this chapter by creating another demo, this time for rendering a 3D cube using an index buffer. You can duplicate your `TriangleDemo` to create a `CubeDemo` class with one addition: an `mIndexBuffer` class member of type `ID3D11Buffer` pointer. This will store the array of indices into the vertex buffer.

Recall the discussion of index buffers from [Chapter 1](#), “[Introducing DirectX](#).” Index buffers enable you to eliminate duplicate vertices in the vertex buffer. A cube has six faces, and each cube face can be decomposed into two triangles. Without an index buffer, you'd need 36 vertices to describe the cube (6 faces \* 2 triangles/face \* 3 vertices/triangle). With an index buffer, you need to store only the unique vertices of the cube: eight total. Then you can build an index buffer with 36 indices.

Building an index buffer follows the same steps as for a vertex buffer: You configure a `D3D11_BUFFER_DESC` structure to describe the buffer and then populate a `D3D11_SUBRESOURCE_DATA` structure to fill the buffer with initial data. With these two structures, you invoke the `ID3D11Device::CreateBuffer()` method. One difference is that you specify `D3D_BIND_INDEX_BUFFER` for the `D3D11_BUFFER_DESC.BindFlags` member; another is that the initial data is an array of unsigned integers instead of vertex structures.

[Listing 14.14](#) presents the code for initializing both the vertex and index buffers for the cube demo. This code should reside in your `CubeDemo::Initialize()` method.

## Listing 14.14 Creating the Vertex and Index Buffers for a Cube

[Click here to view code image](#)

```
BasicEffectVertex vertices[] =
{
    BasicEffectVertex(XMFLOAT4(-1.0f, +1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Green))),
    BasicEffectVertex(XMFLOAT4(+1.0f, +1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Yellow))),
    BasicEffectVertex(XMFLOAT4(+1.0f, +1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::White))),
    BasicEffectVertex(XMFLOAT4(-1.0f, +1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>
(&ColorHelper::BlueGreen))),

    BasicEffectVertex(XMFLOAT4(-1.0f, -1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Blue))),
    BasicEffectVertex(XMFLOAT4(+1.0f, -1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Purple))),
    BasicEffectVertex(XMFLOAT4(+1.0f, -1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Red))),
    BasicEffectVertex(XMFLOAT4(-1.0f, -1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Black)))
};

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) * 
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData,
sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()-
>CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, &mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}

UINT indices[] =
{
    0, 1, 2,
```

```

0, 2, 3,
4, 5, 6,
4, 6, 7,
3, 2, 5,
3, 5, 4,
2, 1, 6,
2, 6, 5,
1, 7, 6,
1, 0, 7,
0, 3, 4,
0, 4, 7
};

D3D11_BUFFER_DESC indexBufferDesc;
ZeroMemory(&indexBufferDesc, sizeof(indexBufferDesc));
indexBufferDesc.ByteWidth = sizeof(UINT) * 36;
indexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

D3D11_SUBRESOURCE_DATA indexSubResourceData;
ZeroMemory(&indexSubResourceData, sizeof(indexSubResourceData));
indexSubResourceData.pSysMem = indices;
if (FAILED(mGame->Direct3DDevice()-
>CreateBuffer(&indexBufferDesc,
&indexSubResourceData, &mIndexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}

```

The code in [Listing 14.14](#) creates a vertex buffer from 8 vertices, each with a different color, and an index buffer for the 12 triangles (2 per face) of the cube. Drawing with an index buffer is only slightly different than drawing without one. First, you must bind the index buffer to the input-assembler stage with a call to `ID3D11DeviceContext::IASetIndexBuffer()`. The prototype and parameters are listed here:

[Click here to view code image](#)

```

void IASetIndexBuffer(
    ID3D11Buffer *pIndexBuffer,
    DXGI_FORMAT Format,
    UINT Offset);

```

■ **pIndexBuffer:** The index buffer to bind.

■ **Format:** The format of the data in the index buffer. Only two possible formats exist:

`DXGI_FORMAT_R16_UINT` and `DXGI_FORMAT_R32_UINT`, 16-bit and 32-bit unsigned integers, respectively.

■ **Offset:** The number of bytes to skip before processing indices.

Note that, unlike `ID3D11DeviceContext::IASetVertexBuffers()`, you bind only a single index buffer to the input-assembler stage. Aside from binding the index buffer, the only other difference is in the specific draw call. You render indexed primitives through the `ID3DDeviceContext::DrawIndexed()` method; its prototype and parameters are listed next:

[Click here to view code image](#)

```
void DrawIndexed(
    UINT IndexCount,
    UINT StartIndexLocation,
    INT BaseVertexLocation);
```

■ **IndexCount:** The number of indices to render

■ **StartIndexLocation:** The location of the first index to process

■ **BaseVertexLocation:** A value to add to each index before it is used to look up a vertex from the vertex buffer

The last parameter of the `ID3DDeviceContext::DrawIndexed()` method needs some additional explanation. Overhead is present for switching out vertex and index buffers, so you might consider *sharing* a vertex buffer between objects. In practice, this means concatenating the vertex buffers. But what happens to the associated index buffers? Each index buffer refers to its own *local* vertex buffer. For example, although two objects will likely have different vertices, the first ten vertices in one vertex buffer will have the same indices (0 to 9) as the first ten vertices of another buffer. When you concatenate the index buffers, their indices will be pointing to incorrect vertices without compensating with the `BaseVertexLocation` parameter. If you aren't using shared vertex and index buffers, this value will be 0.

[Listing 14.15](#) presents the `CubeDemo::Draw()` method and demonstrates the `ID3D11DeviceContext::IASetIndexBuffer()` and `ID3D11DeviceContext::DrawIndexed()` calls.

## Listing 14.15 Drawing an Indexed Cube

[Click here to view code image](#)

---

```
void CubeDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);
```

```

    UINT stride = sizeof(BasicEffectVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_FORMAT
R32_UINT, 0);

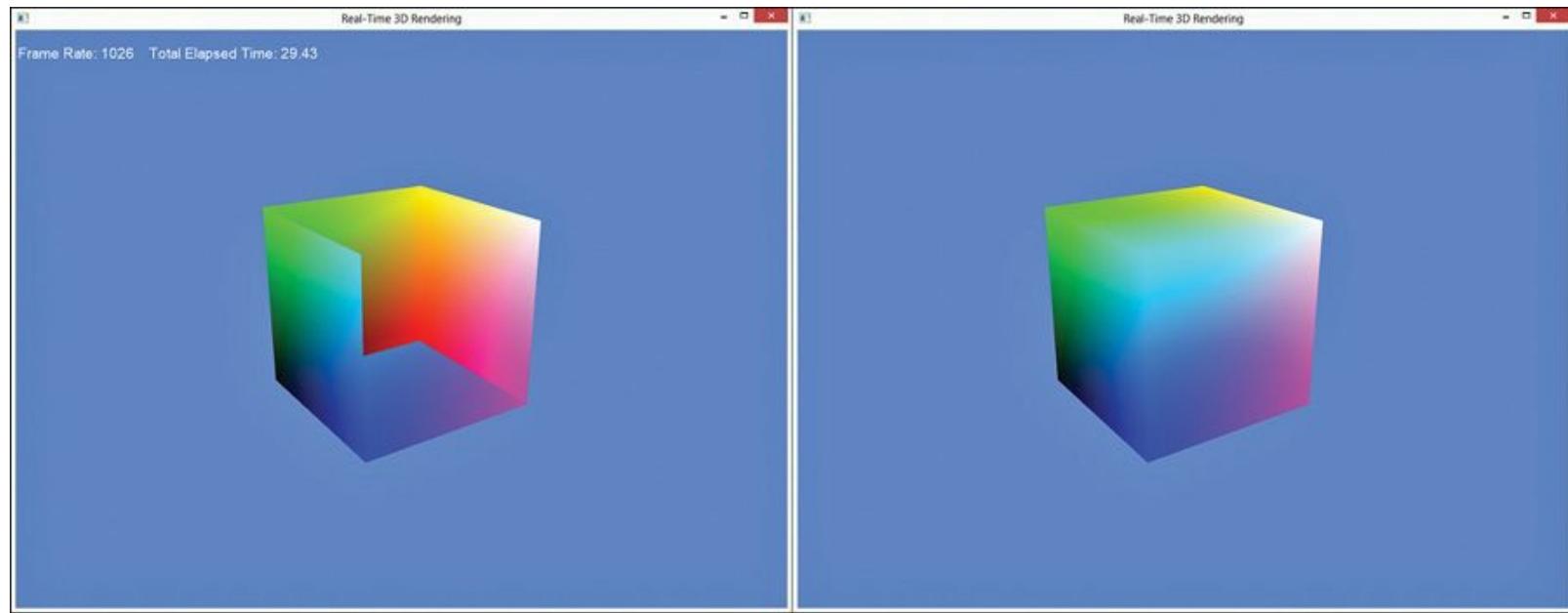
    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mWvpVariable->SetMatrix(reinterpret_cast<const float*>
(&wvp));

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(36, 0, 0);
}

```

To use your CubeDemo, you'll likely want to disable the TriangleDemo (they'll happily coexist, but the triangle will be rendered inside the cube unless you apply a translation transform). Also, if you are using the FpsComponent and you run the application now, you'll see some strange output, like the left side of [Figure 14.2](#).



**Figure 14.2** The cube demo output with the frame rate component (left) and without it (right).

A depth buffer issue occurs that's only now expressed because this is the first time you've rendered triangles that occlude each other. If you remove the frame rate component, your cube will render correctly (the right-side image of [Figure 14.2](#)). The problem stems from the SpriteBatch system that the frame rate component uses; it changes some render states that you need to address.

## A Render State Helper Class

Recall the rasterizer state and blend state objects you worked with when authoring shaders. These have C++ counterparts of type `ID3D11RasterizerState` and `ID3D11BlendState`, and there is also a `ID3D11DepthStencilState` interface. Sometimes (such as in the scenario you just encountered with the cube demo) you'll want to get and set these render states. For example, you could capture these states just before drawing 2D objects with the `SpriteBatch` class and then restore them after the sprite rendering is complete. Following is the list of getter and setter methods for each of the three render state objects:

[Click here to view code image](#)

```
ID3D11DeviceContext::RSGetState()
ID3D11DeviceContext::OMGetBlendState()
ID3D11DeviceContext::OMGetDepthStencilState()

ID3D11DeviceContext::RSSetState()
ID3D11DeviceContext::OMSetBlendState()
ID3D11DeviceContext::OMSetDepthStencilState()
```

These methods all follow the same pattern: You *get* the current state of the Direct3D pipeline into an object with the corresponding interface, and you can *set* such an object back to update the pipeline. If you *set* a state to `NULL`, you update the pipeline to its default state. Instead of listing the prototypes and parameters for each of these methods, [Listings 14.16](#) and [14.17](#) present a `RenderStateHelper` class that provides functionality to save and restore these states individually or as a group. You can use this class to restore the render states after drawing the frame rate component's text to the screen.

### **Listing 14.16** The `RenderStateHelper.h` File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;

    class RenderStateHelper
    {
        public:
            RenderStateHelper(Game& game);
            ~RenderStateHelper();

            static void ResetAll(ID3D11DeviceContext*
deviceContext);

            ID3D11RasterizerState* RasterizerState();
```

```

ID3D11BlendState* BlendState();
ID3D11DepthStencilState* DepthStencilState();

void SaveRasterizerState();
void RestoreRasterizerState() const;

void SaveBlendState();
void RestoreBlendState() const;

void SaveDepthStencilState();
void RestoreDepthStencilState() const;

void SaveAll();
void RestoreAll() const;

private:
    RenderStateHelper(const RenderStateHelper& rhs);
    RenderStateHelper& operator=(const RenderStateHelper&
rhs);

    Game& mGame;

    ID3D11RasterizerState* mRasterizerState;
    ID3D11BlendState* mBlendState;
    FLOAT* mBlendFactor;
    UINT mSampleMask;
    ID3D11DepthStencilState* mDepthStencilState;
    UINT mStencilRef;
};

}

```

---

### Listing 14.17 The RenderStateHelper.cpp File

[Click here to view code image](#)

---

```

#include "RenderStateHelper.h"
#include "Game.h"

namespace Library
{
    RenderStateHelper::RenderStateHelper(Game& game)
        : mGame(game), mRasterizerState(nullptr),
        mBlendState(nullptr),
        mBlendFactor(new FLOAT[4]), mSampleMask(UINT_MAX),
        mDepthStencilState(nullptr), mStencilRef(UINT_MAX)
    {

```

```
}

RenderStateHelper::~RenderStateHelper()
{
    ReleaseObject(mRasterizerState);
    ReleaseObject(mBlendState);
    ReleaseObject(mDepthStencilState);
    DeleteObjects(mBlendFactor);
}

void RenderStateHelper::ResetAll(ID3D11DeviceContext*
deviceContext)
{
    deviceContext->RSSetState(nullptr);
    deviceContext->OMSetBlendState(nullptr, nullptr,
UINT_MAX);
    deviceContext->OMSetDepthStencilState(nullptr,
UINT_MAX);
}

void RenderStateHelper::SaveRasterizerState()
{
    ReleaseObject(mRasterizerState);
    mGame.Direct3DDeviceContext()-
>RSGetState(&mRasterizerState);
}

void RenderStateHelper::RestoreRasterizerState() const
{
    mGame.Direct3DDeviceContext()-
>RSSetState(mRasterizerState);
}

void RenderStateHelper::SaveBlendState()
{
    ReleaseObject(mBlendState);
    mGame.Direct3DDeviceContext()-
>OMGetBlendState(&mBlendState, mBlendFactor, &mSampleMask);
}

void RenderStateHelper::RestoreBlendState() const
{
    mGame.Direct3DDeviceContext()-
>OMSetBlendState(mBlendState, mBlendFactor, mSampleMask);
}

void RenderStateHelper::SaveDepthStencilState()
```

```

    {
        ReleaseObject(mDepthStencilState);
        mGame.Direct3DDeviceContext () ->OMGetDepthStencilState
        (&mDepthStencilState, &mStencilRef);
    }

    void RenderStateHelper::RestoreDepthStencilState() const
    {
        mGame.Direct3DDeviceContext () ->OMSetDepthStencilState
        (mDepthStencilState, mStencilRef);
    }

    void RenderStateHelper::SaveAll()
    {
        SaveRasterizerState();
        SaveBlendState();
        SaveDepthStencilState();
    }

    void RenderStateHelper::RestoreAll() const
    {
        RestoreRasterizerState();
        RestoreBlendState();
        RestoreDepthStencilState();
    }
}

```

---

You can integrate this class directly into the `FpsComponent` class, within the `CubeDemo` class, or within the `RenderingGame` class. If you integrate it within the `RenderingGame` class, you'll want to pull the `FpsComponent` out of the components container and manually draw it with calls such as the following:

[Click here to view code image](#)

```
mRenderStateHelper->SaveAll();
mFpsComponent->Draw(gameTime);
mRenderStateHelper->RestoreAll();
```

Such code guarantees that the render states, changed within the frame rate component's `Draw()` method, don't affect anything else. See the book's companion site for a full code listing of the cube demo.

## Summary

In this chapter, you completed your first full 3D rendering applications! You examined the API calls for compiling and using effects, creating input layouts, and binding vertex and index buffers to the graphics pipeline. You learned about indexed and nonindexed draw calls and how to save and restore render states. In short, you have completed the foundation on which your remaining investigation of Direct3D will stand.

In the next chapter, you begin loading 3D models and develop a more sophisticated material system to accommodate the library of shaders you have developed.

## Exercises

1. Apply transformations to either the triangle or the cube to view them simultaneously (their untransformed vertices place the triangle inside the cube and, thus, out of view of the camera).
2. Attach the keyboard to the cube to translate or rotate it (for example, you can use the arrow keys or the number pad).

# Chapter 15. Models

In this chapter, you learn how to use 3D models in your applications. You integrate a third-party library for importing myriad file formats, and you author a set of classes to represent models at run-time. Along the way, you get some more hands-on experience creating DirectX applications.

## Motivation

Now that you've authored a couple applications, and you've manually specified vertices and indices, you're probably thinking about a better way to do this. Clearly, you cannot hard-code the vertices for every object you'll render. Instead, you'll get your objects from an artist who uses a 3D modeling package such as Autodesk Maya or 3D Studio Max. Your job is to read these assets at runtime and extract the data necessary to render them to the screen. This is a bit easier said than done. The next few sections detail an approach for using 3D models.

## Model File Formats

A lot of file formats for storing 3D models exist, and they can be grossly categorized into three groups: **authoring formats**, **game/graphics engine formats**, and **interchange formats**. An authoring format is specific to a modeling application. For example, Autodesk Maya has two native file formats, Maya ASCII ( .ma) and Maya Binary ( .mb). 3D Studio Max uses .3DS files. Such formats typically contain authoring-related data that's not applicable within the game engine. For example, Autodesk Maya files store a creation history that details the steps used to produce the final model; your rendering engine just needs the final result.

A game/graphics engine file format stores distilled versions of assets that don't contain authoring-related data. Furthermore, these formats are commonly optimized for loading into a particular engine. If a game/graphics engine is intended to be used outside the game studio (for example, for a game that's intended to be modified, or **modded**), the format will be publicly documented or otherwise supported so that user-created assets can be stored correctly. For instance, Blizzard's Starcraft II format MDX3 ( .m3) has been publicly released and has a 3D Studio Max importer/exporter plug-in.

The third category, interchange formats, contains a set of file formats that are independent of an authoring tool or game/graphics engine. These formats are intended to allow the exchange of assets between software applications—sort of a one-format-to-rule-them-all idea. Perhaps the chief interchange format is COLLADA (Collaborative Design Activity), which a variety of authoring tools and game engines support.

Regardless of category, a file can be stored as text (for example the XML-based COLLADA format) or binary. Text formats require more disk space to store the same asset, but any text editor can read and edit them. However, just because a file format can be visualized as strings doesn't mean that it's really human readable. You can find some hairy text formats out there that you'll corrupt just as easily as a binary file if you're not careful. It's usually a good idea to let the authoring tool save the file and to leave hand-editing as a last resort.

Something else to consider is that not all formats are capable of storing the data your application might require. For example, the Wavefront OBJ ( .obj) format is a simple and widely supported file format, but it does not store animation data.

In practice, you want to settle on a specific authoring format. Such standardization makes debugging

your assets simpler. (Yes, you will find problems in your assets than can cause a runtime crash or otherwise produce incorrect rendering.) If you are unable to standardize, you'll have to wrangle multiple formats into a consistent representation that your rendering engine can access.

## The Content Pipeline

Asset wrangling is the job of the content pipeline. In the graphics pipeline, primitives are fed in and pixels come out. For the content pipeline, an artist provides assets in one format, and game-ready assets are output. For example, you might convert an `.obj` model file into a format your game engine can use. But unlike the Direct3D graphics pipeline, no preconstructed content pipeline can help manipulate your assets. You have to build any such system or adopt a third-party utility.

Typically, assets are processed through the content pipeline at build time and saved into a game engine format that you've created (or adopted). At runtime, you load these assets and use their data. You could perform any content manipulation at runtime, but you would incur the corresponding overhead. It's not uncommon for a game engine to support both build-time and runtime asset conditioning, with runtime use removed from a release version of the game. In this chapter, you convert your assets at runtime, but the system you develop can be easily applied to a build-time content pipeline.

A content pipeline has many aspects, from the workflow for converting assets (which might include a GUI or a revision-control system) to the myriad types of content needed for your application (such as 3D models, textures, or audio files). But the underpinning of a content pipeline is serialization and deserialization—writing and reading files. At its core, the content pipeline revolves around parsing one or more input formats to produce a uniform output that the game engine can consume. You might be able to standardize on a single authoring format, but this book makes no such assumption and opts to support as many input formats as possible. However, writing parsers for many file formats is outside the scope of this text. Instead, we adopt an open source package called the Open Asset Import Library, which is available at <http://assimp.sourceforge.net>.

## The Open Asset Import Library

The Open Asset Import Library is a C++ API that imports a large variety of 3D model formats and represents them in a consistent fashion. [Table 15.1](#) lists just a few of the supported formats. Visit the Open Asset Import Library website for a complete list.

Format	File Extension
Collada	.dae
3D Studio Max	.3ds
Wavefront OBJ	.obj
Blender	.blend
Stanford Polygon Library	.ply
DirectX	.x
Starcraft II MDX3	.m3
Quake III	.md3

**Table 15.1** Common 3D Model Formats the Open Asset Import Library Supports

To use the library, first extract the package to the *external* directory and add *Include Directory* and *Library Directory* search paths to your Library project. Then add a reference to the `assimp.lib` file in your Game or Library projects (or to an entirely separate project to create a build-time content pipeline). The Open Asset Import Library is distributed as a dynamic link library (`.dll`), and `assimp.lib` is an import library. The DLL contains the actual implementation of the API. Therefore, you must copy the DLL to your output directory (where your executable resides) before launching your application. A post-build step can handle this nicely. If you have trouble with these steps, you can find complete projects on the book’s companion website.

Although you use the Open Asset Import Library to read various model formats, you’ll create your own classes to represent 3D models. In this way, you remove any explicit runtime dependency on this library.

## What’s in a Model?

So what exactly does a 3D model contain? This question has no single answer, but models are commonly described as a set of **meshes**. A mesh is a group of vertices, edges, and faces that describe the shape of a 3D object. By describing a model as a set of meshes, you allow for multiple “subobjects” to be treated both separately and as a whole. For example, a car might be described by separate meshes for the chassis, two doors, and four wheels. The doors could be transformed independently from the wheels and the chassis, but all the objects might have a shared position in world space. Furthermore, you could apply a different shader to each mesh. For instance, the chassis and doors might use an environment mapping effect, whereas the wheels use a diffuse shader.

A model can also contain inputs to a shader, or what could be called **model materials**. If a material represents an instance of an effect with values for shader variables, a model material is where (at least some of) the values of those variables originate. For example, a model material would refer to the name of the shader to use for rendering and a list of name/value pairs matching initial values for shader constants. A model material could also include a list of texture references (a texture reference is a filename that refers to a texture) or embedded textures (textures whose actual data is stored inside the model file). However, embedded textures are less common than texture references because

duplicate embedded textures waste memory.

Using this definition of a model, [Listing 15.1](#) presents the declaration of the Model class.

## Listing 15.1 The Model.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;
    class Mesh;
    class ModelMaterial;

    class Model
    {
        public:
            Model(Game& game, const std::string& filename, bool
flipUVs =
false);
            ~Model();

            Game& GetGame();
            bool HasMeshes() const;
            bool HasMaterials() const;

            const std::vector<Mesh*>& Meshes() const;
            const std::vector<ModelMaterial*>& Materials() const;

        private:
            Model(const Model& rhs);
            Model& operator=(const Model& rhs);

            Game& mGame;
            std::vector<Mesh*> mMeshes;
            std::vector<ModelMaterial*> mMaterials;
    };
}
```

---

As you can see, the Model class contains collections for Mesh and ModelMaterial objects and carries around a Game class reference. Note that model materials are contained within the model, but they describe the properties of a mesh. Multiple meshes can thus refer to the same model material.

The Model constructor accepts a filename to read and a Boolean to flip a model's vertical texture

coordinates (for loading OpenGL-style models). Note that this declaration has no dependency on the Open Asset Import Library. The `Model` class implementation hides the specifics of asset importing from anything consuming the `Model` interface. We defer a full discussion of the `Model` class Implementation until after we present the associated `Mesh` and `ModelMaterial` classes.

## Meshes

A mesh can be minimally described by a set of vertices, but it can also include indices, normals, tangents, binormals, texture coordinates, and vertex colors. A mesh can also reference a model material. [Listing 15.2](#) presents the declaration of the `Mesh` class.

### **Listing 15.2** The `Mesh.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

struct aiMesh;

namespace Library
{
    class Material;
    class ModelMaterial;

    class Mesh
    {
        friend class Model;

    public:
        Mesh(Model& model, ModelMaterial* material);
        ~Mesh();

        Model& GetModel();
        ModelMaterial* GetMaterial();
        const std::string& Name() const;

        const std::vector<XMFLOAT3>& Vertices() const;
        const std::vector<XMFLOAT3>& Normals() const;
        const std::vector<XMFLOAT3>& Tangents() const;
        const std::vector<XMFLOAT3>& BiNormals() const;
        const std::vector<std::vector<XMFLOAT3>*>&
        TextureCoordinates()
        const;
        const std::vector<std::vector<XMFLOAT4>*>&
        VertexColors()
```

```

const;
    UINT FaceCount() const;
    const std::vector<UINT>& Indices() const;
    void CreateIndexBuffer(ID3D11Buffer** indexBuffer);

private:
    Mesh(Model& model, aiMesh& mesh);
    Mesh(const Mesh& rhs);
    Mesh& operator=(const Mesh& rhs);

    Model& mModel;
    ModelMaterial* mMaterial;
    std::string mName;
    std::vector<XMFLOAT3> mVertices;
    std::vector<XMFLOAT3> mNormals;
    std::vector<XMFLOAT3> mTangents;
    std::vector<XMFLOAT3> mBiNormals;
    std::vector<std::vector<XMFLOAT3>*> mTextureCoordinates;
    std::vector<std::vector<XMFLOAT4>*> mVertexColors;
    UINT mFaceCount;
    std::vector<UINT> mIndices;
};

}

```

Note the forward declaration of the `aiMesh` structure and its use within a private constructor of the `Mesh` class. This is the mesh representation from the Open Asset Import Library. It's included here because the implementation of the `Model` class uses the Open Asset Import Library to convert assets at runtime. Making it private within the `Mesh` class keeps the interface free of this dependency and implies that you could excise the Open Asset Import Library without affecting the rest of your application. You would do exactly that if you made a build-time content pipeline.

Also note the `Mesh::CreateIndexBuffer()` method. If a mesh contains indices, it has all the information required to populate an index buffer. This isn't the case for creating vertex buffers because the definition of a vertex buffer depends on the shader used to render the mesh.

## Model Materials

The final type in the 3D model system is the `ModelMaterial` class. As with models and meshes, you can use a variety of representations to describe a model's shader associations and inputs. The `ModelMaterial` class stores only a name and a list of textures. The name should match the effect to use when rendering a mesh. For example, the model material's name would be `BasicEffect` to use the `BasicEffect.fx` file. The texture list stores filenames, and each element in the collection is associated with the texture's intended usage. For example, a texture could be identified as the diffuse color texture or a specular map. The `TextureType` enumeration defines the supported texture types.

[Listing 15.3](#) presents the declarations of the `ModelMaterial` class and the `TextureType`

enumeration.

### Listing 15.3 The ModelMaterial.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

struct aiMaterial;

namespace Library
{
    enum TextureType
    {
        TextureTypeDiffuse = 0,
        TextureTypeSpecularMap,
        TextureTypeAmbient,
        TextureTypeEmissive,
        TextureTypeHeightmap,
        TextureTypeNormalMap,
        TextureTypeSpecularPowerMap,
        TextureTypeDisplacementMap,
        TextureTypeLightMap,
        TextureTypeEnd
    };

    class ModelMaterial
    {
        friend class Model;

    public:
        ModelMaterial(Model& model);
        ~ModelMaterial();

        Model& GetModel();
        const std::string& Name() const;
        const std::map<TextureType, std::vector<std::wstring>>
Textures() const;

    private:
        static void InitializeTextureTypeMappings();
        static std::map<TextureType, UINT> sTextureTypeMappings;

        ModelMaterial(Model& model, aiMaterial* material);
        ModelMaterial(const ModelMaterial& rhs);
```

```

    ModelMaterial& operator=(const ModelMaterial& rhs);

    Model& mModel;
    std::string mName;
    std::map<TextureType, std::vector<std::wstring>*>
mTextures;
};

}

```

---

## Asset Loading

With the declarations of the `Model`, `Mesh`, and `ModelMaterial` classes in hand, we can now discuss loading assets into these types. This is where the Open Asset Import Library comes in.

## Loading Models

Just as the `Model` class is the root type of your custom model system, the `aiScene` type is the root structure for data imported with the Open Asset Import Library. An `aiScene` object is populated with a call to `Importer::ReadFile()` and exposes members for a model's meshes and materials. All asset loading originates in the `Model` constructor, which loads meshes and model materials by instantiating their respective classes. [Listing 15.4](#) presents the implementation of the `Model` class.

### [Listing 15.4](#) The `Model.cpp` File

[Click here to view code image](#)

---

```

#include "Model.h"
#include "Game.h"
#include "GameException.h"
#include "Mesh.h"
#include "ModelMaterial.h"
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

namespace Library
{
    Model::Model(Game& game, const std::string& filename, bool
flipUVs)
        : mGame(game), mMeshes(), mMaterials()
    {
        Assimp::Importer importer;

        UINT flags = aiProcess_Triangulate | aiProcess_
JoinIdenticalVertices | aiProcess_SortByPType | aiProcess_FlipWindingOrder;
        if (flipUVs)

```

```
    {
        flags |= aiProcess_FlipUVs;
    }

    const aiScene* scene = importer.ReadFile(filename,
flags);
    if (scene == nullptr)
    {
        throw GameException(importer.GetErrorString());
    }

    if (scene->HasMaterials())
    {
        for (UINT i = 0; i < scene->mNumMaterials; i++)
        {
            mMaterials.push_back(new ModelMaterial(*this,
scene->mMaterials[i]));
        }
    }

    if (scene->HasMeshes())
    {
        for (UINT i = 0; i < scene->mNumMeshes; i++)
        {
            Mesh* mesh = new Mesh(*this, *(scene-
>mMeshes[i]));
            mMeshes.push_back(mesh);
        }
    }
}

Model::~Model()
{
    for (Mesh* mesh : mMeshes)
    {
        delete mesh;
    }

    for (ModelMaterial* material : mMaterials)
    {
        delete material;
    }
}

Game& Model::GetGame()
{
    return mGame;
```

```
}

bool Model::HasMeshes() const
{
    return (mMeshes.size() > 0);
}

bool Model::HasMaterials() const
{
    return (mMaterials.size() > 0);
}

const std::vector<Mesh*>& Model::Meshes() const
{
    return mMeshes;
}

const std::vector<ModelMaterial*>& Model::Materials() const
{
    return mMaterials;
}
```

---

If you were authoring a build-time content pipeline, the work performed in the `Model` constructor would be saved to an intermediate file format. Then you would add a second constructor/method that deserialized these intermediary files and could (optionally) remove the Open Asset Import Library functionality from your runtime Library project. Furthermore, any flipping of the vertical texture coordinates would be performed in a build-time content pipeline and would be removed from runtime model loading.

## Loading Meshes

Loading a mesh entails reading the actual vertices, indices, normals, and so on, so it involves more code than for the `Model` class. Still, the patterns are the same. The Open Asset Import Library `aiMesh` structure exposes the necessary data, and you simply transfer it to your custom `Mesh` class. [Listing 15.5](#) presents an abbreviated implementation of the `Mesh` class. The omitted methods are single-line accessors to expose the private data members. You can find a full implementation on the book's companion website.

### Listing 15.5 The `Mesh.cpp` File (Abbreviated)

[Click here to view code image](#)

---

```
#include "Mesh.h"
#include "Model.h"
#include "Game.h"
#include "GameException.h"
```

```
#include <assimp/scene.h>

namespace Library
{
    Mesh::Mesh(Model& model, aiMesh& mesh)
        : mModel(model), mMaterial(nullptr),
mName(mesh.mName.C_Str()),
        mVertices(), mNormals(), mTangents(), mBiNormals(),
        mTextureCoordinates(), mVertexColors(), mFaceCount(0),
mIndices()
    {
        mMaterial = mModel.Materials().at(mesh.mMaterialIndex);

        // Vertices
        mVertices.reserve(mesh.mNumVertices);
        for (UINT i = 0; i < mesh.mNumVertices; i++)
        {
            mVertices.push_back(XMFLOAT3(reinterpret_cast<const
float*>(&mesh.mVertices[i])));
        }

        // Normals
        if (mesh.HasNormals())
        {
            mNormals.reserve(mesh.mNumVertices);
            for (UINT i = 0; i < mesh.mNumVertices; i++)
            {
                mNormals.push_back(XMFLOAT3(reinterpret_cast<cons
float*>(&mesh.mNormals[i])));
            }
        }

        // Tangents and Binormals
        if (mesh.HasTangentsAndBitangents())
        {
            mTangents.reserve(mesh.mNumVertices);
            mBiNormals.reserve(mesh.mNumVertices);
            for (UINT i = 0; i < mesh.mNumVertices; i++)
            {
                mTangents.push_back(XMFLOAT3(reinterpret_cast<con
float*>(&mesh.mTangents[i])));
                mBiNormals.push_back(XMFLOAT3(reinterpret_cast<cc
float*>(&mesh.mBitangents[i])));
            }
        }

        // Texture Coordinates
    }
}
```

```

    UINT uvChannelCount = mesh.GetNumUVChannels();
    for (UINT i = 0; i < uvChannelCount; i++)
    {
        std::vector<XMFLOAT3>* textureCoordinates = new
std::vector<XMFLOAT3>();
        textureCoordinates->reserve(mesh.mNumVertices);
        mTextureCoordinates.push_back(textureCoordinates);

        aiVector3D* aiTextureCoordinates =
mesh.mTextureCoords[i];
        for (UINT j = 0; j < mesh.mNumVertices; j++)
        {
            textureCoordinates-
>push_back(XMFLOAT3(reinterpret_
cast<const float*>(&aiTextureCoordinates[j])));
        }
    }

    // Vertex Colors
    UINT colorChannelCount = mesh.GetNumColorChannels();
    for (UINT i = 0; i < colorChannelCount; i++)
    {
        std::vector<XMFLOAT4>* vertexColors = new
std::vector<XMFLOAT4>();
        vertexColors->reserve(mesh.mNumVertices);
        mVertexColors.push_back(vertexColors);

        aiColor4D* aiVertexColors = mesh.mColors[i];
        for (UINT j = 0; j < mesh.mNumVertices; j++)
        {
            vertexColors-
>push_back(XMFLOAT4(reinterpret_cast
<const float*>(&aiVertexColors[j])));
        }
    }

    // Faces (note: could pre-reserve if we limit primitive
types)
    if (mesh.HasFaces())
    {
        mFaceCount = mesh.mNumFaces;
        for (UINT i = 0; i < mFaceCount; i++)
        {
            aiFace* face = &mesh.mFaces[i];

            for (UINT j = 0; j < face->mNumIndices; j++)
            {

```

```

        mIndices.push_back(face->mIndices[j]);
    }
}
}

Mesh::~Mesh()
{
    for (std::vector<XMFLOAT3>* textureCoordinates :
mTextureCoordinates)
    {
        delete textureCoordinates;
    }

    for (std::vector<XMFLOAT4>* vertexColors :
mVertexColors)
    {
        delete vertexColors;
    }
}

void Mesh::CreateIndexBuffer(ID3D11Buffer** indexBuffer)
{
    assert(indexBuffer != nullptr);

    D3D11_BUFFER_DESC indexBufferDesc;
    ZeroMemory(&indexBufferDesc, sizeof(indexBufferDesc));
    indexBufferDesc.ByteWidth = sizeof(UINT) *
mIndices.size();
    indexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
    indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

    D3D11_SUBRESOURCE_DATA indexSubResourceData;
    ZeroMemory(&indexSubResourceData, sizeof
(indexSubResourceData));
    indexSubResourceData.pSysMem = &mIndices[0];
    if (FAILED(mModel.GetGame().Direct3DDevice()->
CreateBuffer(&indexBufferDesc, &indexSubResourceData,
indexBuffer)))
    {
        throw GameException("ID3D11Device::CreateBuffer() failed.");
    }
}

```

The specific syntax of the Mesh constructor might require close inspection to fully understand, but the

broad strokes are pretty simple. You just iterate through the various `aiMesh` structure members and copy their data to `Mesh` class members. Visit the Open Asset Import Library website for more details on the `aiMesh` type.

Note that the `Model::mTextureCoordinates` and `Model::mVertexColors` class members are vectors of vectors. This means that a mesh can have multiple sets of coordinates and colors, which would apply to separate textures or rendering techniques. Also note the `Model::mMaterial` class member. This is a pointer back to one of the `ModelMaterial` objects stored in the `Model` class.

Finally, take a look at the `Mesh::CreateIndexBuffer()` method. On examination, this code should look familiar: This is the same code you wrote to create an index buffer for the cube demo (in the last chapter). It's merely been added to the `Mesh` class for reusability.

## Loading Model Materials

[Listing 15.6](#) presents the implementation for the `ModelMaterial` class. This is the last piece of the model system before you can exercise your code with a demo. The `ModelMaterial` constructor pulls data from the `aiMaterial` structure of the Open Asset Import Library. The only odd bit of implementation here is the mapping between the Open Asset Import Library texture types and your custom `TextureType` enumeration. Again, this is done to hide the Open Asset Import Library from the rest of your application. The static mapping is initialized within the `ModelMaterial::InitializeTextureTypeMappings()` method.

### **Listing 15.6** The `ModelMaterial.cpp` File

[Click here to view code image](#)

```
#include "ModelMaterial.h"
#include "GameException.h"
#include "Utility.h"
#include <assimp/scene.h>

namespace Library
{
    std::map<TextureType, UINT>
    ModelMaterial::sTextureTypeMappings;

    ModelMaterial::ModelMaterial(Model& model)
        : mModel(model), mTextures()
    {
        InitializeTextureTypeMappings();
    }

    ModelMaterial::ModelMaterial(Model& model, aiMaterial* material)
        : mModel(model), mTextures()
    {
        InitializeTextureTypeMappings();
    }
}
```

```

aiString name;
material->Get(AI_MATKEY_NAME, name);
mName = name.C_Str();

    for (TextureType textureType = (TextureType)0;
textureType < TextureTypeEnd; textureType = (TextureType)
(textureType + 1))
    {
        aiTextureType mappedTextureType =
(aiTextureType)sTextureTypeMappings[textureType];

        UINT textureCount = material->GetTextureCount
(mappedTextureType);
        if (textureCount > 0)
        {
            std::vector<std::wstring>* textures = new
std::vector
<std::wstring>();
            mTextures.insert(std::pair<TextureType,
std::vector<std
::wstring>*>(textureType, textures));

            textures->reserve(textureCount);
            for (UINT textureIndex = 0; textureIndex <
textureCount; textureIndex++)
            {
                aiString path;
                if (material->GetTexture(mappedTextureType,
textureIndex, &path) == AI_SUCCESS)
                {
                    std::wstring wPath;
                    Utility::ToWideString(path.C_Str(), wPath);

                    textures->push_back(wPath);
                }
            }
        }
    }

ModelMaterial::~ModelMaterial()
{
    for (std::pair<TextureType, std::vector<std::wstring>*>
textures : mTextures)
    {

```

```

        DeleteObject(textures.second);
    }

}

Model& ModelMaterial::GetModel()
{
    return mModel;
}

const std::string& ModelMaterial::Name() const
{
    return mName;
}

const std::map<TextureType, std::vector<std::wstring>*>
ModelMaterial::Textures() const
{
    return mTextures;
}

void ModelMaterial::InitializeTextureTypeMappings()
{
    if (sTextureTypeMappings.size() != TextureTypeEnd)
    {
        sTextureTypeMappings[TextureTypeDiffuse] =
aiTextureType_DIFFUSE;
        sTextureTypeMappings[TextureTypeSpecularMap] =
aiTextureType_SPECULAR;
        sTextureTypeMappings[TextureTypeAmbient] =
aiTextureType_AMBIENT;
        sTextureTypeMappings[TextureTypeHeightmap] =
aiTextureType_HEIGHT;
        sTextureTypeMappings[TextureTypeNormalMap] =
aiTextureType_NORMALS;
        sTextureTypeMappings[TextureTypeSpecularPowerMap] =
aiTextureType_SHININESS;
        sTextureTypeMappings[TextureTypeDisplacementMap] =
aiTextureType_DISPLACEMENT;
        sTextureTypeMappings[TextureTypeLightMap] =
aiTextureType_LIGHTMAP;
    }
}

```

Some functionality is missing from this implementation of a model material—specifically, a set of name/value pairs matching shader variables. We have intentionally left this out at this stage because we haven't introduced a full-fledged material system yet. You'll develop a material system in the next

chapter.

## A Model Rendering Demo

It's time to put the model system to work. For this demo, you'll render a sphere model that contains just a single mesh. The sphere model (`Sphere.obj`) is stored in Wavefront OBJ format, and you can find it on the companion website. You'll use the `BasicEffect.fx` file for rendering (which doesn't use a texture; just interpolated vertex colors).

Start by adding a `ModelDemo` class whose declaration is identical to the `CubeDemo` type of the last chapter. Add two new class members, an unsigned integer for storing the number of indices in the model (`ModelDemo::mIndexCount`) and a `CreateVertexBuffer()` method with the following prototype:

[Click here to view code image](#)

```
void CreateVertexBuffer(ID3D11Device* device, const Mesh& mesh,  
ID3D11Buffer** vertexBuffer) const;
```

The chief differences between the demos is how the vertex and index buffers are initialized. This happens within the `ModelDemo::Initialize()` method. In the cube demo, you manually constructed the vertex and index buffers. In the model demo, you load the `Sphere.obj` file and build the buffers from the single mesh within the model. [Listing 15.7](#) presents an abbreviated `ModelDemo::Initialize()` method and the newly added `ModelDemo::CreateVertexBuffer()` method. The code omitted from the `ModelDemo::Initialize()` method comes directly from the cube demo—specifically, the code for compiling a shader; creating an effect object; looking up the technique, pass, and `WorldViewProjection` variable; and creating the input layout.

### Listing 15.7 Loading a Model

[Click here to view code image](#)

---

```
void ModelDemo::Initialize()  
{  
    // ... Previously presented code omitted for brevity ...  
  
    // Load the model  
    std::unique_ptr<Model> model(new Model(*mGame,  
"Content\\Models\\Sphere.obj", true));  
  
    // Create the vertex and index buffers  
    Mesh* mesh = model->Meshes().at(0);  
    CreateVertexBuffer(mGame->Direct3DDevice(), *mesh,  
&mVertexBuffer);  
    mesh->CreateIndexBuffer(&mIndexBuffer);  
    mIndexCount = mesh->Indices().size();  
}
```

```

void ModelDemo::CreateVertexBuffer(ID3D11Device* device, const
Mesh&
mesh, ID3D11Buffer** vertexBuffer) const
{
    const std::vector<XMFLOAT3>& sourceVertices =
mesh.Vertices();

    std::vector<BasicEffectVertex> vertices;
    vertices.reserve(sourceVertices.size());
    if (mesh.VertexColors().size() > 0)
    {
        std::vector<XMFLOAT4>* vertexColors =
mesh.VertexColors().
at(0);
        assert(vertexColors->size() == sourceVertices.size());

        for (UINT i = 0; i < sourceVertices.size(); i++)
        {
            XMFLOAT3 position = sourceVertices.at(i);
            XMFLOAT4 color = vertexColors->at(i);
            vertices.push_back(BasicEffectVertex(XMFLOAT4(position.x,
position.y, position.z, 1.0f), color));
        }
    }
    else
    {
        for (UINT i = 0; i < sourceVertices.size(); i++)
        {
            XMFLOAT3 position = sourceVertices.at(i);
            XMFLOAT4 color = ColorHelper::RandomColor();
            vertices.push_back(BasicEffectVertex(XMFLOAT4(position.x,
position.y, position.z, 1.0f), color));
        }
    }
}

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) *
vertices.
size();
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData,
sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = &vertices[0];

```

```

if (FAILED(device->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, vertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer()
failed.");
}

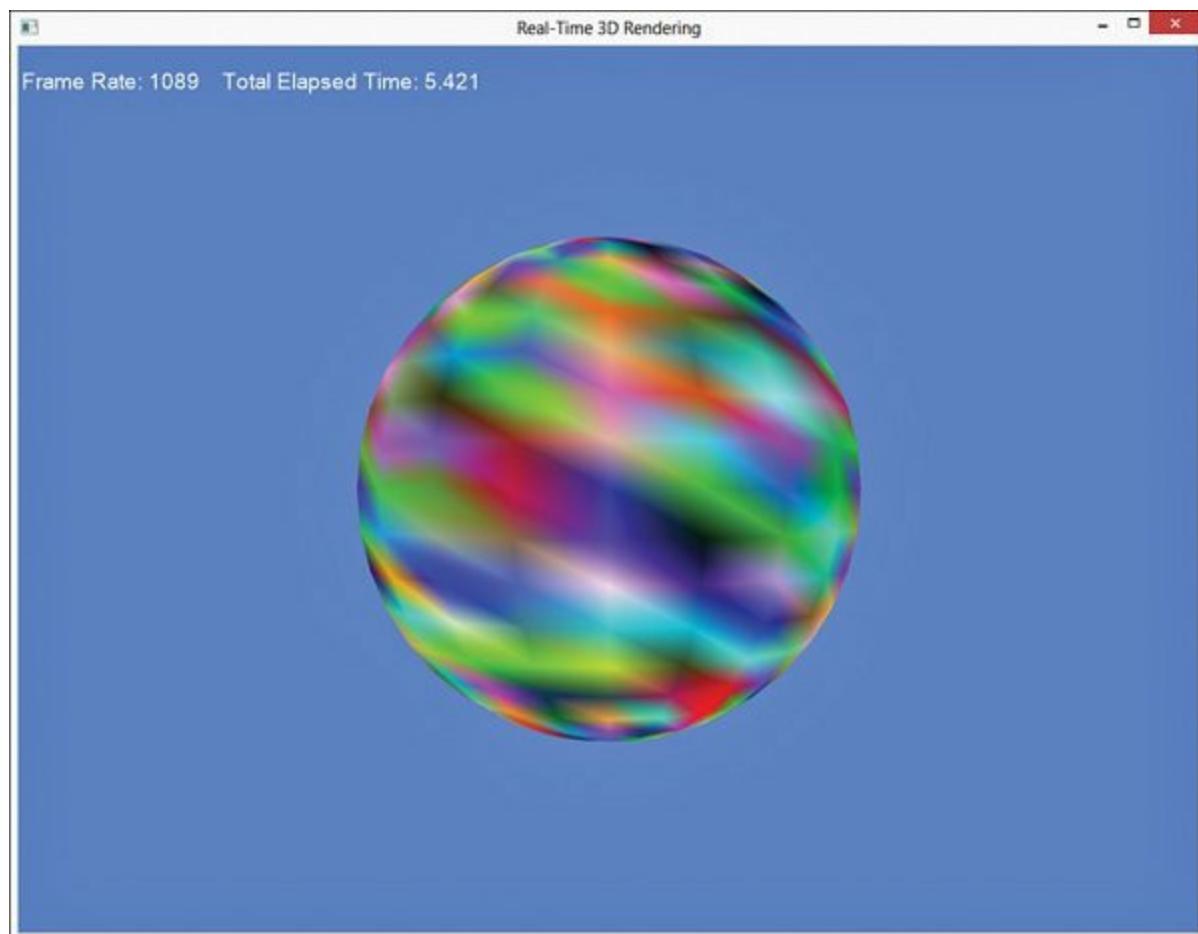
```

---

The ModelDemo::Initialize() method loads the Sphere.obj file by instantiating a Model object. If loading succeeds, there is exactly one mesh for this particular object, which is retrieved through the Model::Meshes() accessor. This mesh is used to initialize the vertex and index buffers.

You've already examined the Mesh::CreateIndexBuffer() method, but new in [Listing 15.7](#) is the ModelDemo::CreateVertexBuffer() method. The last few lines of this method build the vertex buffer and are the same as for the manually constructed vertices of the cube demo. But in this method, you build your BasicEffectVertex array dynamically from the data stored in the mesh. Note the test for vertex colors: If the mesh contains vertex colors, they are used. Otherwise, a random color is assigned to each vertex (assign a solid color, if you prefer).

The ModelDemo::Draw() method is identical to the cube demo, except that you specify mIndexCount as the first argument to the ID3D11DeviceContext::Draw() method. Integrate this demo into your RenderingGame class, and you'll see output similar to [Figure 15.1](#).



**Figure 15.1** Output of the model demo.

## Texture Mapping

Let's extend the work in the model demo to apply a texture. Start by creating a new class called `TexturedModelDemo` that begins as a copy of the `ModelDemo` class. Next, add the following class members:

[Click here to view code image](#)

```
ID3D11ShaderResourceView* mTextureShaderResourceView;  
ID3DX11EffectShaderResourceVariable* mColorTextureVariable;
```

The `ID3D11ShaderResourceView` interface specifies a resource (for example, a texture) that a shader can use. The `ID3DX11EffectShaderResourceVariable` interface is the Effect 11 type that represents the shader variable that will accept a shader resource (for example, the HLSL `Texture2D` type).

For this demo, you'll use the `TextureMapping.fx` effect that you authored back in [Chapter 5, “Texture Mapping.”](#) To refresh your memory, this effect has a `WorldViewProjection` matrix and a `ColorTexture` as shader constants, and the vertex shader accepts the object position and texture coordinates for each vertex. You also want to replace the `main10` technique to `main11` and use the `technique11` keyword.

Because this effect has a different input signature, you need a different vertex structure. Instead of the `BasicEffectVertex` type, declare the `TextureMappingVertex` structure as follows:

[Click here to view code image](#)

```
typedef struct _TextureMappingVertex  
{  
    XMFBYTE4 Position;  
    XMFBYTE2 TextureCoordinates;  
  
    _TextureMappingVertex() { }  
  
    _TextureMappingVertex(XMFBYTE4 position, XMFBYTE2  
    textureCoordinates)  
        : Position(position),  
        TextureCoordinates(textureCoordinates) {  
    }  
} TextureMappingVertex;
```

The remaining changes lie within the `TexturedModelDemo` class implementation. [Listing 15.8](#) presents the `TexturedModelDemo::Initialize()`, `TexturedModelDemo::Draw()`, and `TexturedModelDemo::CreateVertexBuffer()` methods. Visit the companion website for a complete class implementation.

## **Listing 15.8** The `TexturedModelDemo` Class Implementation (Abbreviated)

[Click here to view code image](#)

---

```
void TexturedModelDemo::Initialize()  
{  
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());
```

```

// Compile the shader
UINT shaderFlags = 0;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3DCOMPILE_DEBUG;
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

ID3D10Blob* compiledShader = nullptr;
ID3D10Blob* errorMessages = nullptr;
HRESULT hr =
D3DCompileFromFile(L"Content\\Effects\\TextureMapping.
fx", nullptr, nullptr, nullptr, "fx_5_0", shaderFlags, 0,
&compiledShader, &errorMessages);
if (errorMessages != nullptr)
{
    GameException ex((char*)errorMessages-
>GetBufferPointer(), hr);
    ReleaseObject(errorMessages);

    throw ex;
}

if (FAILED(hr))
{
    throw GameException("D3DX11CompileFromFile() failed.",
hr);
}

// Create an effect object from the compiled shader
hr = D3DX11CreateEffectFromMemory(compiledShader->
GetBufferPointer(), compiledShader->GetBufferSize(), 0, mGame->
Direct3DDevice(), &mEffect);
if (FAILED(hr))
{
    throw GameException("D3DX11CreateEffectFromMemory()
failed.", hr);
}

ReleaseObject(compiledShader);

// Look up the technique, pass, and WVP variable from the
effect
mTechnique = mEffect->GetTechniqueByName("main11");
if (mTechnique == nullptr)
{

```

```
        throw GameException("ID3DX11Effect::GetTechniqueByName() could not find the specified technique.", hr);
    }

mPass = mTechnique->GetPassByName("p0");
if (mPass == nullptr)
{
    throw GameException("ID3DX11EffectTechnique::GetPassByName() could not find the specified pass.", hr);
}

ID3DX11EffectVariable* variable = mEffect->GetVariableByName("WorldViewProjection");
if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName() could not find the specified variable.", hr);
}

mWvpVariable = variable->AsMatrix();
if (mWvpVariable->IsValid() == false)
{
    throw GameException("Invalid effect variable cast.");
}

variable = mEffect->GetVariableByName("ColorTexture");
if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName() could not find the specified variable.", hr);
}

mColorTextureVariable = variable->AsShaderResource();
if (mColorTextureVariable->IsValid() == false)
{
    throw GameException("Invalid effect variable cast.");
}

// Create the input layout
D3DX11_PASS_DESC passDesc;
mPass->GetDesc(&passDesc);

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
```

```

    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
    };
}

if (FAILED(hr = mGame->Direct3DDevice()->CreateInputLayout(
InputElementDescriptions, ARRAYSIZE(inputElementDescriptions),
passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
&mInputLayout)))
{
    throw GameException("ID3D11Device::CreateInputLayout() failed.", hr);
}

// Load the model
std::unique_ptr<Model> model(new Model(*mGame,
"Content\\Models\\Sphere.obj", true));

// Create the vertex and index buffers
Mesh* mesh = model->Meshes().at(0);
CreateVertexBuffer(mGame->Direct3DDevice(), *mesh,
&mVertexBuffer);
mesh->CreateIndexBuffer(&mIndexBuffer);
mIndexCount = mesh->Indices().size();

// Load the texture
std::wstring textureName =
L"Content\\Textures\\EarthComposite.jpg";
if (FAILED(hr = DirectX::CreateWICTextureFromFile(mGame->
Direct3DDevice(), mGame->Direct3DDeviceContext(),
textureName.c_str(),
nullptr, &mTextureShaderResourceView)))
{
    throw GameException("CreateWICTextureFromFile() failed.", hr);
}
}

void TexturedModelDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =

```

```

mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(TextureMappingVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_FORMAT
R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mWvpVariable->SetMatrix(reinterpret_cast<const float*>
(&wvp));
    mColorTextureVariable-
>SetResource(mTextureShaderResourceView);

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}

void TexturedModelDemo::CreateVertexBuffer(ID3D11Device* device,
const Mesh& mesh, ID3D11Buffer** vertexBuffer) const
{
    const std::vector<XMFLOAT3>& sourceVertices =
mesh.Vertices();

    std::vector<TextureMappingVertex> vertices;
    vertices.reserve(sourceVertices.size());

    std::vector<XMFLOAT3>* textureCoordinates = mesh.
TextureCoordinates().at(0);
    assert(textureCoordinates->size() == sourceVertices.size());

    for (UINT i = 0; i < sourceVertices.size(); i++)
    {
        XMFLOAT3 position = sourceVertices.at(i);
        XMFLOAT3 uv = textureCoordinates->at(i);
        vertices.push_back(TextureMappingVertex(XMFLOAT4(position
position.y, position.z, 1.0f), XMFLOAT2(uv.x, uv.y)));
    }
}

```

```

    }

    D3D11_BUFFER_DESC vertexBufferDesc;
    ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
    vertexBufferDesc.ByteWidth = sizeof(TextureMappingVertex) * 
vertices.size();
    vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
    vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

    D3D11_SUBRESOURCE_DATA vertexSubResourceData;
    ZeroMemory(&vertexSubResourceData,
    sizeof(vertexSubResourceData));
    vertexSubResourceData.pSysMem = &vertices[0];
    if (FAILED(device->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, vertexBuffer)))
    {
        throw GameException("ID3D11Device::CreateBuffer() 
failed.");
    }
}

```

---

In the `TexturedModelDemo::Initialize()` method, the `TextureMapping.fx` file is compiled, the effect object is created, and the `main11` technique and `p0` pass are located. Then the `WorldViewProjection` and `ColorTexture` variables are queried. Next, the input layout is created. Note the updated `D3D11_INPUT_ELEMENT_DESC` array that specifies the `POSITION` and `TEXCOORD` semantics for the two shader inputs.

Next, the `Sphere.obj` model is loaded and the vertex and index buffers are created. Note that the `TexturedModelDemo::CreateVertexBuffer()` method now builds `TextureMappingVertex` objects with positions and texture coordinates read from the model. No vertex colors are used for this shader.

Finally, the `EarthComposite.jpg` file is loaded into the `mTextureShaderResourceView` member using the `CreateWICTextureFromFile()` function. This function resides in the DirectX namespace and is declared in the `WICTextureLoader.h` header file. This header file comes from the DirectXTK library, which [Chapter 3, “Tools of the Trade,”](#) introduced. Be sure you’ve set up your projects to correctly reference this library. You can find the `EarthComposite.jpg` file on the companion website.

The `TexturedModelDemo::Draw()` method is similar to the original model demo. It now uses the size of the `TextureMappingVertex` struct when calling the `ID3D11DeviceContext::IASetVertexBuffers()` method and calls the `SetResource()` method of the `mColorTextureVariable` member to update the texture.

[Figure 15.2](#) shows the output of the textured model demo.



**Figure 15.2** The output of the textured model demo. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

## Summary

In this chapter, you learned how to use 3D models in your applications. You employed the Open Asset Import Library to read various model file formats, and you authored a set of classes to represent models at runtime. Then you wrote two demos to exercise this system, and you learned how to apply 2D textures to your models.

In the next chapter, you author a material system to support the myriad shaders you've authored and facilitate code reuse.

## Exercises

1. Create a build-time content pipeline from the code you've written in this chapter. This can be as simple as a command-line application that accepts a single input file and produces a single output, or your application can operate on a batch of file inputs. Ideally, your content pipeline will be presented through a graphical user interface.
2. Experiment with loading various model formats. For example, write demos to load .x and .3ds files (DirectX and 3D Studio Max formats). If your models use .dds texture files, investigate the `CreateDDSTextureFromFile()` method from the DirectXTK library.

# Chapter 16. Materials

In this chapter, you develop a set of types to encapsulate and simplify the interactions with effects. Then you exercise these types through a set of demo applications. You also create a reusable component for rendering skyboxes.

## Motivation

If you examine the last several demos (the triangle demo, the cube demo, and the two model demos), you'll see quite a bit of duplicated code: the shader compilation, the effect object creation, the technique, pass and variable lookup, input layout creation, and vertex and index buffer creation—all this code is repeated. If you're like me, you're itching to modularize this code. Furthermore, the types and data members for interfacing with the `BasicEffect` shader (the `BasicEffectVertex` structure, `mEffect`, `mTechnique`, `mPass`, and `mWvpVariable`) are duplicated between demos. The `BasicEffect` shader is one of the simplest effects you can apply. Consider some of the more sophisticated shaders you've authored. You might have a dozen or more shader variables and multiple techniques and passes. Clearly, you don't want to duplicate the data members and initialization of such shaders each time you want to use them. This calls for a reusable data structure that can represent a specific effect.

Recall the NVIDIA FX Composer terminology for describing an instance of an effect: **material**. We adopt this terminology to create a set of data structures that support interactions with a given effect. A `Material` class is the root structure for this system. Along the way, you modularize the systems for compiling shaders and creating vertex buffers, and you learn about compiling shaders at build time.

The material system consists of the following classes: `Effect`, `Technique`, `Pass`, `Variable`, and `Material`. The `Effect`, `Technique`, `Pass`, and `Variable` classes are thin wrappers around corresponding Effects 11 types (they add functionality and simplify the Effects 11 library usage). A `Material` object references a single `Effect`. The next few sections discuss these types in detail.

## The Effect Class

The `Effect` class wraps the `ID3DX11Effect` type (the root interface of the Effects 11 library) and exposes the contained techniques and shader variables through standard template library (STL) maps. The `Effect` class also encapsulates the code for compiling and loading shaders. Essentially, this is just a helper class that makes using effects a bit easier. [Listing 16.1](#) presents the declaration of the `Effect` class.

### **Listing 16.1** The `Effect.h` Header File

[Click here to view code image](#)

```
#pragma once

#include "Common.h"
#include "Technique.h"
#include "Variable.h"
```

```
namespace Library
{
    class Game;

    class Effect
    {
    public:
        Effect(Game& game);
        virtual ~Effect();

        static void CompileEffectFromFile(ID3D11Device*
direct3DDevice, ID3DX11Effect** effect, const std::wstring&
filename);
        static void LoadCompiledEffect(ID3D11Device*
direct3DDevice, ID3DX11Effect** effect, const std::wstring&
filename);

        Game& GetGame();
        ID3DX11Effect* GetEffect() const;
        void SetEffect(ID3DX11Effect* effect);
        const D3DX11_EFFECT_DESC& EffectDesc() const;
        const std::vector<Technique*>& Techniques() const;
        const std::map<std::string, Technique*>&
TechniquesByName()
const;
        const std::vector<Variable*>& Variables() const;
        const std::map<std::string, Variable*>&
VariablesByName()
const;

        void CompileFromFile(const std::wstring& filename);
        void LoadCompiledEffect(const std::wstring& filename);

    private:
        Effect(const Effect& rhs);
        Effect& operator=(const Effect& rhs);

        void Initialize();

        Game& mGame;
        ID3DX11Effect* mEffect;
        D3DX11_EFFECT_DESC mEffectDesc;
        std::vector<Technique*> mTechniques;
        std::map<std::string, Technique*> mTechniquesByName;
        std::vector<Variable*> mVariables;
        std::map<std::string, Variable*> mVariablesByName;
```

```
};
```

```
}
```

The most interesting elements of the Effect class are the `CompileFromFile()`, `LoadCompiledEffect()`, and `Initialize()` methods. [Listing 16.2](#) shows these implementations. The remaining implementation is made of single-line accessors, the destructor, and nonstatic pass-through methods for compiling and loading effects. For brevity, the listing omits these. You can find the full source code on the companion website.

## **Listing 16.2** The Effect Class Implementation (Abbreviated)

[Click here to view code image](#)

```
void Effect::CompileEffectFromFile(ID3D11Device* direct3DDevice,
ID3DX11Effect** effect, const std::wstring& filename)
{
    UINT shaderFlags = 0;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3DCOMPILE_DEBUG;
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

    ID3D10Blob* compiledShader = nullptr;
    ID3D10Blob* errorMessages = nullptr;
    HRESULT hr = D3DCompileFromFile(filename.c_str(), nullptr,
nullptr,
nullptr, "fx_5_0", shaderFlags, 0, &compiledShader,
&errorMessages);
    if (errorMessages != nullptr)
    {
        GameException ex((char*)errorMessages-
>GetBufferPointer(), hr);
        ReleaseObject(errorMessages);

        throw ex;
    }

    if (FAILED(hr))
    {
        throw GameException("D3DX11CompileFromFile() failed.",
hr);
    }

    hr = D3DX11CreateEffectFromMemory(compiledShader-
>GetBufferPointer(), compiledShader->GetBufferSize(), NULL,
direct3DDevice, effect);
}
```

```

    if (FAILED(hr))
    {
        throw GameException("D3DX11CreateEffectFromMemory()
failed.",
hr);
    }

    ReleaseObject(compiledShader);
}

void Effect::LoadCompiledEffect(ID3D11Device* direct3DDevice,
ID3DX11Effect** effect, const std::wstring& filename)
{
    std::vector<char> compiledShader;
    Utility::LoadBinaryFile(filename, compiledShader);

    HRESULT hr =
D3DX11CreateEffectFromMemory(&compiledShader.front(),
compiledShader.size(), NULL, direct3DDevice, effect);
    if (FAILED(hr))
    {
        throw GameException("D3DX11CreateEffectFromMemory()
failed."
, hr);
    }
}
void Effect::Initialize()
{
    HRESULT hr = mEffect->GetDesc(&mEffectDesc);
    if (FAILED(hr))
    {
        throw GameException("ID3DX11Effect::GetDesc() failed.",
hr);
    }

    for (UINT i = 0; i < mEffectDesc.Techniques; i++)
    {
        Technique* technique = new Technique(mGame, *this,
mEffect->GetTechniqueByIndex(i));
        mTechniques.push_back(technique);
        mTechniquesByName.insert(std::pair<std::string,
Technique*>(technique->Name(), technique));
    }

    for (UINT i = 0; i < mEffectDesc.GlobalVariables; i++)
    {
        Variable* variable = new Variable(*this,

```

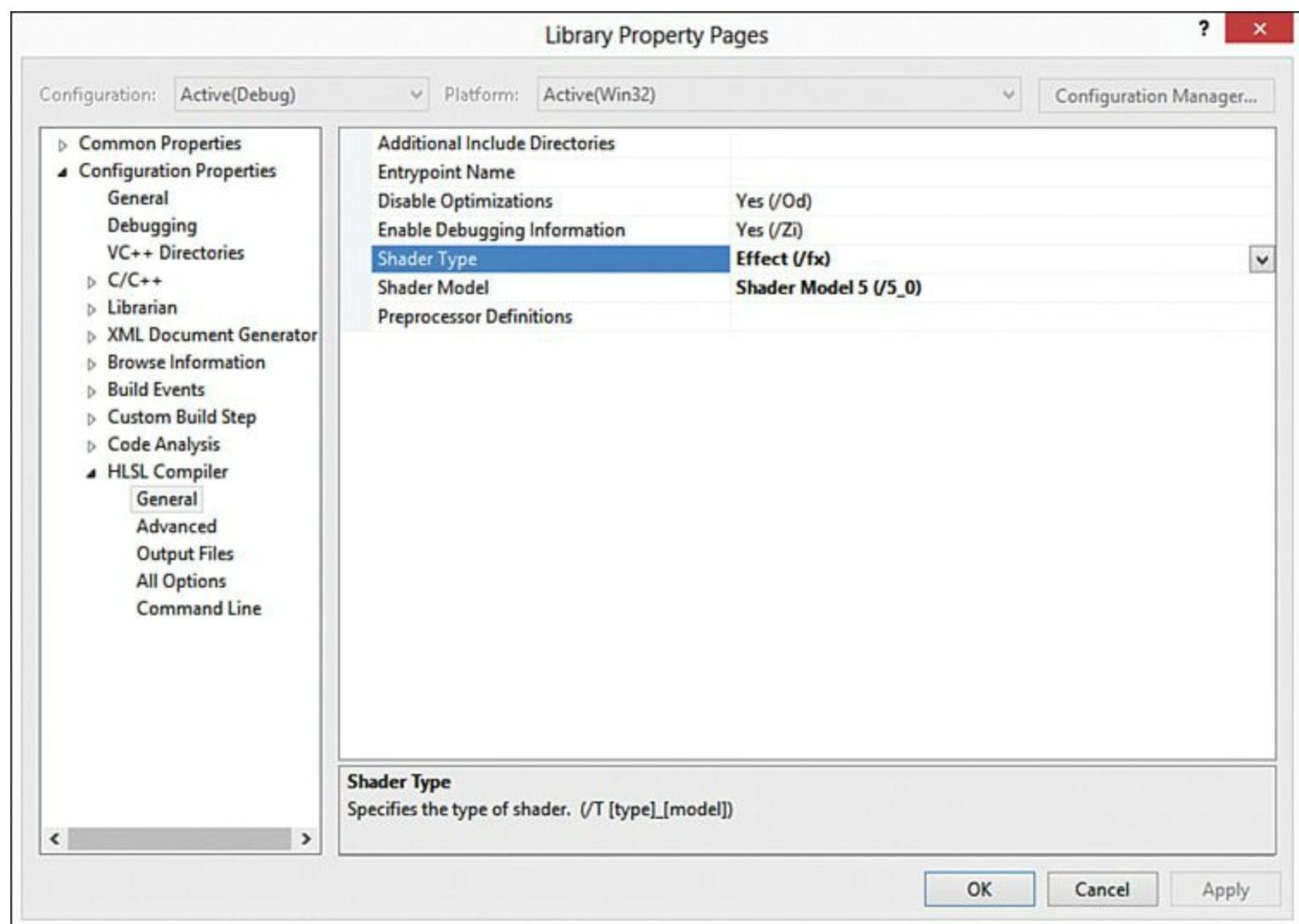
```

mEffect->GetVariableByIndex(i));
    mVariables.push_back(variable);
    mVariablesByName.insert(std::pair<std::string,
Variable*>(variable->Name(), variable));
}
}

```

You've seen the code in the `Effect::CompileEffectFromFile()` method before. This is just runtime shader compilation encapsulated within a class that stores the associated `ID3DX11Effect` object. New is the `Effect::LoadCompiledEffect()` method, which loads a **compiled shader object** into memory and invokes the same `D3DX11CreateEffectFromMemory()` method to instantiate an effect object. A compiled shader object is the output of the compilation process and can be saved to a file with a `.cso` extension. So how do you produce a `.cso` file to load?

First, add the `BasicEffect.fx` file to one of your Visual Studio projects, either the Library project or the Game project. Then open the Property Pages for the project and navigate to **Configuration Properties, HLSL Compiler, General**. Set the Shader Type parameter to Effect (/fx) and the Shader Model parameter to Shader Model 5 (/5\_0), as in [Figure 16.1](#). Be sure to do this for both the debug and release configurations.



**Figure 16.1** Visual Studio's property pages for HLSL compiler parameters.

Now when you build your Visual Studio project, .cso files will be built for the included effects. By default, these .cso files will be written to the same directory as your library or executable (the Visual Studio \$(OutDir) macro). If you prefer a different location, modify the **Configuration Properties**, **HLSL Compiler**, **Output Files**, Object File Name parameter. The projects on the book's companion website use this setting:

[Click here to view code image](#)

```
$(OutDir)Content\Effects\% (Filename).cso
```

Furthermore, if you include your source .fx files in your Library project, be sure that all .cso files are copied to a path accessible by your application's executable. A post-build step can handle this. The last Effect topic to discuss is the Initialize() method (see [Listing 16.2](#)). This method retrieves the D3DX11\_EFFECT\_DESC structure (the description structure for the effect), iterates through the available techniques and shader variables, and instantiates their wrapping classes.

## The Technique Class

The Technique class is the simplest of the Effects 11 wrapping types. It just exposes the name of the technique and its associated passes. [Listings 16.3](#) presents the declaration of the Technique class.

### **Listing 16.3** The Technique.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"
#include "Pass.h"

namespace Library
{
    class Game;
    class Effect;

    class Technique
    {
public:
    Technique(Game& game, Effect& effect,
ID3DX11EffectTechnique* technique);
    ~Technique();

    Effect& GetEffect();
    ID3DX11EffectTechnique* GetTechnique() const;
    const D3DX11_TECHNIQUE_DESC& TechniqueDesc() const;
    const std::string& Name() const;
    const std::vector<Pass*>& Passes() const;
```

```

        const std::map<std::string, Pass*>& PassesByName()
const;

private:
    Technique(const Technique& rhs);
    Technique& operator=(const Technique& rhs);

    Effect& mEffect;
    ID3DX11EffectTechnique* mTechnique;
    D3DX11_TECHNIQUE_DESC mTechniqueDesc;
    std::string mName;
    std::vector<Pass*> mPasses;
    std::map<std::string, Pass*> mPassesByName;
}
}

```

---

The Technique class has only one interesting element: the constructor, which iterates through the technique's passes, instantiates the wrapping Pass objects, and adds them to the STL containers. [Listing 16.4](#) presents this implementation.

#### **Listing 16.4** The Technique Class Constructor

[Click here to view code image](#)

---

```

Technique::Technique(Game& game, Effect& effect,
ID3DX11EffectTechnique* technique)
: mEffect(effect), mTechnique(technique), mTechniqueDesc(),
mName(), mPasses(), mPassesByName()
{
    mTechnique->GetDesc(&mTechniqueDesc);
    mName = mTechniqueDesc.Name;

    for (UINT i = 0; i < mTechniqueDesc.Passes; i++)
    {
        Pass* pass = new Pass(game, *this,
mTechnique->GetPassByIndex(i));
        mPasses.push_back(pass);
        mPassesByName.insert(std::pair<std::string, Pass*>(pass-
>Name(), pass));
    }
}

```

---

## The Pass Class

The Pass class follows the same patterns as the Effect and Technique classes. It wraps the corresponding Effects 11 type (`ID3DX11EffectPass`) and adds a bit of functionality.

Specifically, the Pass class encapsulates input layout creation. Using a Pass object to create the input layout makes sense because it contains the input signature (the definition of the shader inputs for the associated vertex shader). [Listing 16.5](#) presents the declaration of the Pass class.

## **Listing 16.5** The Pass.h Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;
    class Technique;

    class Pass
    {
        public:
            Pass(Game& game, Technique& technique,
ID3DX11EffectPass*
pass);

            Technique& GetTechnique();
            ID3DX11EffectPass* GetPass() const;
            const D3DX11_PASS_DESC& PassDesc() const;
            const std::string& Name() const;

            void CreateInputLayout(const D3D11_INPUT_ELEMENT_DESC*
InputElementDesc, UINT numElements, ID3D11InputLayout
**inputLayout);
            void Apply(UINT flags, ID3D11DeviceContext* context);

        private:
            Pass(const Pass& rhs);
            Pass& operator=(const Pass& rhs);

            Game& mGame;
            Technique& mTechnique;
            ID3DX11EffectPass* mPass;
            D3DX11_PASS_DESC mPassDesc;
            std::string mName;
    };
}
```

---

The interesting elements of the Pass class implementation are the constructor and the

CreateInputLayout() method (see [Listing 16.6](#)).

## **Listing 16.6** The Pass Class Implementation (Abbreviated)

[Click here to view code image](#)

---

```
Pass::Pass(Game& game, Technique& technique, ID3DX11EffectPass*  
pass)  
    : mGame(game), mTechnique(technique), mPass(pass),  
mPassDesc(),  
    mName()  
{  
    mPass->GetDesc(&mPassDesc);  
    mName = mPassDesc.Name;  
}  
  
void Pass::CreateInputLayout(const D3D11_INPUT_ELEMENT_DESC*  
InputElementDesc, UINT numElements, ID3D11InputLayout  
**inputLayout)  
{  
    HRESULT hr = mGame.Direct3DDevice()->CreateInputLayout(input  
ElementDesc, numElements, mPassDesc.pIAInputSignature,  
mPassDesc.  
IAInputSignatureSize, inputLayout);  
    if (FAILED(hr))  
    {  
        throw GameException("ID3D11Device::CreateInputLayout()  
failed.", hr);  
    }  
}
```

---

## The Variable Class

The Variable class encapsulates the ID3DX11EffectVariable type and adds some friendly syntax for updating a shader variable. Recall that the ID3DX11EffectVariable interface represents any type of shader variable. Therefore, the Variable class exposes the associated D3DX11\_EFFECT\_VARIABLE\_DESC and D3DX11\_EFFECT\_TYPE\_DESC structures that provide details for a variable's semantic, annotations, and the internals of the variable's actual data type. [Listing 16.7](#) presents the declaration of the Variable class.

## **Listing 16.7** The Variable.h Header File

[Click here to view code image](#)

---

```
#pragma once  
  
#include "Common.h"
```

```

namespace Library
{
    class Effect;

    class Variable
    {
public:
    Variable(Effect& effect, ID3DX11EffectVariable* variable);

    Effect& GetEffect();
    ID3DX11EffectVariable* GetVariable() const;
    const D3DX11_EFFECT_VARIABLE_DESC& VariableDesc() const;
    ID3DX11EffectType* Type() const;
    const D3DX11_EFFECT_TYPE_DESC& TypeDesc() const;
    const std::string& Name() const;

    Variable& operator<<(CXMMATRIX value);
    Variable& operator<<(ID3D11ShaderResourceView* value);
    Variable& operator<<(FXMVECTOR value);
    Variable& operator<<(float value);

private:
    Variable(const Variable& rhs);
    Variable& operator=(const Variable& rhs);

    Effect& mEffect;
    ID3DX11EffectVariable* mVariable;
    D3DX11_EFFECT_VARIABLE_DESC mVariableDesc;
    ID3DX11EffectType* mType;
    D3DX11_EFFECT_TYPE_DESC mTypeDesc;
    std::string mName;
};

}

```

---

The most interesting aspects of the `Variable` class implementation are the `<<` operator overloads. These methods provide a syntax for updating shader variables that draws its inspiration from the `iostream` library. For example, to update the `WorldViewProjection` projection matrix of a `BasicEffect` shader, you would write code such as the following:

[Click here to view code image](#)

```

XMMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
XMMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
mBasicMaterial->WorldViewProjection() << wvp;

```

In this example, the `mBasicMaterial::WorldViewProjection()` accessor returns a reference to a `Variable` object. [Listing 16.8](#) presents the code for these overloaded operators.

## **Listing 16.8 Overloaded Operators for the Variable Class**

[Click here to view code image](#)

---

```
Variable& Variable::operator<<(CXMMATRIX value)
{
    ID3DX11EffectMatrixVariable* variable = mVariable-
>AsMatrix();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetMatrix(reinterpret_cast<const float*>(&value));
    return *this;
}

Variable& Variable::operator<<(ID3D11ShaderResourceView* value)
{
    ID3DX11EffectShaderResourceVariable* variable =
mVariable->AsShaderResource();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetResource(value);

    return *this;
}

Variable& Variable::operator<<(FXMVECTOR value)
{
    ID3DX11EffectVectorVariable* variable = mVariable-
>AsVector();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetFloatVector(reinterpret_cast<const float*>
(&value));
```

```

        return *this;
    }

Variable& Variable::operator<<(float value)
{
    ID3DX11EffectScalarVariable* variable = mVariable-
>AsScalar();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetFloat(value);

    return *this;
}

```

---

With the presented syntax, you get the benefit of runtime type checking, but you can store generic Variable objects within the Effect class. Note that [Listing 16.8](#) doesn't include overloads for every possible data type—just those that are useful for the next few demos.

## The Material Class

The Effect class contains Variable and Technique objects, and a Technique contains Pass objects. This type of system needs one more class that pulls everything together: Material. The Material class references an Effect and adds helper methods for creating input layouts and vertex buffers that are specific to the associated Effect. The Material class is intended as a base class in which the derived class represents one particular effect. In the upcoming demos, for example, you will create BasicMaterial and SkyboxMaterial classes that aid the interaction with the BasicEffect.fx and Skybox.fx effects. These derived classes cache the variables of an effect and expose them through public accessors. This is accomplished through a set of macros that makes creating new materials simple. In fact, you could create a utility to read a .fx file and generate the corresponding .h and .cpp files for the material.

[Listing 16.9](#) presents the declaration of the Material class.

### **Listing 16.9** The Material.h Header File

[Click here to view code image](#)

---

```

#pragma once

#include "Common.h"
#include "Effect.h"

namespace Library
{
    class Model;

```

```

class Mesh;

class Material : public RTTI
{
    RTTI_DECLARATIONS(Material, RTTI)

public:
    Material();
    Material(const std::string& defaultTechniqueName);
    virtual ~Material();

    Variable* operator[] (const std::string& variableName);
    Effect* GetEffect() const;
    Technique* CurrentTechnique() const;
    void SetCurrentTechnique(Technique* currentTechnique);
    const std::map<Pass*, ID3D11InputLayout*>&
    InputLayouts()
    const;

        virtual void Initialize(Effect* effect);
        virtual void CreateVertexBuffer(ID3D11Device* device,
    const
    Model& model, std::vector<ID3D11Buffer*>& vertexBuffers) const;
        virtual void CreateVertexBuffer(ID3D11Device* device,
    const
    Mesh& mesh, ID3D11Buffer** vertexBuffer) const = 0;
        virtual UINT VertexSize() const = 0;

protected:
    Material(const Material& rhs);
    Material& operator=(const Material& rhs);

        virtual void CreateInputLayout(const std::string&
techniqueName, const std::string& passName,
D3D11_INPUT_ELEMENT_DESC*
inputElementDescriptions, UINT inputElementDescriptionCount);

    Effect* mEffect;
    Technique* mCurrentTechnique;
    std::string mDefaultTechniqueName;
    std::map<Pass*, ID3D11InputLayout*> mInputLayouts;
};

#define MATERIAL_VARIABLE_DECLARATION(VariableName) \
public: \
    Variable& VariableName()

```

```

const;
    \ 
private:
    Variable* m ## VariableName;

#define MATERIAL_VARIABLE_DEFINITION(Material,
VariableName) \
    Variable& Material::VariableName()
const \
{
    return *m ## \
VariableName;
}

#define MATERIAL_VARIABLE_INITIALIZATION(VariableName) m ## \
VariableName(NULL)

#define \
MATERIAL_VARIABLE_RETRIEVE(VariableName) \
    m ## VariableName = mEffect->VariablesByName(). \
at(#VariableName);
}

```

---

A `Material` is initialized after instantiation, through the `Material::Initialize()` method. Initialization associates the effect to the material and sets the notion of a current technique (the `Material::mCurrentTechnique` member). The current technique is simply the effect technique that should be applied during rendering if not otherwise specified. The `Material::Initialize()` method sets the current technique through the `Material::mDefaultTechniqueName` data member (provided on instantiation).

The `Material::mInputsLayouts` data member is an STL map that's populated through the `Material::CreateInputLayout()` method. It uses the associated `Pass` object as the map's key and should be used for calls to `ID3DX11DeviceContext::IASetInputLayout()`. The following code provides sample usage:

[Click here to view code image](#)

```

Pass* pass = mBasicMaterial->CurrentTechnique()->Passes().at(0);
ID3D11InputLayout* inputLayout = mBasicMaterial->InputLayouts(). \
at(pass);
direct3DDeviceContext->IASetInputLayout(inputLayout);

```

Two `Material::CreateVertexBuffer()` methods are in use here, one that builds a set of vertex buffers from a `Model` object and another that builds a single vertex buffer from a `Mesh`. The latter is a pure virtual method that must be specified for each derived class. This makes sense because each effect (represented by the material) will likely have a unique input signature. The model-version of `Material::CreateVertexBuffer()` iteratively invokes the mesh-version to create a collection of vertex buffers.

`Material::VertexSize()` is another pure virtual method that's intended to return the size (in

bytes) of a vertex that's compatible with the material.

Finally, examine the four macros at the end of [Listing 16.9](#):

[Click here to view code image](#)

```
MATERIAL_VARIABLE_DECLARATION  
MATERIAL_VARIABLE_DEFINITION  
MATERIAL_VARIABLE_INITIALIZATION  
MATERIAL_VARIABLE_RETRIEVE
```

These macros are used within the derived material classes to expose cached Variable objects through public accessors. We cover their usage shortly.

[Listing 16.10](#) presents the full implementation of the Material class.

## **Listing 16.10** The Material.cpp File

[Click here to view code image](#)

---

```
#include "Material.h"  
#include "GameException.h"  
#include "Model.h"  
  
namespace Library  
{  
    RTTI_DEFINITIONS(Material)  
  
    Material::Material()  
        : mEffect(nullptr), mCurrentTechnique(nullptr),  
          mDefaultTechniqueName(), mInputLayouts()  
    {  
    }  
  
    Material::Material(const std::string& defaultTechniqueName)  
        : mEffect(nullptr), mCurrentTechnique(nullptr),  
          mDefaultTechniqueName(defaultTechniqueName),  
          mInputLayouts()  
    {  
    }  
  
    Material::~Material()  
    {  
        for (std::pair<Pass*, ID3D11InputLayout*> inputLayout :  
             mInputLayouts)  
        {  
            ReleaseObject(inputLayout.second);  
        }  
    }  
}
```

```
Variable* Material::operator[] (const std::string&
variableName)
{
    const std::map<std::string, Variable*>& variables =
mEffect->VariablesByName();
    Variable* foundVariable = nullptr;

    std::map<std::string, Variable*>::const_iterator found =
variables.find(variableName);
    if (found != variables.end())
    {
        foundVariable = found->second;
    }

    return foundVariable;
}

Effect* Material::GetEffect() const
{
    return mEffect;
}

Technique* Material::CurrentTechnique() const
{
    return mCurrentTechnique;
}

void Material::SetCurrentTechnique(Technique*
currentTechnique)
{
    mCurrentTechnique = currentTechnique;
}

const std::map<Pass*, ID3D11InputLayout*>&
Material::InputLayouts()
const
{
    return mInputLayouts;
}

void Material::Initialize(Effect* effect)
{
    mEffect = effect;
    assert(mEffect != nullptr);

    Technique* defaultTechnique = nullptr;
    assert(mEffect->Techniques().size() > 0);
```

```

    if (mDefaultTechniqueName.empty() == false)
    {
        defaultTechnique = mEffect->TechniquesByName().at(mDefaultTechniqueName);
        assert(defaultTechnique != nullptr);
    }
    else
    {
        defaultTechnique = mEffect->Techniques().at(0);
    }

    SetCurrentTechnique(defaultTechnique);
}

void Material::CreateVertexBuffer(ID3D11Device* device,
const
Model& model, std::vector<ID3D11Buffer*>& vertexBuffers) const
{
    vertexBuffers.reserve(model.Meshes().size());
    for (Mesh* mesh : model.Meshes())
    {
        ID3D11Buffer* vertexBuffer;
        CreateVertexBuffer(device, *mesh, &vertexBuffer);
        vertexBuffers.push_back(vertexBuffer);
    }
}

void Material::CreateInputLayout(const std::string&
techniqueName, const std::string& passName,
D3D11_INPUT_ELEMENT_DESC*
InputElementDescriptions, UINT inputElementDescriptionCount)
{
    Technique* technique = mEffect->TechniquesByName().at(techniqueName);
    assert(technique != nullptr);

    Pass* pass = technique->PassesByName().at(passName);
    assert(pass != nullptr);

    ID3D11InputLayout* inputLayout;
    pass->CreateInputLayout(inputElementDescriptions,
    inputElementDescriptionCount, &inputLayout);

    mInputLayouts.insert(std::pair<Pass*,
ID3D11InputLayout*>(pass, inputLayout));
}
}

```

## A Basic Effect Material

The best way to understand the material system is through application. In this section, you create a material for the `BasicEffect.fx` shader and use it to render a sphere. [Listing 16.11](#) presents the declaration of the `BasicMaterial` class and an associated `BasicMaterialVertex` structure.

### **Listing 16.11** The `BasicMaterial.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "Common.h"
#include "Material.h"

namespace Library
{
    typedef struct _BasicMaterialVertex
    {
        XMFLOAT4 Position;
        XMFLOAT4 Color;

        _BasicMaterialVertex() { }

        _BasicMaterialVertex(const XMFLOAT4& position, const
XMFLOAT4&
color)
            : Position(position), Color(color) { }
    } BasicMaterialVertex;

    class BasicMaterial : public Material
    {
        RTTI_DECLARATIONS(BasicMaterial, Material)

        MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)

    public:
        BasicMaterial();

        virtual void Initialize(Effect* effect) override;
        virtual void CreateVertexBuffer(ID3D11Device* device,
const
Mesh& mesh, ID3D11Buffer** vertexBuffer) const override;
        void CreateVertexBuffer(ID3D11Device* device,
BasicMaterialVertex* vertices, UINT vertexCount, ID3D11Buffer** vertexBuffer) const;
```

```
    virtual UINT VertexSize() const override;  
};  
}
```

---

The `BasicMaterialVertex` replaces the duplicated `BasicEffectVertex` structures you created in previous demos. Its name and residence within the `BasicMaterial.h` file indicates its intended usage.

The `BasicMaterial` class has little declaration outside what it inherits from the base `Material` type. It declares the single `WorldViewProjection` shader variable through the `MATERIAL_VARIABLE_DECLARATION` macro. Then it overrides the `Material::Initialize()` and `Material::VertexSize()` methods, as well as the mesh-version of the `Material::CreateVertexBuffer()` method. It also adds a new `CreateVertexBuffer()` method that builds a vertex buffer from an array of `BasicMaterialVertex` objects. These declarations set the pattern for all derived material classes.

[Listing 16.12](#) presents the implementation of the `BasicMaterial` class.

### **Listing 16.12** The `BasicMaterial.cpp` File

[Click here to view code image](#)

```
#include "BasicMaterial.h"  
#include "GameException.h"  
#include "Mesh.h"  
#include "ColorHelper.h"  
  
namespace Library  
{  
    RTTI_DEFINITIONS(BasicMaterial)  
  
    BasicMaterial::BasicMaterial()  
        : Material("main11"),  
          MATERIAL_VARIABLE_INITIALIZATION(WorldViewProjection)  
    {}  
}  
  
MATERIAL_VARIABLE_DEFINITION(BasicMaterial,  
WorldViewProjection)  
  
void BasicMaterial::Initialize(Effect* effect)  
{  
    Material::Initialize(effect);  
  
    MATERIAL_VARIABLE_RETRIEVE(WorldViewProjection)  
    D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
```

```

    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
    };
}

CreateInputLayout("main11", "p0",
InputElementDescriptions, ARR
AYSIZE(ElementDescriptions));
}

void BasicMaterial::CreateVertexBuffer(ID3D11Device* device,
const
Mesh& mesh, ID3D11Buffer** vertexBuffer) const
{
    const std::vector<XMFLOAT3>& sourceVertices =
mesh.Vertices();

    std::vector<BasicMaterialVertex> vertices;
    vertices.reserve(sourceVertices.size());
    if (mesh.VertexColors().size() > 0)
    {
        std::vector<XMFLOAT4>* vertexColors =
mesh.VertexColors().
at(0);
        assert(vertexColors->size() ==
sourceVertices.size());

        for (UINT i = 0; i < sourceVertices.size(); i++)
        {
            XMFLOAT3 position = sourceVertices.at(i);
            XMFLOAT4 color = vertexColors->at(i);
            vertices.push_back(BasicMaterialVertex(XMFLOAT4
(position.x, position.y, position.z, 1.0f), color));
        }
    }
    else
    {
        XMFLOAT4 color = XMFLOAT4(reinterpret_cast<const
float*>(&ColorHelper::White));
        for (UINT i = 0; i < sourceVertices.size(); i++)
        {
            XMFLOAT3 position = sourceVertices.at(i);
            vertices.push_back(BasicMaterialVertex(XMFLOAT4
(position.x, position.y, position.z, 1.0f), color));
        }
    }
}

```

```

        }

        CreateVertexBuffer(device, &vertices[0],
vertices.size(),
vertexBuffer);
    }

    void BasicMaterial::CreateVertexBuffer(ID3D11Device* device,
BasicMaterialVertex* vertices, UINT vertexCount, ID3D11Buffer** vertexBuffer) const
{
    D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = VertexSize() * vertexCount;
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

    D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof
(vertexSubResourceData));
    vertexSubResourceData.pSysMem = vertices;
    if (FAILED(device->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, vertexBuffer)))
    {
        throw GameException("ID3D11Device::CreateBuffer()
failed.");
    }
}

UINT BasicMaterial::VertexSize() const
{
    return sizeof(BasicMaterialVertex);
}

```

The `BasicMaterial` constructor specifies `main11` as the default technique name and uses the `MATERIAL_VARIABLE_INITIALIZATION` macro to initialize the `WorldViewProjection` variable. Next, a `BasicMaterial::WorldViewProjection()` accessor is defined through the `MATERIAL_VARIABLE_DEFINITION` macro. All accessors created in this way return a reference to a `Variable` object stored within the associated `Effect` class. The `WorldViewProjection` variable reference is retrieved from the `Effect` class and cached using the `MATERIAL_VARIABLE_RETRIEVE` macro. Such variable retrieval has two benefits: First, caching obviates the need for repeated variable lookups. Second, it validates (at runtime) the expected variables within the effect.

Next, the normal array of `D3D11_INPUT_ELEMENT_DESC` structures is created and passed to the

`Material::CreateInputLayout()` method.

Now inspect the `BasicMaterial::CreateVertexBuffer()` methods. The mesh-version of this method has the same structure as for the model and textured model demos. Placing this code within the `BasicMaterial` class merely encapsulates it properly. The mesh-version calls the `BasicMaterialVertex` version of the `BasicMaterial::CreateVertexBuffer()` method, which actually creates the vertex buffer. In this way, you can build a vertex buffer for the `BasicEffect.fx` shader with a plain array of `BasicMaterialVertex` objects, a `Mesh`, or a `Model` object (using the model-version of the `CreateVertexBuffer()` method inherited from the `Material` class).

Finally, the `BasicMaterial` class provides an implementation for the `VertexSize()` method, which returns the size of the `BasicMaterialVertex` structure.

## A Basic Material Demo

With the material system in place, you can re-create the model demo with much less code. [Listing 16.13](#) presents the declaration of the `MaterialDemo` class.

### **Listing 16.13** The `MaterialDemo.h` Header File

[Click here to view code image](#)

---

```
#pragma once

#include "DrawableGameComponent.h"

using namespace Library;

namespace Library
{
    class Effect;
    class BasicMaterial;
}

namespace Rendering
{
    class MaterialDemo : public DrawableGameComponent
    {
        RTTI_DECLARATIONS(MaterialDemo, DrawableGameComponent)

    public:
        MaterialDemo(Game& game, Camera& camera);
        ~MaterialDemo();

        virtual void Initialize() override;
        virtual void Draw(const GameTime& gameTime) override;

    private:
}
```

```

MaterialDemo();
MaterialDemo(const MaterialDemo& rhs);
MaterialDemo& operator=(const MaterialDemo& rhs);

Effect* mBasicEffect;
BasicMaterial* mBasicMaterial;
ID3D11Buffer* mVertexBuffer;
ID3D11Buffer* mIndexBuffer;
UINT mIndexCount;

XMFLOAT4X4 mWorldMatrix;
};

}

```

---

Compare the `MaterialDemo` declaration to the `ModelDemo` class of the last chapter. Gone are the `mEffect`, `mTechnique`, `mPass`, `mWvpVariable`, and `mInputLayout` class members. The `BasicEffectVertex` structure declaration has likewise been removed.

The class implementation, in [Listing 16.14](#), is correspondingly simplified.

### **Listing 16.14** The `MaterialDemo.cpp` File

[Click here to view code image](#)

---

```

#include "MaterialDemo.h"
#include "Game.h"
#include "GameException.h"
#include "MatrixHelper.h"
#include "Camera.h"
#include "Utility.h"
#include "Model.h"
#include "Mesh.h"
#include "BasicMaterial.h"

namespace Rendering
{
    RTTI_DEFINITIONS(MaterialDemo)

    MaterialDemo::MaterialDemo(Game& game, Camera& camera)
        : DrawableGameComponent(game, camera),
          mBasicMaterial(nullptr), mBasicEffect(nullptr),
          mWorldMatrix
          (MatrixHelper::Identity),
          mVertexBuffer(nullptr), mIndexBuffer(nullptr),
          mIndexCount(0)
    {
    }

```

```
MaterialDemo::~MaterialDemo()
{
    DeleteObject(mBasicMaterial);
    DeleteObject(mBasicEffect);
    ReleaseObject(mVertexBuffer);
    ReleaseObject(mIndexBuffer);
}

void MaterialDemo::Initialize()
{
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str();

    // Load the model
    std::unique_ptr<Model> model(new Model(*mGame,
"Content\\
Models\\Sphere.obj", true));

    // Initialize the material
    mBasicEffect = new Effect(*mGame);
    mBasicEffect->LoadCompiledEffect(L"Content\\Effects\\
BasicEffect.cso");
    mBasicMaterial = new BasicMaterial();
    mBasicMaterial->Initialize(mBasicEffect);

    // Create the vertex and index buffers
    Mesh* mesh = model->Meshes().at(0);
    mBasicMaterial->CreateVertexBuffer(mGame-
>Direct3DDevice(),
*mesh, &mVertexBuffer);
    mesh->CreateIndexBuffer(&mIndexBuffer);
    mIndexCount = mesh->Indices().size();
}

void MaterialDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mBasicMaterial->CurrentTechnique()-
>Passes().
at(0);
    ID3D11InputLayout* inputLayout = mBasicMaterial-
>InputLayouts().at(pass);
```

```

    direct3DDeviceContext->IASetInputLayout(inputLayout);

    UINT stride = mBasicMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_
FORMAT_R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mBasicMaterial->WorldViewProjection() << wvp;

    pass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}
}

```

---

All the code for explicitly compiling shaders; finding techniques, passes, and variables; and creating vertex buffers has been removed from the `MaterialDemo` implementation. But the functionality of the demo hasn't been altered. This new format is the pattern for all future demos.

## A Skybox Material

We can reinforce the content in this chapter by creating a skybox material and an associated demo. Create a `SkyboxMaterial` using the declaration in [Listing 16.15](#). This matches the `Skybox.fx` shader you wrote in [Chapter 8, “Gleaming the Cube.”](#)

### **Listing 16.15 Declaration of the SkyboxMaterial Class**

[Click here to view code image](#)

---

```

class SkyboxMaterial : public Material
{
    RTTI_DECLARATIONS(SkyboxMaterial, Material)

    MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)
    MATERIAL_VARIABLE_DECLARATION(SkyboxTexture)

public:
    SkyboxMaterial();

    virtual void Initialize(Effect* effect) override;

```

```

    virtual void CreateVertexBuffer(ID3D11Device* device, const
Mesh&
mesh, ID3D11Buffer** vertexBuffer) const override;
    void CreateVertexBuffer(ID3D11Device* device, XMFLOAT4*
vertices,
UINT vertexCount, ID3D11Buffer** vertexBuffer) const;
    virtual UINT VertexSize() const override;
};

```

---

[Listing 16.16](#) presents an abbreviated implementation of the SkyboxMaterial class. It omits the CreateVertexBuffer() and VertexSize() methods for brevity. Visit the companion website for complete source code.

## Listing 16.16 Implementation of the SkyboxMaterial Class (Abbreviated)

[Click here to view code image](#)

---

```

SkyboxMaterial::SkyboxMaterial()
: Material("main11"),
MATERIAL_VARIABLE_INITIALIZATION(WorldViewProjection),
MATERIAL_VARIABLE_INITIALIZATION(SkyboxTexture)
{
}

MATERIAL_VARIABLE_DEFINITION(SkyboxMaterial,
WorldViewProjection)
MATERIAL_VARIABLE_DEFINITION(SkyboxMaterial, SkyboxTexture)

void SkyboxMaterial::Initialize(Effect* effect)
{
    Material::Initialize(effect);

    MATERIAL_VARIABLE_RETRIEVE(WorldViewProjection)
    MATERIAL_VARIABLE_RETRIEVE(SkyboxTexture)

    D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
0, D3D11_INPUT_PER_VERTEX_DATA, 0 }
    };

    CreateInputLayout("main11", "p0",
inputElementDescriptions,
ARRAYSIZE(inputElementDescriptions));
}

```

---

Notice the similarities between the SkyboxMaterial and the BasicMaterial implementations. The differences are the shader variables and the input layout. Also note that no vertex structure is created for the SkyboxMaterial. This is because only one shader input exists—the vertex position—and it's of type XMFLOAT4.

## A Skybox Component

To exercise the SkyboxMaterial class, create a reusable Skybox component with the declaration in [Listing 16.17](#).

### Listing 16.17 Declaration of the Skybox Class

[Click here to view code image](#)

```
class Skybox : public DrawableGameComponent
{
    RTTI_DECLARATIONS(Skybox, DrawableGameComponent)

public:
    Skybox(Game& game, Camera& camera, const std::wstring&
cubeMapFileName, float scale);
    ~Skybox();

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    Skybox();
    Skybox(const Skybox& rhs);
    Skybox& operator=(const Skybox& rhs);

    std::wstring mCubeMapFileName;
    Effect* mEffect;
    SkyboxMaterial* mMaterial;
    ID3D11ShaderResourceView* mCubeMapShaderResourceView;
    ID3D11Buffer* mVertexBuffer;
    ID3D11Buffer* mIndexBuffer;
    UINT mIndexCount;

    XMFLOAT4X4 mWorldMatrix;
    XMFLOAT4X4 mScaleMatrix;
};
```

The Skybox::mCubeMapFileName is initialized within the constructor, as is the Skybox::ScaleMatrix member (with a call to XMMatrixScaling()). The rest of the work is split between the Initialize(), Update() and Draw() methods (see [Listing 16.18](#)).

## Listing 16.18 Implementation of the Skybox Class (Abbreviated)

[Click here to view code image](#)

```
void Skybox::Initialize()
{
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

    std::unique_ptr<Model> model(new Model(*mGame,
"Content\\Models\\
Sphere.obj", true));

    mEffect = new Effect(*mGame);
    mEffect-
>LoadCompiledEffect(L"Content\\Effects\\Skybox.cso");

    mMaterial = new SkyboxMaterial();
    mMaterial->Initialize(mEffect);

    Mesh* mesh = model->Meshes().at(0);
    mMaterial->CreateVertexBuffer(mGame->Direct3DDevice(),
*mesh,
&mVertexBuffer);
    mesh->CreateIndexBuffer(&mIndexBuffer);
    mIndexCount = mesh->Indices().size();

    HRESULT hr = DirectX::CreateDDSTextureFromFile
(mGame->Direct3DDevice(), mCubeMapFileName.c_str(), nullptr,
&mCubeMapShaderResourceView);
    if (FAILED(hr))
    {
        throw GameException("CreateDDSTextureFromFile()
failed.", hr);
    }
}

void Skybox::Update(const GameTime& gameTime)
{
    const XMFLOAT3& position = mCamera->Position();

    XMStoreFloat4x4(&mWorldMatrix, XMLoadFloat4x4(&mScaleMatrix)
*
XMMatrixTranslation(position.x, position.y, position.z));
}

void Skybox::Draw(const GameTime& gametime)
{
```

```

ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

Pass* pass = mMaterial->CurrentTechnique()->Passes().at(0);
ID3D11InputLayout* inputLayout = mMaterial->InputLayouts().
at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);

UINT stride = mMaterial->VertexSize();
UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_FORMAT
R32_UINT, 0);

XMMATRIX wvp = XMLoadFloat4x4(&mWorldMatrix) * mCamera->
ViewMatrix() * mCamera->ProjectionMatrix();
mMaterial->WorldViewProjection() << wvp;
mMaterial->SkyboxTexture() << mCubeMapShaderResourceView;

pass->Apply(0, direct3DDeviceContext);

direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}

```

Within the `Skybox::Initialize()` method, the `Skybox.cso` file is loaded, as is the DDS file containing the cubemap. The `Skybox::Update()` method updates the world matrix by concatenating the scale matrix and a translation matrix built from the camera's position (recall that the skybox should always be centered on the camera). Finally, the `Skybox::Draw()` method renders the output. Note the updates to the two `SkyboxMaterial` variable accessors, `WorldViewProjection()` and `SkyboxTexture()`, using the `<<` operator syntax.

Add an instance of the `Skybox` class to your game components vector with calls such as the following:

[Click here to view code image](#)

```

mSkybox = new Skybox(*this, *mCamera, L"Content\\Textures\\
Maskonaive2_1024.dds", 500.0f);
mComponents.push_back(mSkybox);

```

This produces output similar to [Figure 16.2](#).



**Figure 16.2** Output of the skybox component, along with a textured model. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

## Summary

In this chapter, you created a material system to encapsulate and simplify the interactions with effects. You wrote a set of classes that wrapped the Effects 11 library types for effects, techniques, passes, and variables, and an extensible `Material` class to bring all these types together. You exercised this system with two demos. For the first demo, you refactored the model demo of the last chapter. For the second demo, you created a reusable skybox component.

In the next chapter, you develop a set of types for directional lights, point lights, and spot-lights. You employ the material system to support the lighting shaders you authored in [Chapter 6, “Lighting Models,”](#) and [Chapter 7, “Additional Lighting Models,”](#) and you create demo applications to render scenes with interactive lighting.

## Exercises

1. From within the debugger, walk through the code to initialize the `Effect`, `Technique`, `Pass`, `Variable`, and `Material` classes in the `MaterialDemo` project. This exercise should help you better understand the design and implementation of these systems.
2. Write a material class for the `TextureMapping.fx` effect. Then write a demo application to render the `sphere.obj` model mapped with a texture. *Note:* You can find many textures on the companion website.

# Chapter 17. Lights

In this chapter, you develop a set of types to support directional, point, and spotlights. This work rounds out the base of your C++ rendering framework and marks the end of [Part III](#), “[Rendering with DirectX](#).”

## Motivation

In [Chapter 6](#), “[Lighting Models](#),” and [Chapter 7](#), “[Additional Lighting Models](#),” you expended a lot of effort authoring shaders to simulate lighting. Specifically, you authored effects for the following topics:

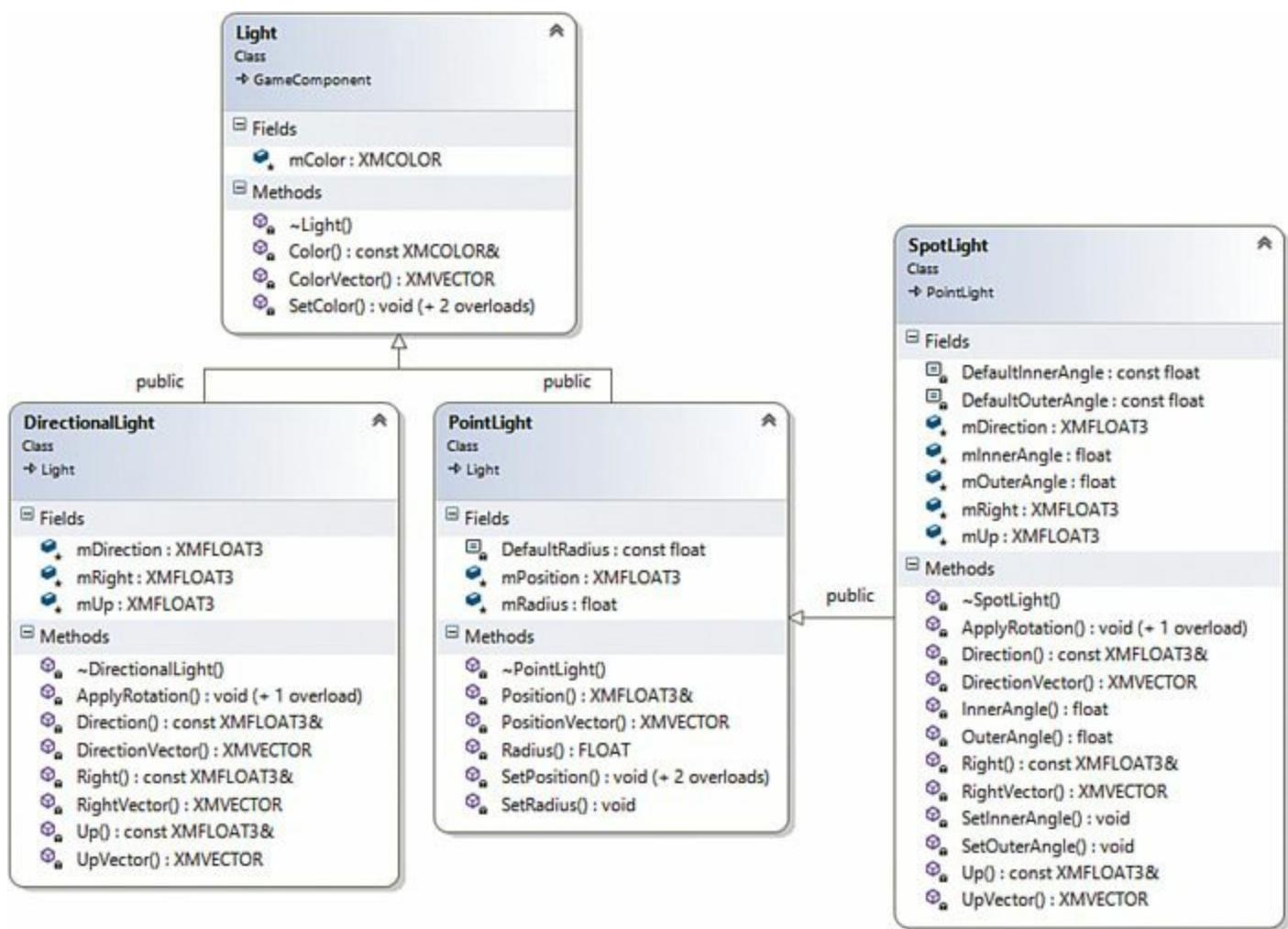
- Ambient lighting
- Diffuse lighting with directional lights
- Specular highlights
- Point lights
- Spotlights

You need a way to represent those lights within your C++ rendering framework. Furthermore, you need to create a set of materials to accommodate the interactions with these effects. That’s the work you do in this chapter. To exercise the code, you author a set of demos that include the capability to manipulate lights at runtime.

## Light Data Types

You’ve modeled three distinct “types” of light in your shaders: directional, point, and spotlights. They all have a color, so you can imagine a base `Light` class that contains a color member. Otherwise, directional and point lights don’t share any data. A directional light aims in a particular direction but has no position (and, thus, no attenuation). Conversely, a point light has a position and a radius of influence, but no specific direction. Thus, you can envision separate `PointLight` and `DirectionalLight` classes that both derive from the base `Light` class. Finally, a spotlight has elements of both directional and point lights; it has a position in space and a radius of influence, but it also casts light in a particular direction. You might envision a `SpotLight` class that inherits from both respective types. However, I tend to avoid multiple inheritance, so the `SpotLight` type derives only from the `PointLight` class and re-creates the directional light members.

[Figure 17.1](#) shows the class diagrams for the light data types.



**Figure 17.1** Class diagrams for the light data types.

Practically no complex implementation exists for any of these data types, so their source files are not listed here. You can find all these types on the companion website.

## A Diffuse Lighting Material

Refer back to the diffuse lighting shader you wrote in [Chapter 6](#). It includes shader constants for an ambient color, a light color, and a light direction, along with a color texture and world-view-projection and world matrices. Add that effect file (`DiffuseLighting.fx`) to one of your projects, and create a `DiffuseLightingMaterial` class and `DiffuseLightingMaterialVertex` structure with the declarations in [Listing 17.1](#).

### Listing 17.1 Declarations of Diffuse Lighting Material and Vertex Data Types

[Click here to view code image](#)

---

```

typedef struct _DiffuseLightingMaterialVertex
{
    XMFLOAT4 Position;
    XMFLOAT2 TextureCoordinates;
    XMFLOAT3 Normal;

    _DiffuseLightingMaterialVertex() { }
}

```

```

_DiffuseLightingMaterialVertex(XMFLOAT4 position, XMFLOAT2
textureCoordinates, XMFLOAT3 normal)
    : Position(position),
TextureCoordinates(textureCoordinates),
Normal(normal) { }
} DiffuseLightingMaterialVertex;

class DiffuseLightingMaterial : public Material
{
    RTTI_DECLARATIONS(DiffuseLightingMaterial, Material)

MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)
MATERIAL_VARIABLE_DECLARATION(World)
MATERIAL_VARIABLE_DECLARATION(AmbientColor)
MATERIAL_VARIABLE_DECLARATION(LightColor)
MATERIAL_VARIABLE_DECLARATION(LightDirection)
MATERIAL_VARIABLE_DECLARATION(ColorTexture)

public:
DiffuseLightingMaterial();

virtual void Initialize(Effect* effect) override;
virtual void CreateVertexBuffer(ID3D11Device* device, const
Mesh&
mesh, ID3D11Buffer** vertexBuffer) const override;
void CreateVertexBuffer(ID3D11Device* device,
DiffuseLightingMaterialVertex* vertices, UINT vertexCount,
ID3D11Buffer** vertexBuffer) const;
virtual UINT VertexSize() const override;
};

```

### Note

We haven't forgotten ambient lighting! But because it's included within the diffuse lighting effect, we have omitted a specific demonstration of how to incorporate this shader into the C++ framework. You can find an independent ambient lighting demo on the companion website.

Each vertex for the diffuse lighting effect consists of a position, a UV, and a normal; these elements are expressed by the `DiffuseLightingMaterialVertex` structure. The `DiffuseLightingMaterial` class declares members and accessors for each of the shader constants through the `MATERIAL_VARIABLE_DECLARATION` macro. Thus, the class listing is a little larger than the materials you wrote in the last chapter but follows the same pattern. The same is true for the class implementation. [Listing 17.2](#) presents the full listing for this implementation, to reinforce the syntax for the material system; however, this is the last material listing included in this

chapter. All the remaining materials follow these same patterns.

## Listing 17.2 The DiffuseLightingMaterial.cpp File

[Click here to view code image](#)

```
#include "DiffuseLightingMaterial.h"
#include "GameException.h"
#include "Mesh.h"

namespace Rendering
{
    RTTI_DEFINITIONS(DiffuseLightingMaterial)

    DiffuseLightingMaterial::DiffuseLightingMaterial()
        : Material("main1"),
          MATERIAL_VARIABLE_INITIALIZATION(WorldViewProjection),
          MATERIAL_VARIABLE_INITIALIZATION(World),
          MATERIAL_VARIABLE_INITIALIZATION(AmbientColor),
          MATERIAL_VARIABLE_INITIALIZATION(LightColor),
          MATERIAL_VARIABLE_INITIALIZATION(LightDirection),
          MATERIAL_VARIABLE_INITIALIZATION(ColorTexture)
    {
    }

    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
        WorldViewProjection)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial, World)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
        AmbientColor)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
        LightColor)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
        LightDirection)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
        ColorTexture)

    void DiffuseLightingMaterial::Initialize(Effect* effect)
    {
        Material::Initialize(effect);

        MATERIAL_VARIABLE_RETRIEVE(WorldViewProjection)
        MATERIAL_VARIABLE_RETRIEVE(World)
        MATERIAL_VARIABLE_RETRIEVE(AmbientColor)
        MATERIAL_VARIABLE_RETRIEVE(LightColor)
        MATERIAL_VARIABLE_RETRIEVE(LightDirection)
        MATERIAL_VARIABLE_RETRIEVE(ColorTexture)
```

```

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

        CreateInputLayout("main11", "p0",
inputElementDescriptions,
ARRAYSIZE(inputElementDescriptions));
}

void
DiffuseLightingMaterial::CreateVertexBuffer(ID3D11Device* device, const Mesh& mesh, ID3D11Buffer** vertexBuffer) const
{
    const std::vector<XMFLOAT3>& sourceVertices =
mesh.Vertices();
    std::vector<XMFLOAT3>* textureCoordinates = mesh.
TextureCoordinates().at(0);
    assert(textureCoordinates->size() ==
sourceVertices.size());
    const std::vector<XMFLOAT3>& normals = mesh.Normals();
    assert(textureCoordinates->size() ==
sourceVertices.size());

    std::vector<DiffuseLightingMaterialVertex> vertices;
    vertices.reserve(sourceVertices.size());
    for (UINT i = 0; i < sourceVertices.size(); i++)
    {
        XMFLOAT3 position = sourceVertices.at(i);
        XMFLOAT3 uv = textureCoordinates->at(i);
        XMFLOAT3 normal = normals.at(i);

vertices.push_back(DiffuseLightingMaterialVertex(XMFLOAT4(position.y, position.z, 1.0f), XMFLOAT2(uv.x, uv.y), normal));
    }

    CreateVertexBuffer(device, &vertices[0],
vertices.size(),
vertexBuffer);
}

```

```

    }

    void DiffuseLightingMaterial::CreateVertexBuffer(ID3D11Device* device, DiffuseLightingMaterialVertex* vertices, UINT vertexCount, ID3D11Buffer** vertexBuffer) const
    {
        D3D11_BUFFER_DESC vertexBufferDesc;
        ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
        vertexBufferDesc.ByteWidth = VertexSize() * vertexCount;
        vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
        vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

        D3D11_SUBRESOURCE_DATA vertexSubResourceData;
        ZeroMemory(&vertexSubResourceData, sizeof(vertexSubResourceData));
        vertexSubResourceData.pSysMem = vertices;
        if (FAILED(device->CreateBuffer(&vertexBufferDesc, &vertexSubResourceData, vertexBuffer)))
        {
            throw GameException("ID3D11Device::CreateBuffer() failed.");
        }
    }

    UINT DiffuseLightingMaterial::VertexSize() const
    {
        return sizeof(DiffuseLightingMaterialVertex);
    }
}

```

The `DiffuseLightingMaterial` constructor initializes each of the shader variables using the `MATERIAL_VARIABLE_INITIALIZATION` macro. Then their accessors are defined and the associated members are cached through the `MATERIAL_VARIABLE_DEFINITION` and `MATERIAL_VARIABLE_RETRIEVE` macros, respectively. Next, the `DiffuseLightingMaterial::Initialize()` method creates the input layout to match the vertex structure. Finally, note how vertex positions, texture coordinates, and normals are retrieved from a mesh within the `DiffuseLightingMaterial::CreateVertexBuffer()` method. These are the implementation details that vary from one material to another.

## A Diffuse Lighting Demo

Now let's exercise the diffuse lighting material with a demo. This application employs the newly created `DirectionalLight` class to feed data into the `LightColor()` and `LightDirection()` members of the `DiffuseMaterial` type. To make the demo a bit more interesting, you'll update the direction of the light (at runtime) using the arrow keys and change the

intensity of the ambient light with the Page Up and Page Down keys. Start by creating a DiffuseLightingDemo class with the declaration in [Listing 17.3](#).

### **Listing 17.3 Declaration of the DiffuseLightingDemo Class**

[Click here to view code image](#)

---

```
class DiffuseLightingDemo : public DrawableGameComponent
{
    RTTI_DECLARATIONS(DiffuseLightingDemo,
DrawableGameComponent)

public:
    DiffuseLightingDemo(Game& game, Camera& camera);
    ~DiffuseLightingDemo();

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    DiffuseLightingDemo();
    DiffuseLightingDemo(const DiffuseLightingDemo& rhs);
    DiffuseLightingDemo& operator=(const DiffuseLightingDemo&
rhs);

    void UpdateAmbientLight(const GameTime& gameTime);
    void UpdateDirectionalLight(const GameTime& gameTime);

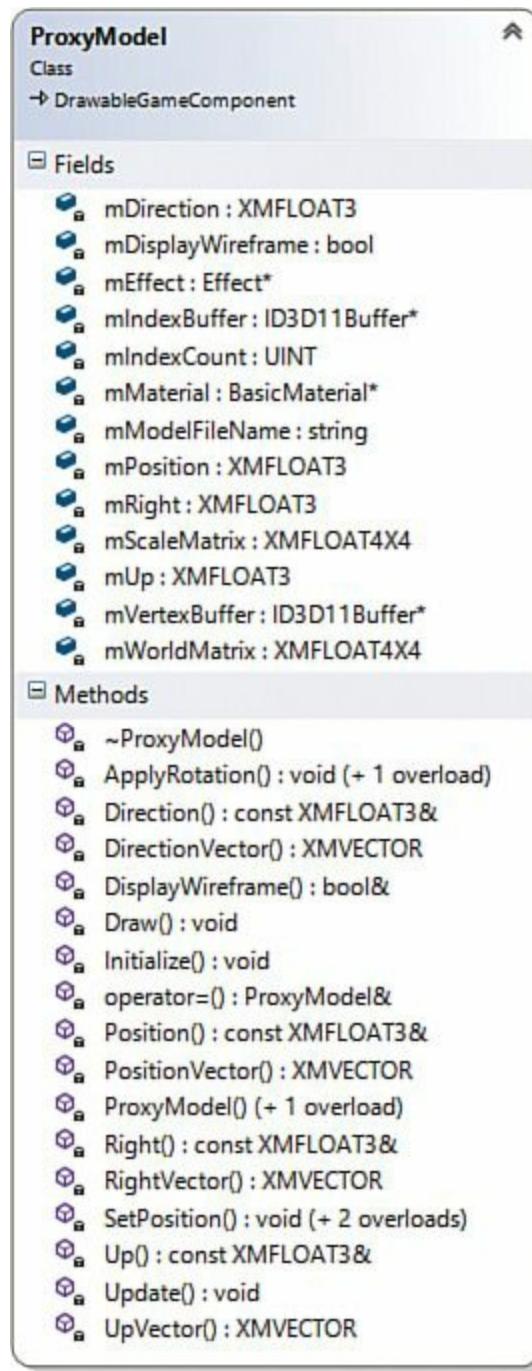
    static const float AmbientModulationRate;
    static const XMFLOAT2 RotationRate;

    Effect* mEffect;
    DiffuseLightingMaterial* mMaterial;
    ID3D11ShaderResourceView* mTextureShaderResourceView;
    ID3D11Buffer* mVertexBuffer;
    ID3D11Buffer* mIndexBuffer;
    UINT mIndexCount;

    XMCOLOR mAmbientColor;
    DirectionalLight* mDirectionalLight;
    Keyboard* mKeyboard;
    XMFLOAT4X4 mWorldMatrix;

    ProxyModel* mProxyModel;
};
```

You have the usual data members within the `DiffuseLightingDemo` class: storage for an effect and a material, a shader resource view for a texture, a world matrix for transforming the object within the scene, and vertex and index buffers. New are class members for storing the ambient color and directional light. Note that the ambient color is represented through the `XMCOLOR` type; this could have been stored as a generic `Light` object, but an `XMCOLOR` suffices. Also new is the `ProxyModel` data type, which renders a model representing a light. In an actual game, you wouldn't render a proxy model, but this is useful during development. Without a proxy model, the position and orientation of a light would be difficult to keep track of. A listing of the `ProxyModel` class is omitted for brevity; [Figure 17.2](#) shows its class diagram. You can find the source code for this class on the companion website.



**Figure 17.2** Class diagram for the `ProxyModel` class.

Rendering for the diffuse lighting demo follows the patterns established over the last few demos. You set the primitive topology and input layout, and you bind the vertex and index buffers to the input-assembler pipeline stage. Then you update the material and execute a draw call. [Listing 17.4](#) presents

the Draw() method of the DiffuseLightingDemo class.

#### Listing 17.4 Implementation of the DiffuseLightingDemo::Draw() Method

[Click here to view code image](#)

```
void DiffuseLightingDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mMaterial->CurrentTechnique()->Passes().at(0);
    ID3D11InputLayout* inputLayout = mMaterial->InputLayouts().
at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);

    UINT stride = mMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_FORMAT
R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    XMVECTOR ambientColor = XMLoadColor(&mAmbientColor);

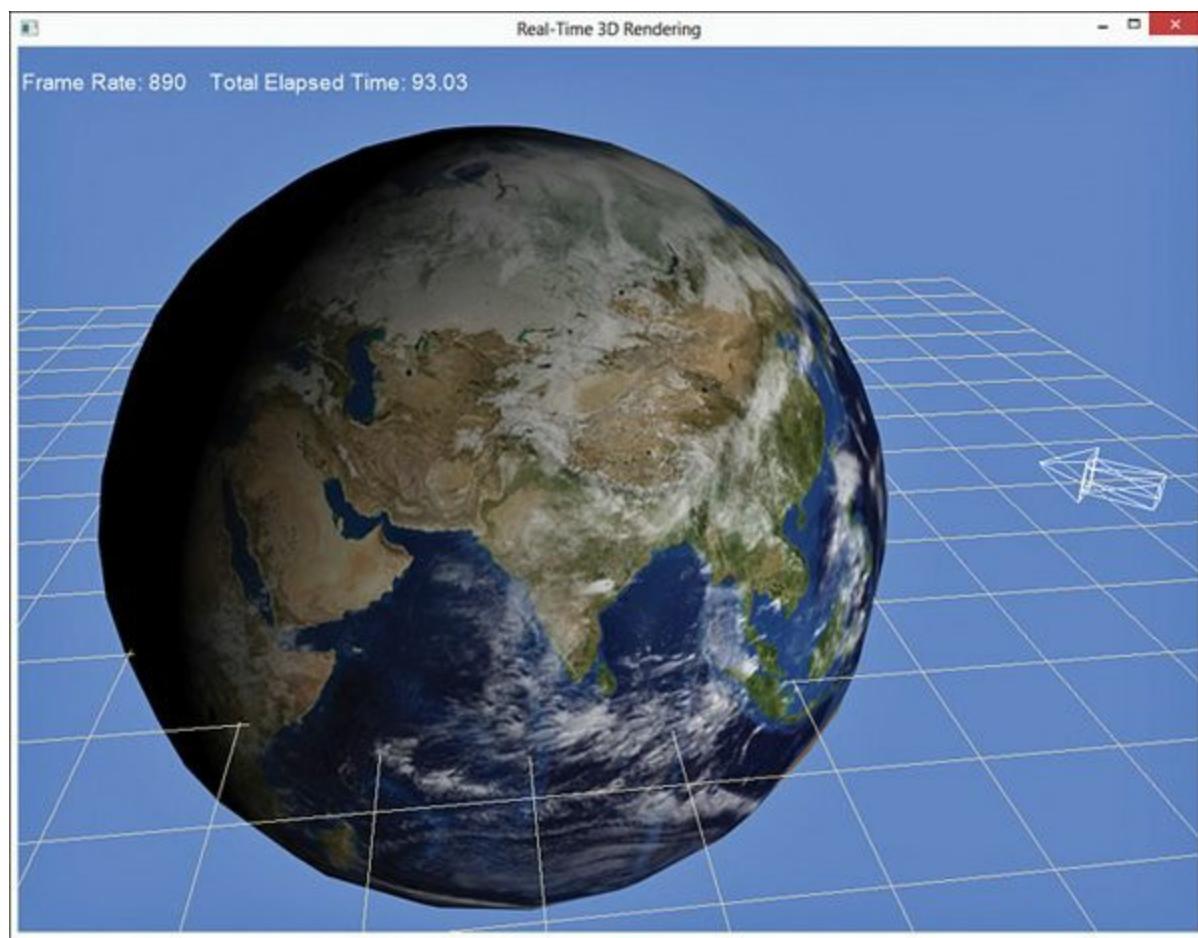
    mMaterial->WorldViewProjection() << wvp;
    mMaterial->World() << worldMatrix;
    mMaterial->AmbientColor() << ambientColor;
    mMaterial->LightColor() << mDirectionalLight->ColorVector();
    mMaterial->LightDirection() <<
mDirectionalLight->DirectionVector();
    mMaterial->ColorTexture() << mTextureShaderResourceView;

    pass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);

    mProxyModel->Draw(gameTime);
}
```

[Figure 17.3](#) shows the output of the diffuse lighting demo. Note the proxy model (the wireframe arrow) depicting the direction of the light. Also note the reference grid within the image. This useful component (the `Grid` class) can help provide a frame of reference for your demos. You can find the source code for the `Grid` class on the book's companion website.



**Figure 17.3** Output of the diffuse lighting demo. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

## Interacting with Lights

Allowing your lights to be updated at runtime is straight forward. For example, to update the ambient light, you merely need to modify the alpha channel of the

`DiffuseLightingDemo::mAmbientColor` member. You can do this in response to a keyboard press or as a function of time (to create, for example, a pulsing or flickering light). [Listing 17.5](#) presents a method for incrementing and decrementing the ambient light intensity when the Page Up and Page Down keys are pressed. This method should be invoked from within `DiffuseLightingDemo::Update()`.

### Listing 17.5 Incrementing and Decrementing Ambient Light Intensity

[Click here to view code image](#)

---

```
void DiffuseLightingDemo::UpdateAmbientLight(const GameTime& gameTime)
```

```

{
    static float ambientIntensity = mAmbientColor.a;

    if (mKeyboard != nullptr)
    {
        if (mKeyboard->IsKeyDown(DIK_PGUP) && ambientIntensity <
UCHAR_MAX)
        {
            ambientIntensity += AmbientModulationRate *
(float)gameTime.ElapsedGameTime();
            mAmbientColor.a = XMMin<UCHAR>(ambientIntensity,
UCHAR_MAX);
        }

        if (mKeyboard->IsKeyDown(DIK_PGDN) && ambientIntensity >
0)
        {
            ambientIntensity -= LightModulationRate *
(float)gameTime.
ElapsedGameTime();
            mAmbientColor.a = XMMax<UCHAR>(ambientIntensity, 0);
        }
    }
}

```

---

The ambientIntensity static variable tracks the current ambient intensity and is updated, over time, whenever the associated keys are pressed. The AmbientModulationRate value dictates how quickly the ambient intensity should increase or decrease. For example, with a LightModulationRate of 255, the ambient intensity goes from its minimum value (0) to its maximum value (255) in 1 second.

You can also interact with the directional light. [Listing 17.6](#) presents a method for rotating a directional light with the arrow keys.

## Listing 17.6 Rotating a Directional Light

[Click here to view code image](#)

---

```

void DiffuseLightingDemo::UpdateDirectionalLight(const GameTime&
gameTime)
{
    float elapsedTime = (float)gameTime.ElapsedGameTime();

    XMFLOAT2 rotationAmount = Vector2Helper::Zero;
    if (mKeyboard->IsKeyDown(DIK_LEFTARROW))
    {
        rotationAmount.x += LightRotationRate.x * elapsedTime;
    }
}

```

```

}

if (mKeyboard->IsKeyDown (DIK_RIGHTARROW) )
{
    rotationAmount.x -= LightRotationRate.x * elapsedTime;
}
if (mKeyboard->IsKeyDown (DIK_UPARROW) )
{
    rotationAmount.y += LightRotationRate.y * elapsedTime;
}
if (mKeyboard->IsKeyDown (DIK_DOWNARROW) )
{
    rotationAmount.y -= LightRotationRate.y * elapsedTime;
}

XMMATRIX lightRotationMatrix = XMMatrixIdentity();
if (rotationAmount.x != 0)
{
    lightRotationMatrix =
XMMatrixRotationY(rotationAmount.x);
}

if (rotationAmount.y != 0)
{
    XMMATRIX lightRotationAxisMatrix = XMMatrixRotationAxis
(mDirectionalLight->RightVector(), rotationAmount.y);
    lightRotationMatrix *= lightRotationAxisMatrix;
}

if (rotationAmount.x != 0.0f || rotationAmount.y != 0.0f)
{
    mDirectionalLight->ApplyRotation(lightRotationMatrix);
    mProxyModel->ApplyRotation(lightRotationMatrix);
}
}

```

In this code, the `LightRotationRate` vector dictates how quickly the light will rotate (using two values allows different rates for horizontal and vertical rotation). The calculated rotation amounts are used to build matrices for rotation around the y-axis and the directional light's *right* vector. The resulting rotation matrix (potentially consisting of both horizontal and vertical rotation) is used to transform the directional light and the proxy model.

## A Point Light Demo

A point light demo is similar to the diffuse lighting demo. Create a `PointLightMaterial` class that matches the `PointLight.fx` effect from [Chapter 7](#). This effect incorporates the Blinn-Phong specular highlighting model, and the material should contain the following variable declarations:

[Click here to view code image](#)

```

MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)
MATERIAL_VARIABLE_DECLARATION(World)
MATERIAL_VARIABLE_DECLARATION(SpecularColor)
MATERIAL_VARIABLE_DECLARATION(SpecularPower)
MATERIAL_VARIABLE_DECLARATION(AmbientColor)
MATERIAL_VARIABLE_DECLARATION(LightColor)
MATERIAL_VARIABLE_DECLARATION(LightPosition)
MATERIAL_VARIABLE_DECLARATION(LightRadius)
MATERIAL_VARIABLE_DECLARATION(CameraPosition)
MATERIAL_VARIABLE_DECLARATION(ColorTexture)

```

The point light vertex shader accepts the same input as the diffuse lighting shader (position, texture coordinates, and a normal) so you can either duplicate the input layout and vertex buffer creation code or roll a system to share code between the materials.

The PointLightDemo class can borrow heavily from the DiffuseLightingDemo type, replacing the DirectionalLight data member with a PointLight and adding members for the specular color and power. Rendering for the point light demo is identical to the diffuse lighting demo, except for the shader variable updates. You update a point light material with code such as the following:

[Click here to view code image](#)

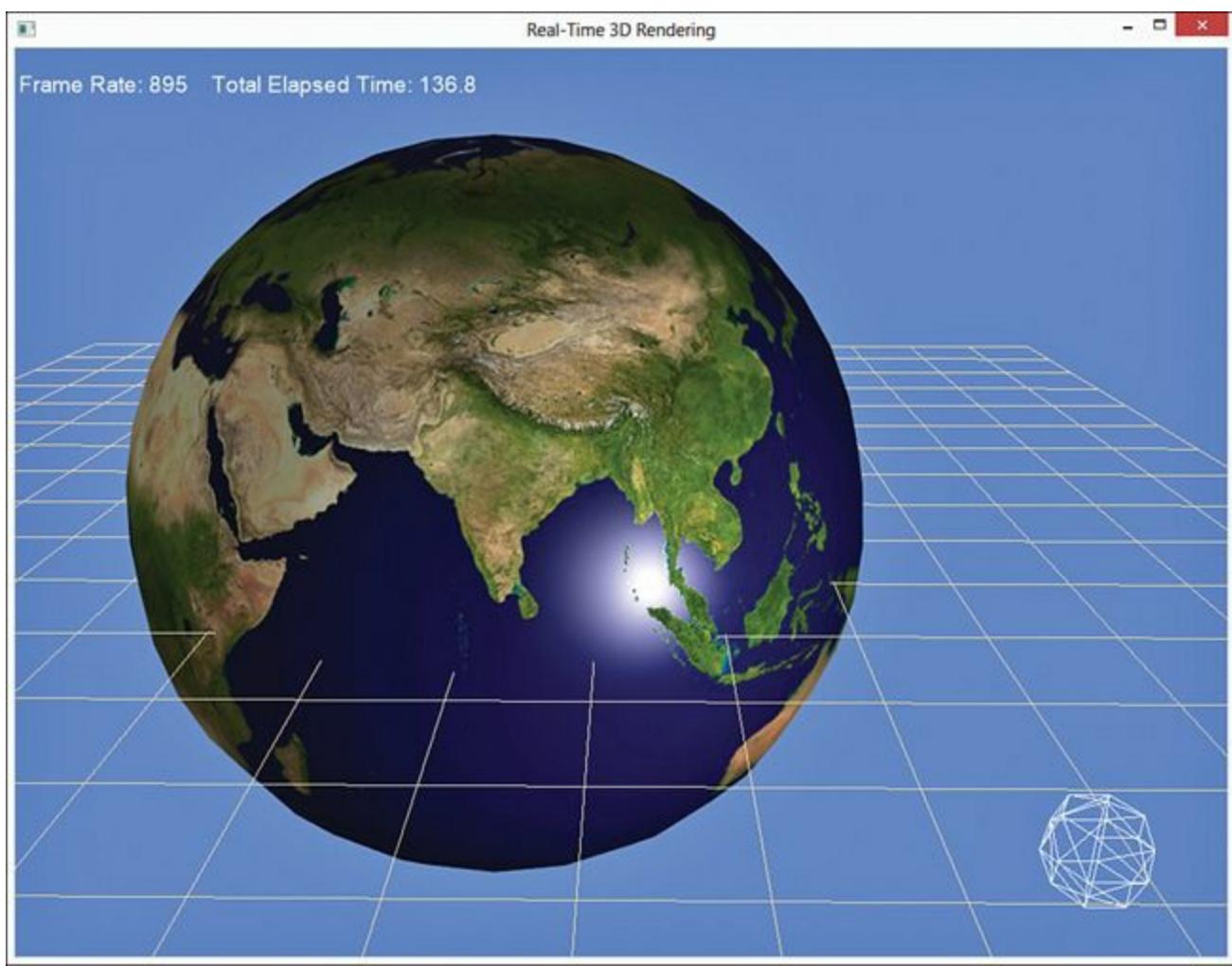
```

XMMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
XMMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
XMVECTOR ambientColor = XMLoadColor(&mAmbientColor);
XMVECTOR specularColor = XMLoadColor(&mSpecularColor);

mMaterial->WorldViewProjection() << wvp;
mMaterial->World() << worldMatrix;
mMaterial->SpecularColor() << specularColor;
mMaterial->SpecularPower() << mSpecularPower;
mMaterial->AmbientColor() << ambientColor;
mMaterial->LightColor() << mPointLight->ColorVector();
mMaterial->LightPosition() << mPointLight->PositionVector();
mMaterial->LightRadius() << mPointLight->Radius();
mMaterial->ColorTexture() << mTextureShaderResourceView;
mMaterial->CameraPosition() << mCamera->PositionVector();

```

Note the updates for the specular color and power, and light position and radius. [Figure 17.4](#) presents the output from the point light demo, with a proxy model representing the point light.



**Figure 17.4** Output of the point light demo. (*Texture from Reto Stöckli, NASA Earth Observatory.*)

Just as with the diffuse lighting demo, you can allow your point light to be manipulated at runtime.

[Listing 17.7](#) presents a method for moving the point light along the x-, y-, and z-axes in response to pressing number pad keys: 4/6 (x-axis), 3/9 (y-axis), and 8/2 (z-axis).

### Listing 17.7 Moving the Point Light

[Click here to view code image](#)

```
void PointLightDemo::UpdatePointLight(const GameTime& gameTime)
{
    XMFLOAT3 movementAmount = Vector3Helper::Zero;
    if (mKeyboard != nullptr)
    {
        if (mKeyboard->IsKeyDown(DIK_NUMPAD4))
        {
            movementAmount.x = -1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_NUMPAD6))
        {
            movementAmount.x = 1.0f;
        }
    }
}
```

```

if (mKeyboard->IsKeyDown(DIK_NUMPAD9))
{
    movementAmount.y = 1.0f;
}

if (mKeyboard->IsKeyDown(DIK_NUMPAD3))
{
    movementAmount.y = -1.0f;
}

if (mKeyboard->IsKeyDown(DIK_NUMPAD8))
{
    movementAmount.z = -1.0f;
}

if (mKeyboard->IsKeyDown(DIK_NUMPAD2))
{
    movementAmount.z = 1.0f;
}

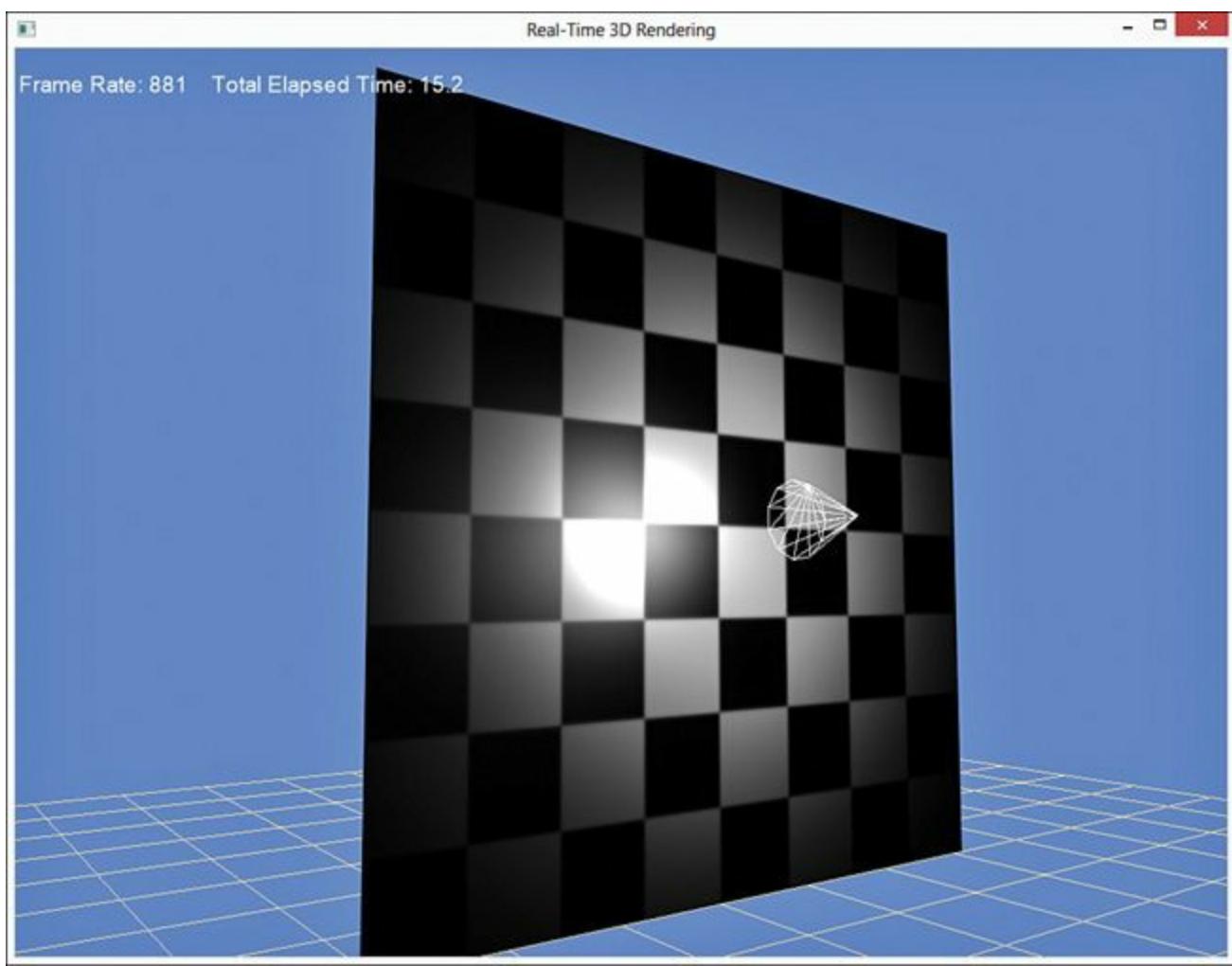
XMVECTOR movement = XMLoadFloat3(&movementAmount) *
MovementRate *
(float)gameTime.ElapsedGameTime();
    mPointLight->SetPosition(mPointLight->PositionVector() +
movement);
    mProxyModel->SetPosition(mPointLight->Position());
}

```

---

## A Spotlight Demo

A spotlight demo follows the same patterns. We've left it as an exercise for you to create the material that matches the `SpotLight.fx` effect (from [Chapter 7](#)) and the corresponding demo component. [Figure 17.5](#) shows the output of the demo available on the companion website.



**Figure 17.5** Output of the spotlight demo.

Interaction with the spotlight is a combination of the methods used for the directional and point light demos. The arrow keys control the direction of the spotlight, and the number pad updates its position.

## Summary

In this chapter, you developed a set of types to support directional, point, and spotlights. You created materials for interacting with the shaders you authored in [Chapters 6](#) and [7](#). Then you developed some demo applications to exercise these new data types. This work completes [Part III](#), “[Rendering with DirectX](#).” In the next section, you begin exploring more intermediate-level rendering topics.

## Exercises

1. Develop the spotlight demo discussed in the chapter.
2. Create materials for the environment mapping, transparency mapping, normal mapping, and displacement mapping shaders you authored in [Chapter 8](#), “[Gleaming the Cube](#),” and [Chapter 9](#), “[Normal Mapping and Displacement Mapping](#).” Develop corresponding demo application to exercise these materials.

# Part IV: Intermediate-Level Rendering Topics

[18 Post-Processing](#)

[19 Shadow Mapping](#)

[20 Skeletal Animation](#)

[21 Geometry and Tessellation Shaders](#)

[22 Additional Topics in Modern Rendering](#)

# Chapter 18. Post-Processing

Post-processing refers to a set of graphics techniques that are applied *after* the scene is rendered. For example, you might want to convert the entire scene into grayscale or make bright areas glow. In this chapter, you author some post-processing effects and integrate them with your C++ rendering framework.

## Render Targets

Thus far, all your demo applications have rendered directly to the back buffer, the 2D texture presented to the monitor when rendering is complete. But for post-processing applications, you render the scene to an intermediate texture and then apply the post-processing effect on that texture. The final image is rendered using a full-screen quadrilateral (two triangles that encompass the entire screen).

The following steps summarize the process:

1. Bind an off-screen render target to the output-merger stage.
2. Draw the scene.
3. Restore the back buffer as the render target bound to the output-merger stage.
4. Draw a full-screen quad using a post-processing effect that accepts the off-screen render target texture as input.

To facilitate this process, create a class named `FullScreenRenderTarget` whose declaration matches [Listing 18.1](#).

### Listing 18.1 Declaration of the FullScreenRenderTarget Class

[Click here to view code image](#)

---

```
class FullScreenRenderTarget
{
public:
    FullScreenRenderTarget(Game& game);
    ~FullScreenRenderTarget();

    ID3D11ShaderResourceView* OutputTexture() const;
    ID3D11RenderTargetView* RenderTargetView() const;
    ID3D11DepthStencilView* DepthStencilView() const;

    void Begin();
    void End();

private:
    FullScreenRenderTarget();
    FullScreenRenderTarget(const FullScreenRenderTarget& rhs);
    FullScreenRenderTarget& operator=(const
```

```

FullScreenRenderTarget&
rhs);

Game* mGame;
ID3D11RenderTargetView* mRenderTargetView;
ID3D11DepthStencilView* mDepthStencilView;
ID3D11ShaderResourceView* mOutputTexture;
};


```

---

The data members of this class should look familiar; your Game class contains identical members of type `ID3D11RenderTargetView` and `ID3D11DepthStencilView`. These are the types used for binding a render target and a depth-stencil buffer to the output-merger stage. Unlike the Game class, the `FullScreenRenderTarget` class also contains a `ID3D11ShaderResourceView` member that references the 2D texture underlying the render target. This *output texture* can be used as *input* into post-processing shaders.

The `FullScreenRenderTarget::Begin()` and `FullScreenRenderTarget::End()` methods bind the render target to the output-merger stage and restore the back buffer, respectively.

[Listing 18.2](#) presents the implementation of the `FullScreenRenderTarget` class.

## Listing 18.2 Implementation of the FullScreenRenderTarget Class

[Click here to view code image](#)

```

#include "FullScreenRenderTarget.h"
#include "Game.h"
#include "GameException.h"

namespace Library
{
    FullScreenRenderTarget::FullScreenRenderTarget(Game& game)
        : mGame(&game), mRenderTargetView(nullptr),
        mDepthStencilView(nullptr), mOutputTexture(nullptr)
    {
        D3D11_TEXTURE2D_DESC fullScreenTextureDesc;
        ZeroMemory(&fullScreenTextureDesc, sizeof
(fullScreenTextureDesc));
        fullScreenTextureDesc.Width = game.ScreenWidth();
        fullScreenTextureDesc.Height = game.ScreenHeight();
        fullScreenTextureDesc.MipLevels = 1;
        fullScreenTextureDesc.ArraySize = 1;
        fullScreenTextureDesc.Format =
DXGI_FORMAT_R8G8B8A8_UNORM;
        fullScreenTextureDesc.SampleDesc.Count = 1;
        fullScreenTextureDesc.SampleDesc.Quality = 0;
        fullScreenTextureDesc.BindFlags =
D3D11_BIND_RENDER_TARGET |

```

```
D3D11_BIND_SHADER_RESOURCE;
    fullScreenTextureDesc.Usage = D3D11_USAGE_DEFAULT;

    HRESULT hr;
    ID3D11Texture2D* fullScreenTexture = nullptr;
    if (FAILED(hr = game.Direct3DDevice()->CreateTexture2D
(&fullScreenTextureDesc, nullptr, &fullScreenTexture)))
    {
        throw GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
    }

    if (FAILED(hr = game.Direct3DDevice()-
>CreateShaderResourceView
(fullScreenTexture, nullptr, &mOutputTexture)))
    {
        ReleaseObject(fullScreenTexture);
        throw
GameException("IDXGIDevice::CreateShaderResource
View() failed.", hr);
    }

    if (FAILED(hr = game.Direct3DDevice()-
>CreateRenderTargetView
(fullScreenTexture, nullptr, &mRenderTargetView)))
    {
        ReleaseObject(fullScreenTexture);
        throw
GameException("IDXGIDevice::CreateRenderTargetView()
failed.", hr);
    }

    ReleaseObject(fullScreenTexture);

D3D11_TEXTURE2D_DESC depthStencilDesc;
ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));
depthStencilDesc.Width = game.ScreenWidth();
depthStencilDesc.Height = game.ScreenHeight();
depthStencilDesc.MipLevels = 1;
depthStencilDesc.ArraySize = 1;
depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthStencilDesc.SampleDesc.Count = 1;
depthStencilDesc.SampleDesc.Quality = 0;
depthStencilDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;

ID3D11Texture2D* depthStencilBuffer = nullptr;
```

```
    if (FAILED(hr = game.Direct3DDevice()->CreateTexture2D
(&depthStencilDesc, nullptr, &depthStencilBuffer)))
    {
        throw GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
    }

    if (FAILED(hr = game.Direct3DDevice()-
>CreateDepthStencilView
(depthStencilBuffer, nullptr, &mDepthStencilView)))
    {
        ReleaseObject(depthStencilBuffer);
        throw
GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
    }

    ReleaseObject(depthStencilBuffer);
}

FullScreenRenderTarget::~FullScreenRenderTarget()
{
    ReleaseObject(mOutputTexture);
    ReleaseObject(mDepthStencilView);
    ReleaseObject(mRenderTargetView);
}

ID3D11ShaderResourceView*
FullScreenRenderTarget::OutputTexture()
const
{
    return mOutputTexture;
}

ID3D11RenderTargetView*
FullScreenRenderTarget::RenderTargetView()
const
{
    return mRenderTargetView;
}

ID3D11DepthStencilView*
FullScreenRenderTarget::DepthStencilView()
const
{
    return mDepthStencilView;
}
```

```

void FullScreenRenderTarget::Begin()
{
    mGame->Direct3DDeviceContext()->OMSetRenderTargets(1,
&mRenderTargetView, mDepthStencilView);
}

void FullScreenRenderTarget::End()
{
    mGame->ResetRenderTargets();
}

```

---

The bulk of this implementation resides within the `FullScreenRenderTarget` constructor. First, you populate a `D3D11_TEXTURE2D_DESC` structure and use it to create the render target's underlying texture. These steps weren't explicit when you initialized the `Game` class because the back buffer was created when you constructed the swap chain. Note the two bind flags specified for the texture: `D3D11_BIND_RENDER_TARGET` and `D3D_BIND_SHADER_RESOURCE`. These flags indicate that the texture can be used both as a render target and as input into a shader.

After the texture is created, it's used to build shader resource and render target views. These views are exposed through public class accessors; after the views are instantiated, the texture reference can be released. Similar steps are used to initialize the depth-stencil view.

The `FullScreenRenderTarget::Begin()` method invokes the `ID3D11DeviceContext::OMSetRenderTargets()` method to bind the render target view and depth-stencil view to the output-merger stage. The `FullScreenRenderTarget::End()` method invokes `Game::ResetRenderTargets()`, which likewise invokes `OMSetRenderTargets()` but does so using its own `Game` class render target and depth-stencil views.

You use the `FullScreenRenderTarget` class with the following pattern:

[Click here to view code image](#)

```

mRenderTarget->Begin();
// 1. Clear mRenderTarget->RenderTargetView()
// 2. Clear mRenderTarget->DepthStencilView()
// 3. Draw Objects
mRenderTarget->End();

```

This pattern renders objects to the 2D texture underlying the `FullScreenRenderTarget` instance, and you can access this texture with the `FullScreenRenderTarget::OutputTexture()` method. After the `FullScreenRenderTarget::End()` method is invoked, anything you render is written to the back buffer.

## A Full-Screen Quad Component

Now that you can render to an off-screen texture, you need a way to apply an effect to the texture and

present its contents to the screen. Such a system needs to encapsulate the rendering code that would otherwise be redundant between applications, but it must also be flexible enough to support myriad post-processing effects. [Listing 18.3](#) presents the declaration of the `FullScreenQuad` class.

### **Listing 18.3 Declaration of the `FullScreenQuad` Class**

[Click here to view code image](#)

---

```
class FullScreenQuad : public DrawableGameComponent
{
    RTTI_DECLARATIONS(FullScreenQuad, DrawableGameComponent)

public:
    FullScreenQuad(Game& game);
    FullScreenQuad(Game& game, Material& material);
    ~FullScreenQuad();

    Material* GetMaterial();
    void SetMaterial(Material& material, const std::string&
techniqueName, const std::string& passName);
    void SetActiveTechnique(const std::string& techniqueName,
const
std::string& passName);
    void SetCustomUpdateMaterial(std::function<void()>
callback);

    virtual void Initialize() override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    FullScreenQuad();
    FullScreenQuad(const FullScreenQuad& rhs);
    FullScreenQuad& operator=(const FullScreenQuad& rhs);

    Material* mMaterial;
    Pass* mPass;
    ID3D11InputLayout* mInputLayout;
    ID3D11Buffer* mVertexBuffer;
    ID3D11Buffer* mIndexBuffer;
    UINT mIndexCount;
    std::function<void()> mCustomUpdateMaterial;
};
```

---

The `FullScreenQuad::mMaterial` data member references the material used to draw the quad, and it can be assigned through the `SetMaterial()` method. In this way, the same `FullScreenQuad` instance can be used with different materials throughout its lifetime. The `mPass` and `mInputLayout` members cache the corresponding data for use in the `Draw()` method.

The familiar vertex and index buffers store the vertices and indices for the quad. If you haven't seen the `std::function<T>` type, this is a general-purpose function wrapper that you can use to store and invoke functions, bind expressions, and lambda expressions (callbacks and closures). It's used here to allow a material to be updated by the calling context. This is necessary because the `FullScreenQuad` class doesn't know what material is being used to render the quad (and hence what shader inputs the material uses). All the `FullScreenQuad` class knows how to do is render a quad; it leaves the material updating to someone else. [Listing 18.4](#) presents the implementation of the `FullScreenQuad` class.

## Listing 18.4 Implementation of the FullScreenQuad Class

[Click here to view code image](#)

```
#include "FullScreenQuad.h"
#include "Game.h"
#include "GameException.h"
#include "Material.h"
#include "VertexDeclarations.h"

namespace Library
{
    RTTI_DEFINITIONS(FullScreenQuad)

    FullScreenQuad::FullScreenQuad(Game& game)
        : DrawableGameComponent(game),
          mMaterial(nullptr), mPass(nullptr),
          mInputLayout(nullptr),
          mVertexBuffer(nullptr), mIndexBuffer(nullptr),
          mIndexCount(0), mCustomUpdateMaterial(nullptr)
    {
    }

    FullScreenQuad::FullScreenQuad(Game& game, Material&
material)
        : DrawableGameComponent(game),
          mMaterial(&material), mPass(nullptr),
          mInputLayout(nullptr),
          mVertexBuffer(nullptr), mIndexBuffer(nullptr),
          mIndexCount(0), mCustomUpdateMaterial(nullptr)
    {
    }

    FullScreenQuad::~FullScreenQuad()
    {
        ReleaseObject(mIndexBuffer);
        ReleaseObject(mVertexBuffer);
```

```

}

Material* FullScreenQuad::GetMaterial()
{
    return mMateral;
}

void FullScreenQuad::SetMaterial(Material& material, const std::string& techniqueName, const std::string& passName)
{
    mMateral = &material;
    SetActiveTechnique(techniqueName, passName);
}

void FullScreenQuad::SetActiveTechnique(const std::string& techniqueName, const std::string& passName)
{
    Technique* technique = mMateral->GetEffect()
->TechniquesByName().at(techniqueName);
    assert(technique != nullptr);

    mPass = technique->PassesByName().at(passName);
    assert(mPass != nullptr);
    mInputLayout = mMateral->InputLayouts().at(mPass);
}

void FullScreenQuad::SetCustomUpdateMaterial(std::function<void()> callback)
{
    mCustomUpdateMaterial = callback;
}

void FullScreenQuad::Initialize()
{
    PositionTextureVertex vertices[] =
    {
        PositionTextureVertex(XMFLOAT4(-1.0f, -1.0f, 0.0f,
1.0f),
XMFLOAT2(0.0f, 1.0f)),
        PositionTextureVertex(XMFLOAT4(-1.0f, 1.0f, 0.0f,
1.0f),
XMFLOAT2(0.0f, 0.0f)),
        PositionTextureVertex(XMFLOAT4(1.0f, 1.0f, 0.0f,
1.0f),
XMFLOAT2(1.0f, 0.0f)),
        PositionTextureVertex(XMFLOAT4(1.0f, -1.0f, 0.0f,
1.0f),
XMFLOAT2(1.0f, 0.0f))
    };
}

```

```
1.0f),
XMFLOAT2(1.0f, 1.0f)),
};

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth =
sizeof(PositionTextureVertex) *
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof
(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()->CreateBuffer
(&vertexBufferDesc, &vertexSubResourceData, &mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer()
failed.");
}

UINT indices[] =
{
    0, 1, 2,
    0, 2, 3
};

mIndexCount = ARRAYSIZE(indices);

D3D11_BUFFER_DESC indexBufferDesc;
ZeroMemory(&indexBufferDesc, sizeof(indexBufferDesc));
indexBufferDesc.ByteWidth = sizeof(UINT) * mIndexCount;
indexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

D3D11_SUBRESOURCE_DATA indexSubResourceData;
ZeroMemory(&indexSubResourceData, sizeof
(indexSubResourceData));
indexSubResourceData.pSysMem = indices;
if (FAILED(mGame->Direct3DDevice()-
>CreateBuffer(&indexBufferDesc, &indexSubResourceData,
&mIndexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer()
failed.");
}
```

```

    }

}

void FullScreenQuad::Draw(const GameTime& gameTime)
{
    assert(mPass != nullptr);
    assert(mInputLayout != nullptr);

    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(PositionTextureVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_
FORMAT_R32_UINT, 0);

    if (mCustomUpdateMaterial != nullptr)
    {
        mCustomUpdateMaterial();
    }

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}
}

```

---

First, examine the `FullScreenQuad::Initialize()` method. This is code you've seen before, but notice the locations of the four vertices. These positions are in screen space where  $(-1, -1)$  is the lower-left corner of the screen and  $(1, 1)$  is the upper right. Because these positions are already in screen space, the vertex shader does not need to transform them.

Next, inspect the `FullScreen::Draw()` method. There's nothing fancy here, except for the invocation of the `mCustomUpdateMaterial()` function wrapper. The `std::function<T>` class is a **function object** (or **functor**) and, therefore, exposes `operator()`. Because of this, it looks like we're calling a function named `mCustomUpdateMaterial`; in reality, we're invoking `std::function<T>::operator()`, which is actually performing our callback or lambda expression.

Finally, observe that the quad's vertices are described by a position and texture coordinates. This

limits the materials that you can use with this class, but its application is still quite broad. With a bit of effort, you could extend this class to dynamically construct the vertex buffer from the specified material.

## Color Filtering

It's time to exercise the `FullScreenRenderTarget` and `FullScreenQuad` classes, and the first demonstration is a **color filter**. A color filter just modifies the output color in some way. If you've ever worn tinted glasses, you have experienced color filtering first hand. In this demo, you author several color filter shaders, including grayscale, inverse, sepia, and a generic color filtering system that can produce any number of color effects.

### A Grayscale Filter

Let's begin by creating a `ColorFilter.fx` file matching the contents in [Listing 18.5](#).

#### Listing 18.5 A Grayscale Shader

[Click here to view code image](#)

```
static const float3 GrayScaleIntensity = { 0.299f, 0.587f,
0.114f };

/********************* Resources *****/
Texture2D ColorTexture;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/********************* Data Structures *****/
struct VS_INPUT
{
    float4 Position : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};
```

```
***** Vertex Shader *****
```

```
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
    OUT.Position = IN.Position;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}
```

```
***** Pixel Shader *****
```

```
float4 grayscale_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float intensity = dot(color.rgb, GrayScaleIntensity);

    return float4(intensity.rrr, color.a);
}
```

```
***** Techniques *****
```

```
technique11 grayscale_filter
```

```
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
grayscale_pixel_shader()));
    }
}
```

---

This is a grayscale filter; it converts the sampled RGB color to grayscale by finding the intensity of the color. A naïve approach to determining a color's intensity is to average the three channels:

$$\text{Intensity} = (R + G + B) / 3$$

However, the human eye sees red, green, and blue differently. Specifically, our eyes are more sensitive to green, then red, and finally blue. Therefore, a better expression of grayscale intensity is through a weighted average of the three color channels, using the following equation:

$$\text{Intensity} = 0.299 * R + 0.587 * G + 0.114 * B$$

Your grayscale pixel shader can calculate this through a single dot product.

Before moving on, notice that the vertex shader passes the vertex position without transforming it. This is because the positions are already specified in screen space.

## A Color Filter Demo

Now you create a fresh Game-derived class that renders one of the demos you've developed over the last several chapters. For example, a class titled `ColorFilteringGame` that initializes a mouse and keyboard, a camera, a skybox, a reference grid, a frame-rate component, and the point light demo component. It also includes `SpriteBatch` and `SpriteFont` members for rendering some text to the screen. To this list, you add the following class members:

[Click here to view code image](#)

```
FullScreenRenderTarget* mRenderTarget;
FullScreenQuad* mFullScreenQuad;
Effect* mColorFilterEffect;
ColorFilterMaterial* mColorFilterMaterial;
```

This list includes members for your new `FullScreenRenderTarget` and `FullScreenQuad` classes. `mColorFilterEffect` stores the compiled color filter shader and is used to initialize the `mColorFilterMaterial` member. The `ColorFilterMaterial` class matches the inputs of the `ColorFilter.fx` shader.

You initialize these members in the `ColorFilteringGame::Initialize()` method with code such as the following:

[Click here to view code image](#)

```
mRenderTarget = new FullScreenRenderTarget(*this);

SetCurrentDirectory(Utility::ExecutableDirectory().c_str());
mColorFilterEffect = new Effect(*this);
mColorFilterEffect-
>LoadCompiledEffect(L"Content\\Effects\\ColorFilter.
cso");

mColorFilterMaterial = new ColorFilterMaterial();
mColorFilterMaterial->Initialize(*mColorFilterEffect);

mFullScreenQuad = new FullScreenQuad(*this,
*mColorFilterMaterial);
mFullScreenQuad->Initialize();
mFullScreenQuad->SetActiveTechnique("grayscale_filter", "p0");
mFullScreenQuad-
>SetCustomUpdateMaterial(std::bind(&ColorFilteringGame::
UpdateColorFilterMaterial, this));
```

The full-screen quad is initialized with the color filtering material and pass `p0` from the `grayscale_filter` technique. The quad's custom material callback is set to the `ColorFilteringGame::UpdateColorFilterMaterial()` method, which has the

following implementation:

[Click here to view code image](#)

```
void ColorFilteringGame::UpdateColorFilterMaterial()
{
    mColorFilterMaterial->ColorTexture()
<< mRenderTarget->OutputTexture();
}
```

This method is invoked when the full-screen quad is drawn, and it passes the render target's output texture to the `ColorTexture` shader variable. The final piece is the `ColorFilteringGame::Draw()` method, whose implementation is listed next:

[Click here to view code image](#)

```
void ColorFilteringGame::Draw(const GameTime &gameTime)
{
    // Render the scene to an off-screen texture.
    mRenderTarget->Begin();

    mDirect3DDeviceContext->ClearRenderTargetView(mRenderTarget
->RenderTargetView() , reinterpret_cast<const float*>
(&BackgroundColor));
    mDirect3DDeviceContext->ClearDepthStencilView(mRenderTarget
->DepthStencilView() , D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL,
1.0f, 0);

    Game::Draw(gameTime);

    mRenderTarget->End();

    // Render a full-screen quad with a post processing effect.
    mDirect3DDeviceContext-
>ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext-
>ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    mFullScreenQuad->Draw(gameTime);

    HRESULT hr = mSwapChain->Present(0, 0);
    if (FAILED(hr))
    {
        throw GameException("IDXGISwapChain::Present() failed.",
hr);
    }
}
```

The Draw() method follows the post-processing steps discussed at the beginning of the chapter. First, it binds the off-screen render target to the output-merger stage with a call to mRenderTarget->Begin(). Then it draws the scene, starting by clearing the off-screen render target and depth-stencil views. Next, it restores the back buffer as the target bound to the output-merger stage by calling mRenderTarget->End(). Finally, it renders the full-screen quad to the back buffer. Because the full-screen quad has been configured to use the ColorFilter.fx shader with the grayscale\_filter technique, it applies this effect to the rendered quad.

[Figure 18.1](#) shows the output of the grayscale filter applied to a scene with a point light, a reference grid, and a skybox. Note that this demo is fully interactive, just as the original point light demo was. You can relocate the point light in the scene with the number pad keys. The post-processing effect is independent of what was rendered to the off-screen render target.



**Figure 18.1** Output of the grayscale post-processing effect. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

## A Color Inverse Filter

You can apply additional color filters with little modification to the code base. For example, you can include a color inverse filter with the following pixel shader and technique:

[Click here to view code image](#)

```
float4 inverse_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
```

```

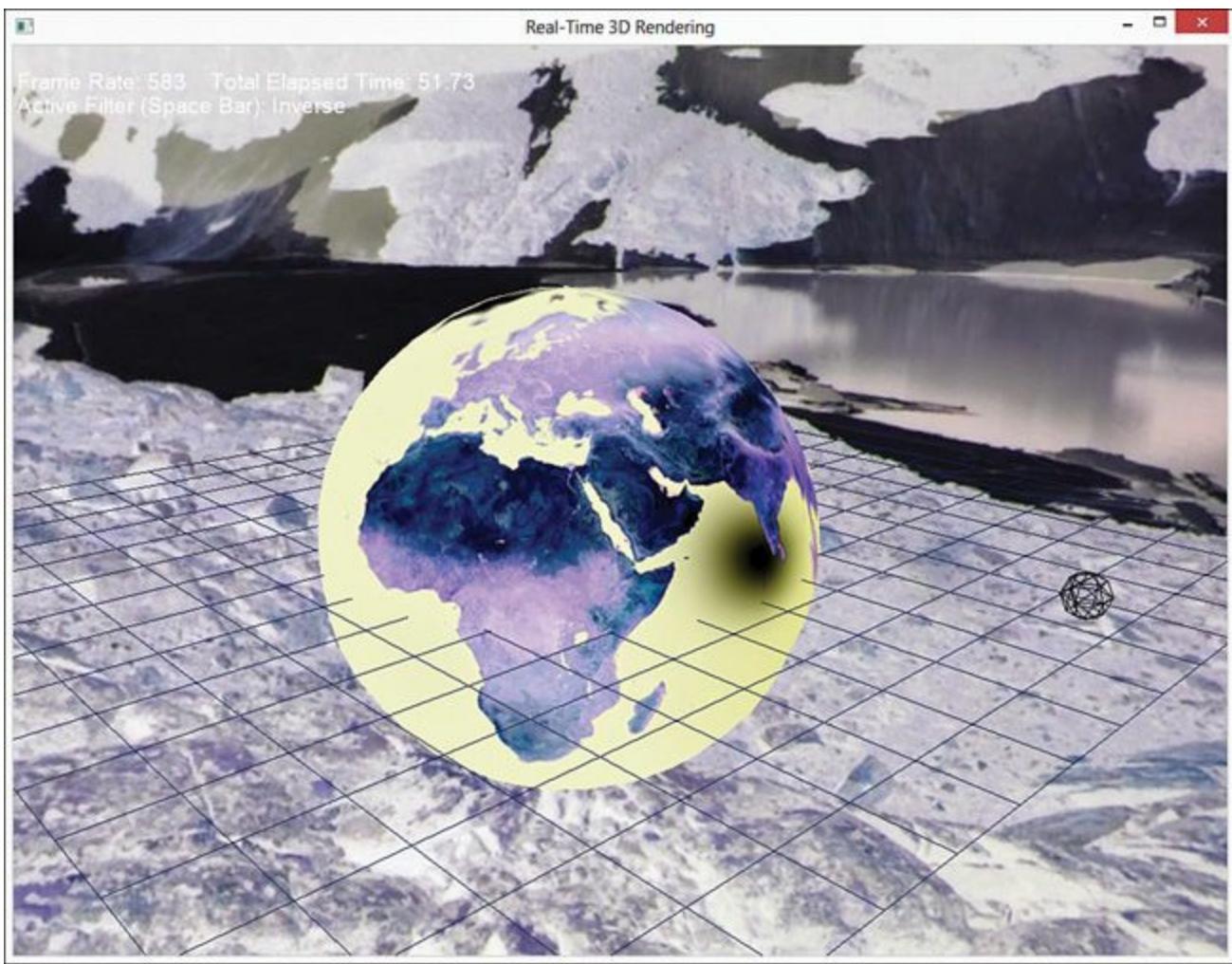
IN.TextureCoordinate);

    return float4(1 - color.rgb, color.a);
}

technique11 inverse_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
inverse_pixel_shader()));
    }
}

```

This just inverts each color to yield results similar to [Figure 18.2](#).



**Figure 18.2** Output of the color inverse post-processing effect. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

## A Sepia Filter

Now let's create a filter to resemble antique photographs. Back then, black-and-white photographs were actually shades of reddish-brown, called sepia tones. You can create a sepia shader using the

same concepts as the grayscale effect, determining the intensity of a pixel through weighted calculations. This time, though, you compute a different intensity for each color channel. The equations are:

$$Sepia_R = 0.393 * R + 0.769 * G + 0.189 * B$$

$$Sepia_G = 0.349 * R + 0.686 * G + 0.168 * B$$

$$Sepia_B = 0.272 * R + 0.534 * G + 0.131 * B$$

You could implement this through three separate dot product operations. But a cleaner alternative is to construct a  $3 \times 3$  matrix for the sepia coefficients and perform a single matrix multiplication (which is just a set of dot product operations). [Listing 18.6](#) presents the pixel shader and technique for the sepia post-processing effect. You can add this code to the `ColorFilter.fx` file instead of creating a completely separate effect.

## **Listing 18.6** A Sepia Shader

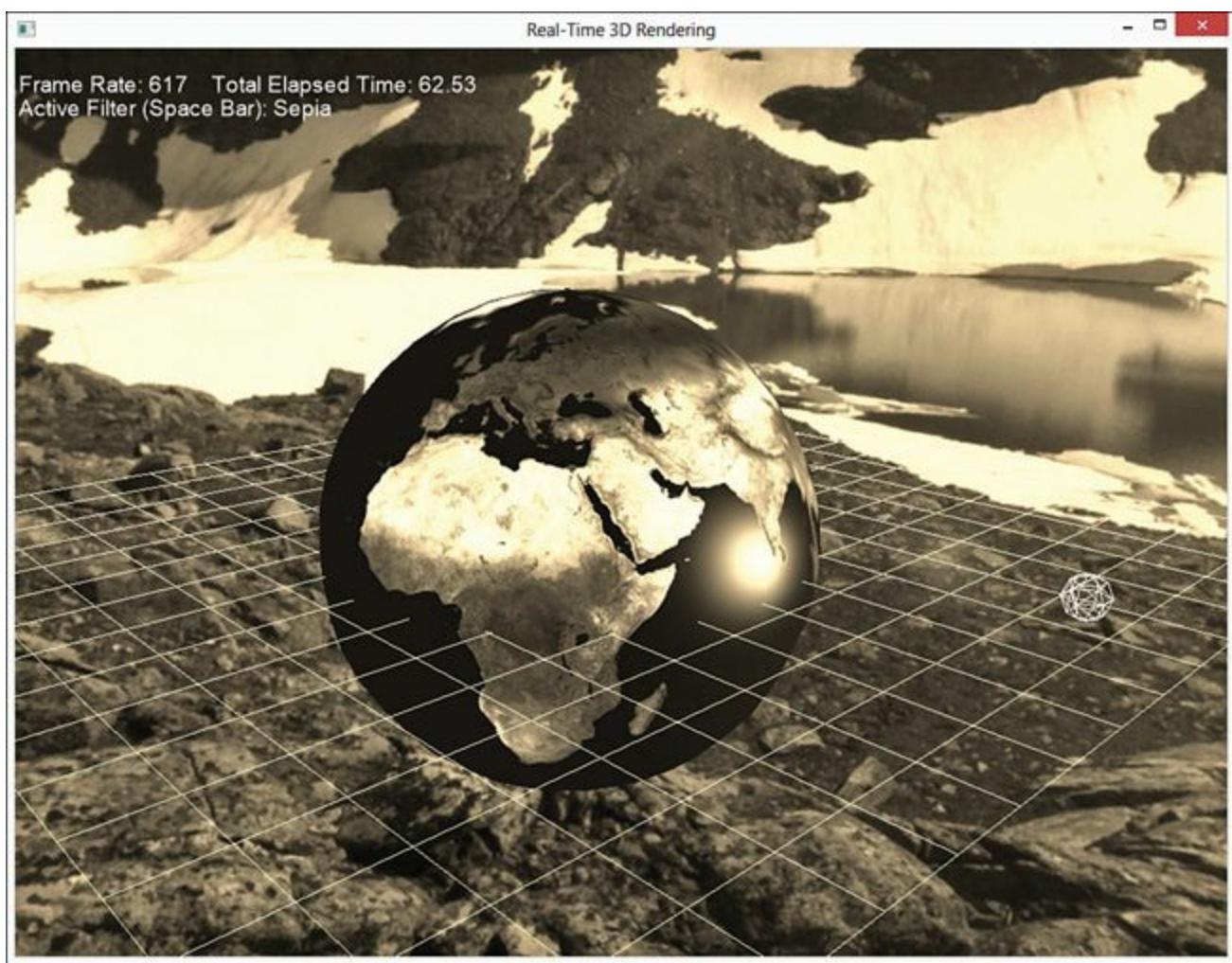
[Click here to view code image](#)

```
static const float3x3 SepiaFilter = { 0.393f, 0.349f, 0.272f,
                                      0.769f, 0.686f, 0.534f,
                                      0.189f, 0.168f, 0.131f };

float4 sepia_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return float4(mul(color.rgb, SepiaFilter), color.a);
}
technique11 sepia_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
sepia_pixel_shader()));
    }
}
```

The shader produces output similar to [Figure 18.3](#).



**Figure 18.3** Output of the sepia post-processing effect. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

## A Generic Color Filter

You can extend the implementation of the sepia shader to allow the color filter matrix to be specified at runtime. In this way, you could express all the aforementioned color filters through a single technique. To be as general-purpose as possible, the generic-filter uses a  $4 \times 4$  color filter matrix. [Listing 18.7](#) presents the listing of the generic filter pixel shader and associated technique.

### **Listing 18.7** A Generic Color Filter Shader

[Click here to view code image](#)

```
cbuffer CBufferPerObject
{
    float4x4 ColorFilter = { 1, 0, 0, 0,
                             0, 1, 0, 0,
                             0, 0, 1, 0,
                             0, 0, 0, 1 };

    float4 genericfilter_pixel_shader(VS_OUTPUT IN) : SV_Target
{
```

```

    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return float4(mul(color, ColorFilter).rgb, color.a);
}

technique11 generic_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
genericfilter_pixel_shader()));
    }
}

```

---

Using the generic filter shader, you could express the grayscale, inverse, and sepia effects as follows:

$$GrayScale = \begin{bmatrix} 0.299 & 0.299 & 0.299 & 0 \\ 0.587 & 0.587 & 0.587 & 0 \\ 0.144 & 0.144 & 0.144 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Inverse = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$Sepia = \begin{bmatrix} 0.393 & 0.349 & 0.272 & 0 \\ 0.769 & 0.686 & 0.534 & 0 \\ 0.189 & 0.168 & 0.131 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The `ColorFilteringGame` class, found on the companion website, demonstrates the generic color filter using a uniform scaling matrix to simulate brightness (values between 0 and 1 along the diagonal of the color filter matrix). You can increase or decrease the brightness by pressing the comma and period keys, respectively. Moreover, the demo enables you to toggle between the color filters by pressing the spacebar. Note that your

`ColorFilteringGame::UpdateColorFilterMaterial()` callback must be updated to pass in the `ColorFilter` shader variable. For example:

[Click here to view code image](#)

```

void ColorFilteringGame::UpdateColorFilterMaterial()
{
    XMATRIX colorFilter = XMLoadFloat4x4(&mGenericColorFilter);

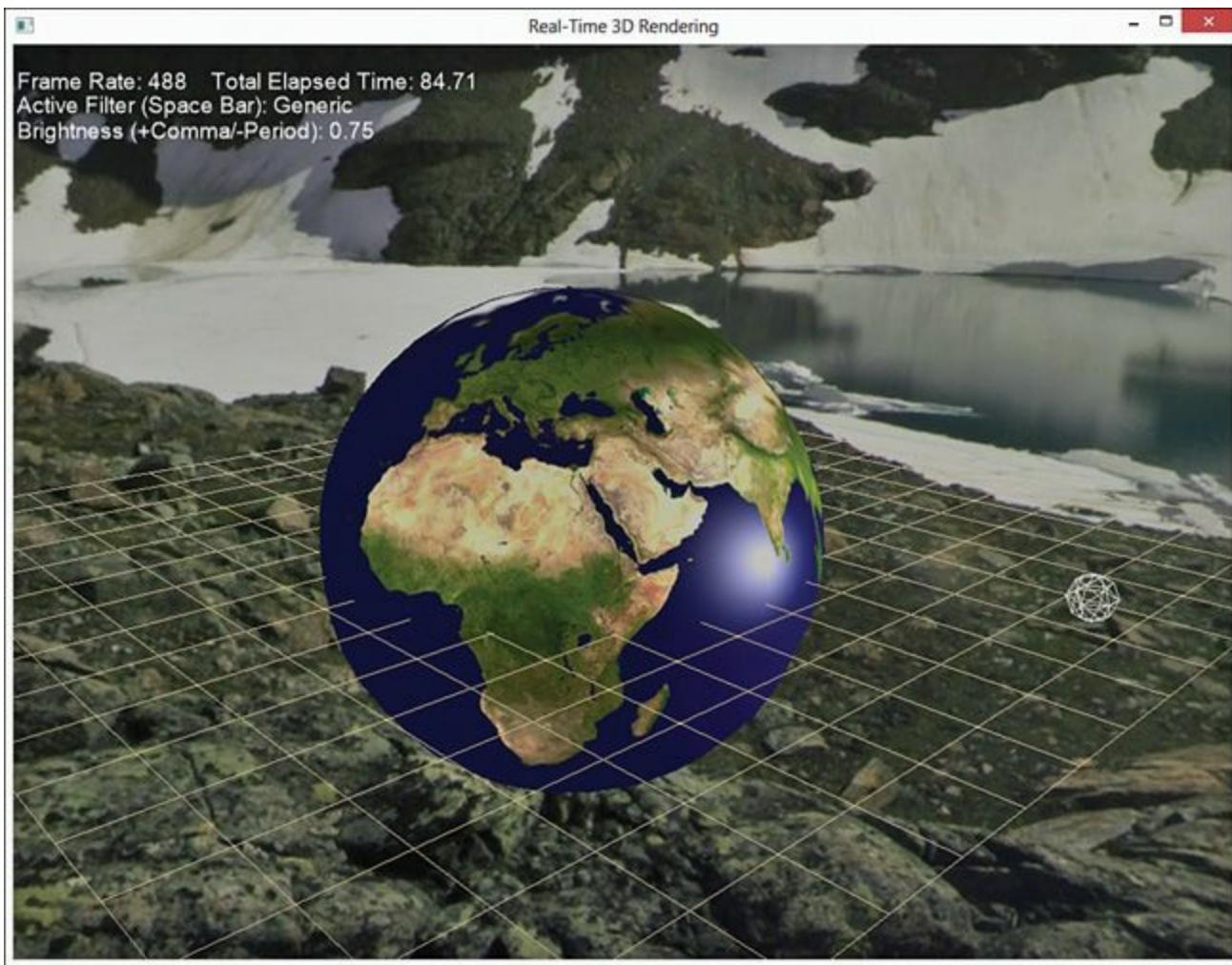
```

```

    mColorFilterMaterial->ColorTexture() << mRenderTarget->
OutputTexture();
    mColorFilterMaterial->ColorFilter() << colorFilter;
}

```

[Figure 18.4](#) shows the output of the generic color filter simulating a full-screen brightness effect.



**Figure 18.4** Output of the generic color filter shader simulating brightness. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

## Gaussian Blurring

Color filtering is just one of myriad effects that you can produce with post-processing. Another common technique is to blur the contents of the texture. You can achieve blurring with several approaches; the one presented here is Gaussian blurring, which takes its name from the Gaussian function (also known as the normal distribution) used to blur the image.

To blur an image, the color of each pixel is derived by sampling neighboring pixels. The weighted average of the sampled pixels becomes the color of the pixel in question. You derive the weights for the samples with the Gaussian function:

$$G(x) = e^{-\frac{x^2}{2\sigma^2}}$$

where,  $\sigma$  is the standard deviation of the Gaussian distribution. Another way to think of  $\sigma$  is that lower values yield a steeper curve, such that only pixels very close to the center (the pixel being

computed) have much bearing on the final color. Higher  $\sigma$  values give more weight to neighboring pixels and thereby blur the image more. In the coming implementation, we refer to  $\sigma$  as the *blur amount*.

Note that the previous equation is expressed in only one dimension, although you'll be using it to blur a two-dimensional texture. Gaussian blurring is *separable*, meaning that a two-dimensional blur can be applied as two independent one-dimensional calculations. In practice, this means that you blur the image first horizontally and then vertically.

Drawing with the Gaussian blurring effect can be broken down to the following steps:

1. Draw the scene to an off-screen render target.
2. Blur the scene texture horizontally to an off-screen render target.
3. Vertically blur the horizontally blurred texture and render it to the screen.

## A Gaussian Blurring Shader

[Listing 18.8](#) presents the GaussianBlur.fx effect. Its vertex inputs and vertex shader are identical to the ColorFilter.fx effect, but new are the SampleOffsets and SampleWeights arrays. The SampleOffsets array stores the locations of the nearby pixels to sample, relative to the pixel being computed. The SampleWeights array stores the coefficients for each sampled pixel.

### **Listing 18.8** The GaussianBlur.fx Effect

[Click here to view code image](#)

```
***** Resources *****

#define SAMPLE_COUNT 9

cbuffer CBufferPerFrame
{
    float2 SampleOffsets[SAMPLE_COUNT];
    float SampleWeights[SAMPLE_COUNT];
}

Texture2D ColorTexture;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = CLAMP;
    AddressV = CLAMP;
};

***** Data Structures *****

struct VS_INPUT
```

```

    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

//***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.ObjectPosition;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}

//***** Pixel Shaders *****/
float4 blur_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = (float4)0;

    for (int i = 0; i < SAMPLE_COUNT; i++)
    {
        color += ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate + SampleOffsets[i]) * SampleWeights[i];
    }

    return color;
}

float4 no_blur_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
}

//***** Techniques *****/
technique11 blur

```

```

{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
blur_pixel_shader()));
    }
}

technique11 no_blur
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
no_blur_pixel_shader()));
    }
}

```

---

In this effect, the number of samples is hard-coded to nine, but you can modify the effect to support additional samples or include multiple techniques that pass a uniform sample count to the pixel shader. Generally, you want to surround your pixel with a grid of neighboring pixels, so pick an odd number for the sample count. Understand, however, that the more samples you specify, the more work your pixel shader has to do.

The blur pixel shader iteratively samples the nearby pixels with the offset texture coordinates.

## A Gaussian Blurring Component

To integrate the blur shader into your C++ framework, create a `GaussianBlur` class with a declaration matching [Listing 18.9](#).

### **Listing 18.9** Declaration of the GaussianBlur Class

[Click here to view code image](#)

---

```

class GaussianBlur : public DrawableGameComponent
{
    RTTI_DECLARATIONS(GaussianBlur, DrawableGameComponent)

public:
    GaussianBlur(Game& game, Camera& camera);
    GaussianBlur(Game& game, Camera& camera, float blurAmount);
    ~GaussianBlur();

    ID3D11ShaderResourceView* SceneTexture();

```

```

void SetSceneTexture(ID3D11ShaderResourceView&
sceneTexture);

ID3D11ShaderResourceView* OutputTexture();

float BlurAmount() const;
void SetBlurAmount(float blurAmount);

virtual void Initialize() override;
virtual void Draw(const GameTime& gameTime) override;

private:
GaussianBlur();
GaussianBlur(const GaussianBlur& rhs);
GaussianBlur& operator=(const GaussianBlur& rhs);

void InitializeSampleWeights();
void InitializeSampleOffsets();
float GetWeight(float x) const;
void UpdateGaussianMaterialWithHorizontalOffsets();
void UpdateGaussianMaterialWithVerticalOffsets();
void UpdateGaussianMaterialNoBlur();

static const float DefaultBlurAmount;

Effect* mEffect;
GaussianBlurMaterial* mMaterial;
ID3D11ShaderResourceView* mSceneTexture;
FullScreenRenderTarget* mHorizontalBlurTarget;
FullScreenQuad* mFullScreenQuad;

std::vector<XMFLOAT2> mHorizontalSampleOffsets;
std::vector<XMFLOAT2> mVerticalSampleOffsets;
std::vector<float> mSampleWeights;
float mBlurAmount;
};


```

This class has data members for the Gaussian blurring effect, its associated material, and a member for the incoming scene texture (the image to blur). It has a render target for the horizontal blur operation, and its output texture becomes the input texture to the vertical blur pass. All rendering (to the render target or to the screen) is performed using the full-screen quad member.

The horizontal and vertical sample offsets and the sample weights members are populated by the `InitializeSampleOffsets()` and `InitializeSampleWeights()` methods. The `GetWeight()` method implements the Gaussian function to compute a single weight. The three methods whose names begin with “`UpdateGaussianMaterial`” are the callbacks for updating the full-screen quad material, according to the phase of the blurring process. The `Update`

`GaussianMaterialNoBlur()` method is used if `mBlurAmount` is zero. In that scenario, the shader's `no_blur` technique is applied to the scene texture, which just presents the unmodified texture to the screen.

You can modify the blur amount at runtime through the `SetBlurAmount()` method. Recall that the blur amount member is the Gaussian function's  $\sigma$  and, therefore, the `InitializeSampleWeights()` method is invoked (to rebuild the sample weights) whenever the blur amount changes. [Listing 18.10](#) presents the methods for initializing the sample offsets and weights.

## **Listing 18.10** Initializing the Gaussian Blurring Sample Offsets and Weights

[Click here to view code image](#)

---

```
void GaussianBlur::InitializeSampleOffsets()
{
    float horizontalPixelSize = 1.0f / mGame->ScreenWidth();
    float verticalPixelSize = 1.0f / mGame->ScreenHeight();

    UINT sampleCount = mMaterial-
>SampleOffsets().TypeDesc().Elements;

    mHorizontalSampleOffsets.resize(sampleCount);
    mVerticalSampleOffsets.resize(sampleCount);
    mHorizontalSampleOffsets[0] = Vector2Helper::Zero;
    mVerticalSampleOffsets[0] = Vector2Helper::Zero;

    for (UINT i = 0; i < sampleCount / 2; i++)
    {
        float sampleOffset = i * 2 + 1.5f;
        float horizontalOffset = horizontalPixelSize *
sampleOffset;
        float verticalOffset = verticalPixelSize * sampleOffset;

        mHorizontalSampleOffsets[i * 2 + 1] =
XMFLOAT2(horizontalOffset, 0.0f);
        mHorizontalSampleOffsets[i * 2 + 2] = XMFLOAT2
(-horizontalOffset, 0.0f);

        mVerticalSampleOffsets[i * 2 + 1] = XMFLOAT2
(0.0f, verticalOffset);
        mVerticalSampleOffsets[i * 2 + 2] = XMFLOAT2
(0.0f, -verticalOffset);
    }
}

void GaussianBlur::InitializeSampleWeights()
```

```

{
    UINT sampleCount = mMaterial-
>SampleOffsets().TypeDesc().Elements;

    mSampleWeights.resize(sampleCount);
    mSampleWeights[0] = GetWeight(0);

    float totalWeight = mSampleWeights[0];
    for (UINT i = 0; i < sampleCount / 2; i++)
    {
        float weight = GetWeight((float)i + 1);
        mSampleWeights[i * 2 + 1] = weight;
        mSampleWeights[i * 2 + 2] = weight;
        totalWeight += weight * 2;
    }

    // Normalize the weights so that they sum to one
    for (UINT i = 0; i < mSampleWeights.size(); i++)
    {
        mSampleWeights[i] /= totalWeight;
    }
}

float GaussianBlur::GetWeight(float x) const
{
    return (float)(exp(-(x * x) / (2 * mBlurAmount *
mBlurAmount)));
}

```

---

The InitializeSampleOffsets() method builds up the vertical and horizontal offsets around the center pixel (at index 0), based on the current resolution of your application. The InitializeSampleWeights() method assigns weights to the corresponding indices and then normalizes them so that they sum to 1. If the weights weren't normalized, they would increase (sum greater than 1) or decrease (sum less than 1) the brightness of the final image.

[Listing 18.11](#) presents the GaussianBlur::Draw() method.

## Listing 18.11 Drawing the Gaussian Blurring Component

[Click here to view code image](#)

---

```

void GaussianBlur::Draw(const GameTime& gameTime)
{
    if (mBlurAmount > 0.0f)
    {
        // Horizontal blur
        mHorizontalBlurTarget->Begin();

```

```

mGame->Direct3DDeviceContext()-
>ClearRenderTargetView(mHorizontalBlurTarget->RenderTargetView(),
, reinterpret_cast<const
float*>(&ColorHelper::Purple)));
    mGame->Direct3DDeviceContext()->ClearDepthStencilView
(mHorizontalBlurTarget->DepthStencilView(), D3D11_CLEAR_DEPTH |
D3D11_
CLEAR_STENCIL, 1.0f, 0);
        mFullScreenQuad->SetActiveTechnique("blur", "p0");
        mFullScreenQuad->SetCustomUpdateMaterial(std::bind
(&GaussianBlur::UpdateGaussianMaterialWithHorizontalOffsets,
this));
    mFullScreenQuad->Draw(gameTime);
    mHorizontalBlurTarget->End();

    // Vertical blur for the final image
    mFullScreenQuad->SetCustomUpdateMaterial(std::bind
(&GaussianBlur::UpdateGaussianMaterialWithVerticalOffsets,
this));
    mFullScreenQuad->Draw(gameTime);
}

else
{
    mFullScreenQuad->SetActiveTechnique("no_blur", "p0");
    mFullScreenQuad->SetCustomUpdateMaterial(std::bind
(&GaussianBlur::UpdateGaussianMaterialNoBlur, this));
    mFullScreenQuad->Draw(gameTime);
}
}

void GaussianBlur::UpdateGaussianMaterialWithHorizontalOffsets()
{
    mMaterial->ColorTexture() << mSceneTexture;
    mMaterial->SampleWeights() << mSampleWeights;
    mMaterial->SampleOffsets() << mHorizontalSampleOffsets;
}

void GaussianBlur::UpdateGaussianMaterialWithVerticalOffsets()
{
    mMaterial->ColorTexture() <<
mHorizontalBlurTarget->OutputTexture();
    mMaterial->SampleWeights() << mSampleWeights;
    mMaterial->SampleOffsets() << mVerticalSampleOffsets;
}

void GaussianBlur::UpdateGaussianMaterialNoBlur()
{

```

```
mMaterial->ColorTexture() << mSceneTexture;
```

```
}
```

---

If `mBlurAmount` is greater than 0, the blur technique is set for the full-screen quad member and the blur process is performed. The horizontal blur render target is bound to the output-merger stage, its views are cleared, and the quad is drawn using the

`UpdateGaussianMaterialWithHorizontalOffsets()` callback. This callback passes in the scene texture, the sample weights, and the horizontal sample offsets. After completing the draw, the back buffer is restored to the output-merger stage.

Next, the quad is drawn again, but this time using the

`UpdateGaussianMaterialWithVerticalOffsets()` callback. This method passes the output texture of the horizontal blur render target, the sample weights, and the vertical sample offsets. Because the back buffer is bound to the output-merger stage, the results of this draw are presented when the swap chain is flipped.

To integrate this component with your `Game`-derived class, initialize the component with code such as this:

[Click here to view code image](#)

```
mGaussianBlur = new GaussianBlur(*this, *mCamera);  
mGaussianBlur->SetSceneTexture(*(mRenderTarget-  
>OutputTexture()));  
mGaussianBlur->Initialize();
```

Here, `mRenderTarget` is used to render the scene. Following is the principal content within the `GaussianBlurGame::Draw()` method used for the demonstration application. You can find the full application on the companion website.

[Click here to view code image](#)

```
mRenderTarget->Begin();  
  
mDirect3DDeviceContext->ClearRenderTargetView(mRenderTarget->  
RenderTargetView(), reinterpret_cast<const float*>  
(&BackgroundColor));  
mDirect3DDeviceContext->ClearDepthStencilView(mRenderTarget->  
DepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL,  
1.0f, 0);  
  
Game::Draw(gameTime);  
  
mRenderTarget->End();  
  
mDirect3DDeviceContext->ClearRenderTargetView(mRenderTargetView,  
reinterpret_cast<const float*>(&BackgroundColor));  
mDirect3DDeviceContext->ClearDepthStencilView(mDepthStencilView,  
D3D11_  
CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
```

```
mGaussianBlur->Draw(gameTime);
```

[Figure 18.5](#) shows the output of the Gaussian blurring demo with a blur amount of 3.0.



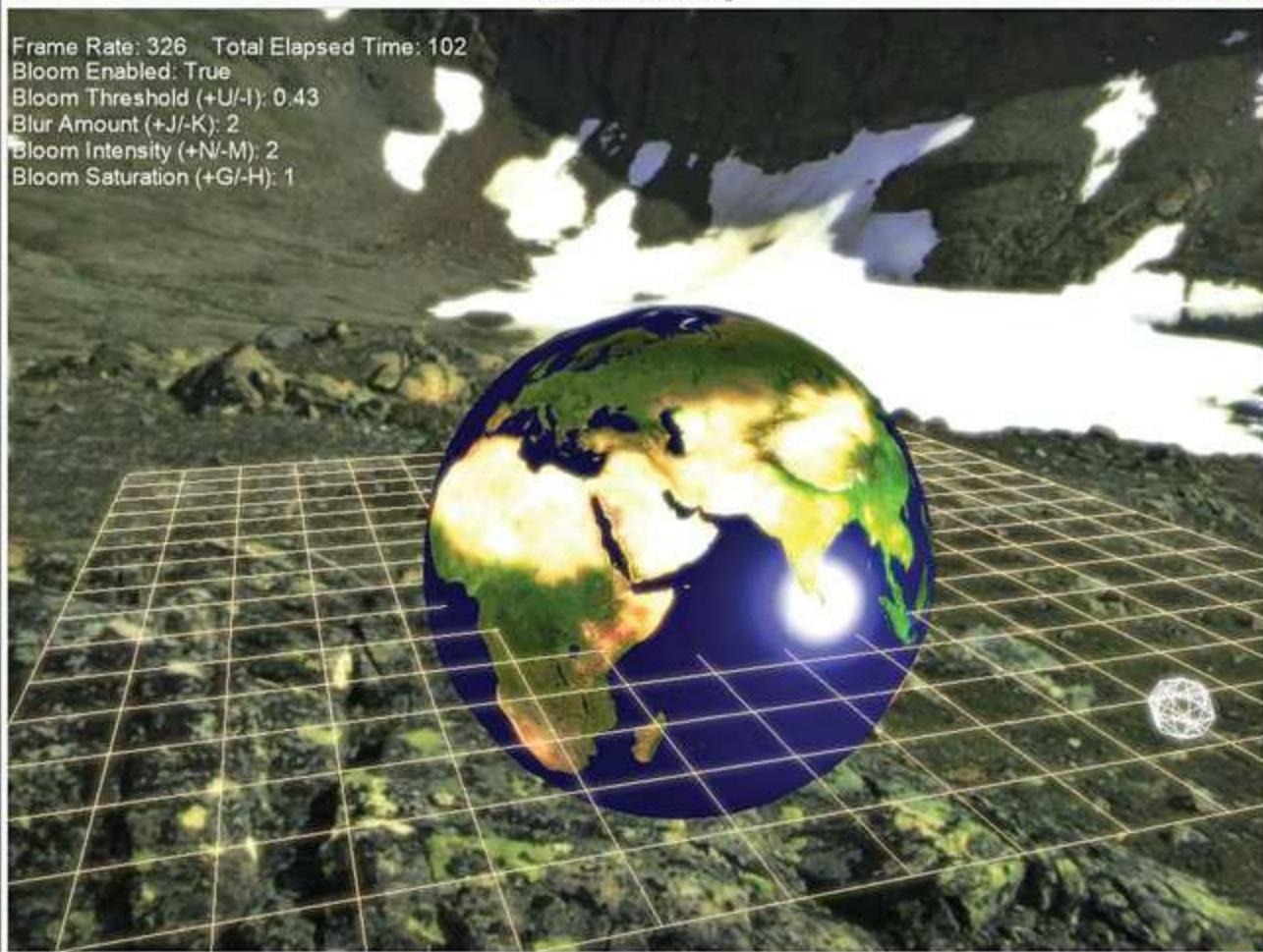
**Figure 18.5** Output of the Gaussian blurring effect. (Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.)

## Bloom

The Gaussian blurring effect has a variety of applications. For example, you could use it to distinguish background (blurred to appear out of focus) and foreground (unblurred) objects. Another use is with a **bloom** (or **glow**) effect. A bloom effect exaggerates bright areas of the scene to simulate a real-world camera. [Figure 18.6](#) shows a scene with bloom (top image) and without bloom.

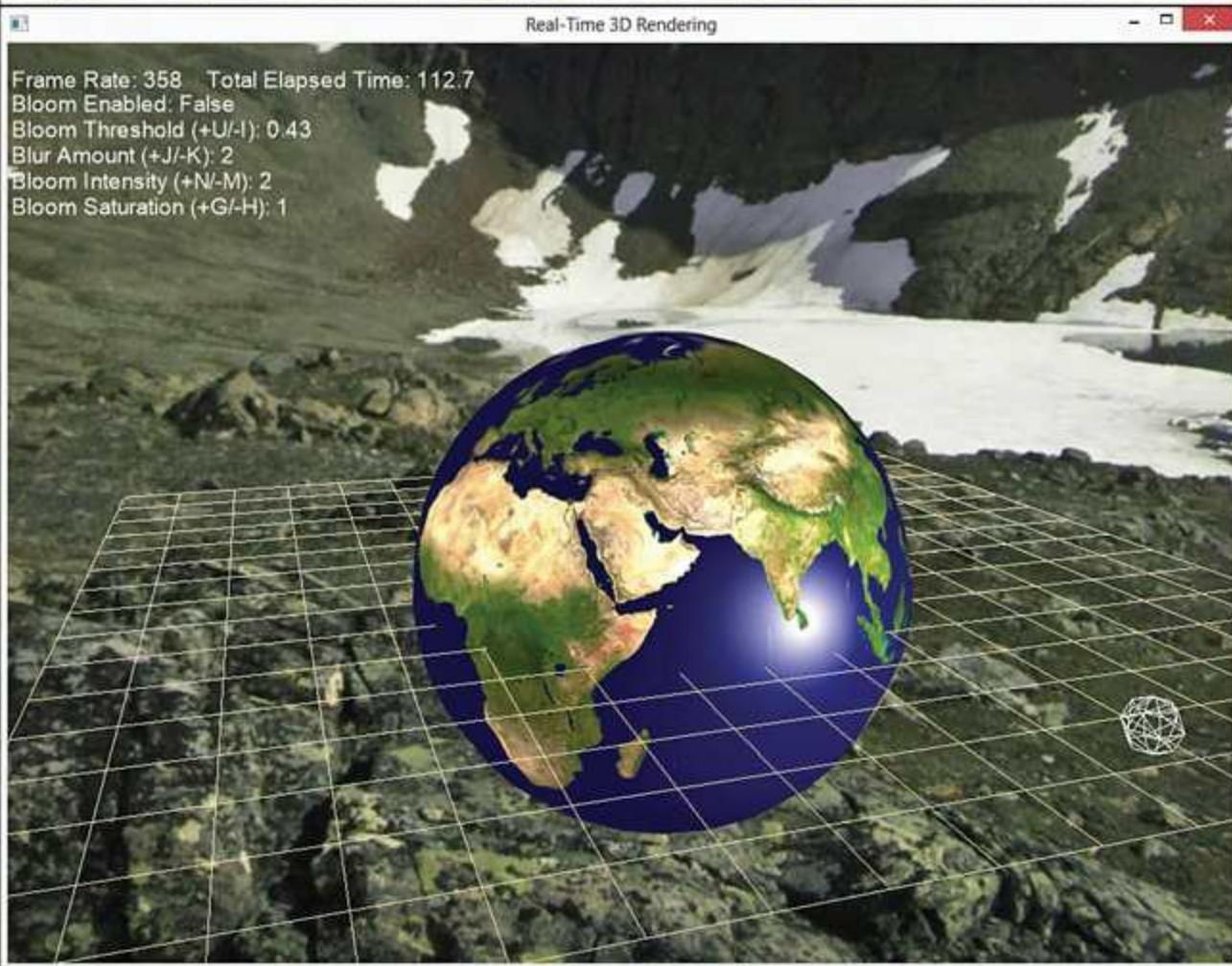
Real-Time 3D Rendering

Frame Rate: 326 Total Elapsed Time: 102  
Bloom Enabled: True  
Bloom Threshold (+U/-I): 0.43  
Blur Amount (+J/-K): 2  
Bloom Intensity (+N/-M): 2  
Bloom Saturation (+G/-H): 1



Real-Time 3D Rendering

Frame Rate: 358 Total Elapsed Time: 112.7  
Bloom Enabled: False  
Bloom Threshold (+U/-I): 0.43  
Blur Amount (+J/-K): 2  
Bloom Intensity (+N/-M): 2  
Bloom Saturation (+G/-H): 1



**Figure 18.6** A scene rendered with (top) and without (bottom) a bloom effect. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

You can create the bloom effect using the following steps:

1. Draw the scene to an off-screen render target.
2. Extract the bright spots from the scene image to an off-screen render target (create a **glow map**).
3. Blur the glow map to an off-screen render target.
4. Combine the blurred glow map with the original scene texture, and render it to the screen.

[Listing 18.12](#) presents the Bloom.fx effect. It includes three separate techniques: one to extract the bright areas of an input texture to create a glow map, another to combine a blurred glow map with the original scene, and a third to render the original scene unmodified. The Gaussian blurring shader is not replicated in this effect because you reuse the previously authored GaussianBlur component to blur the extracted glow map.

### **Listing 18.12** The Bloom.fx Effect

[Click here to view code image](#)

```
***** Resources *****/
static const float3 GrayScaleIntensity = { 0.299f, 0.587f,
0.114f };

Texture2D ColorTexture;
Texture2D BloomTexture;

cbuffer CBufferPerObject
{
    float BloomThreshold = 0.45f;
    float BloomIntensity = 1.25f;
    float BloomSaturation = 1.0f;
    float SceneIntensity = 1.0f;
    float SceneSaturation = 1.0f;
};

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

***** Data Structures *****/
struct VS_INPUT
{
```

```

    float4 Position : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/************************************************************************************************ Utility Functions *****/
float4 AdjustSaturation(float4 color, float saturation)
{
    float intensity = dot(color.rgb, GrayScaleIntensity);

    return float4(lerp(intensity.rrr, color.rgb, saturation), color.a);
}

/************************************************************************************************ Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}

/************************************************************************************************ Pixel Shaders *****/
float4 bloom_extract_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return saturate((color - BloomThreshold) / (1 -
BloomThreshold));
}

float4 bloom_composite_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 sceneColor = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

```

```
    float4 bloomColor = BloomTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    sceneColor = AdjustSaturation(sceneColor, SceneSaturation) *
SceneIntensity;
    bloomColor = AdjustSaturation(bloomColor, BloomSaturation) *
BloomIntensity;

    sceneColor *= (1 - saturate(bloomColor));

    return sceneColor + bloomColor;
}

float4 no_bloom_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
}

/************* Techniques *****/
technique11 bloom_extract
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
bloom_extract_pixel_shader()));
    }
}

technique11 bloom_composite
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
bloom_composite_pixel_shader()));
    }
}

technique11 no_bloom
{
    pass p0
{
```

```
SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
SetGeometryShader(NULL);
SetPixelShader(CompileShader(ps_5_0,
no_bloom_pixel_shader())));
}
}
```

---

The bloom\_extract\_pixel\_shader compares the sampled color to a passed-in threshold. If the color is less than the threshold, the output pixel is black. The color is further modulated by the inverse of the threshold. This isn't the only way you might create a glow map: An alternative is to use the intensity of the sampled texture in comparison with the bloom threshold. For example:

[Click here to view code image](#)

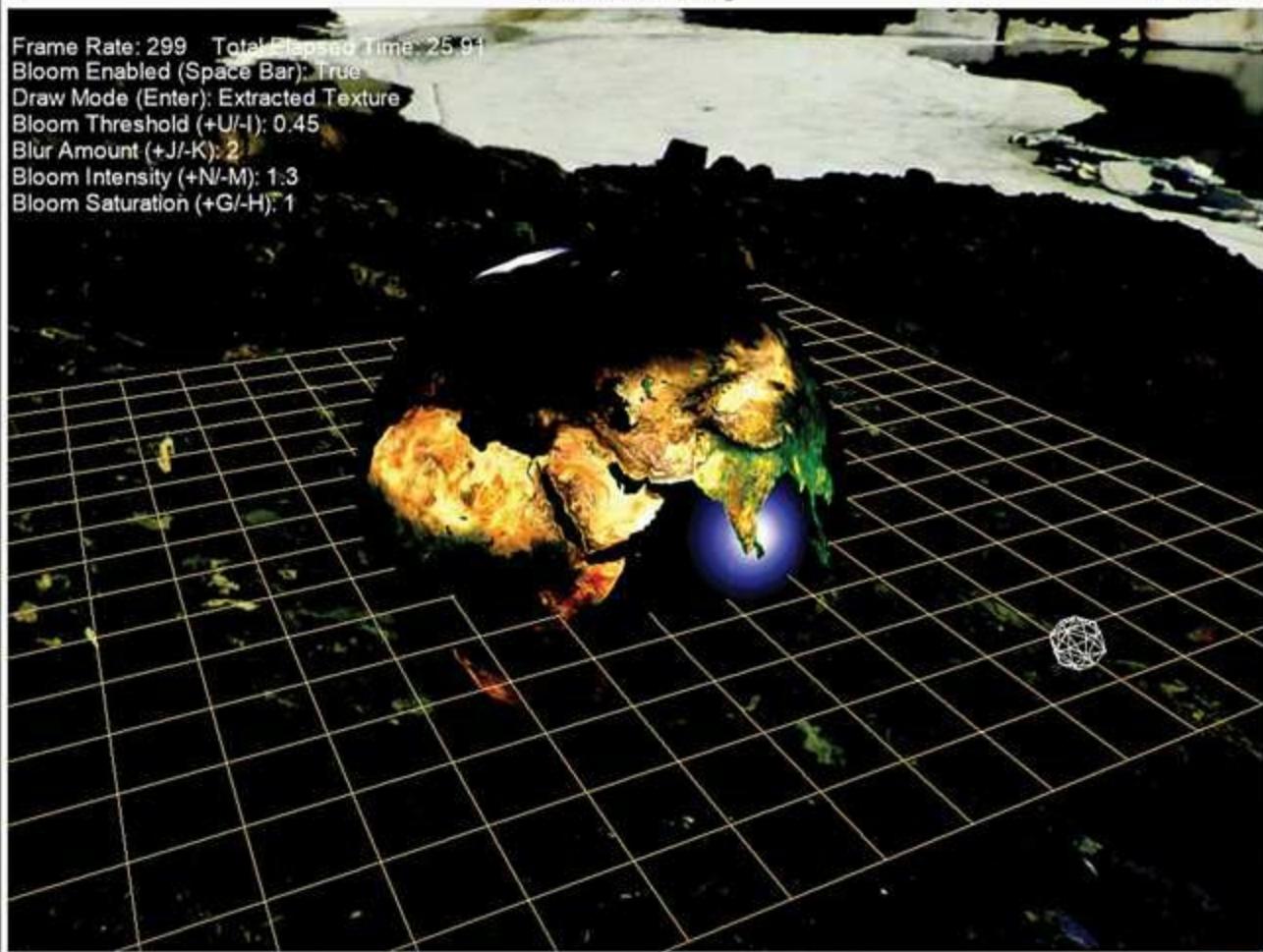
```
float4 bloom_extract_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float intensity = dot(color.rgb, GrayScaleIntensity);

    return (intensity > BloomThreshold ? color : float4(0, 0, 0,
1));
}
```

[Figure 18.7](#) shows a glow map extracted from a scene using both of these pixel shaders.

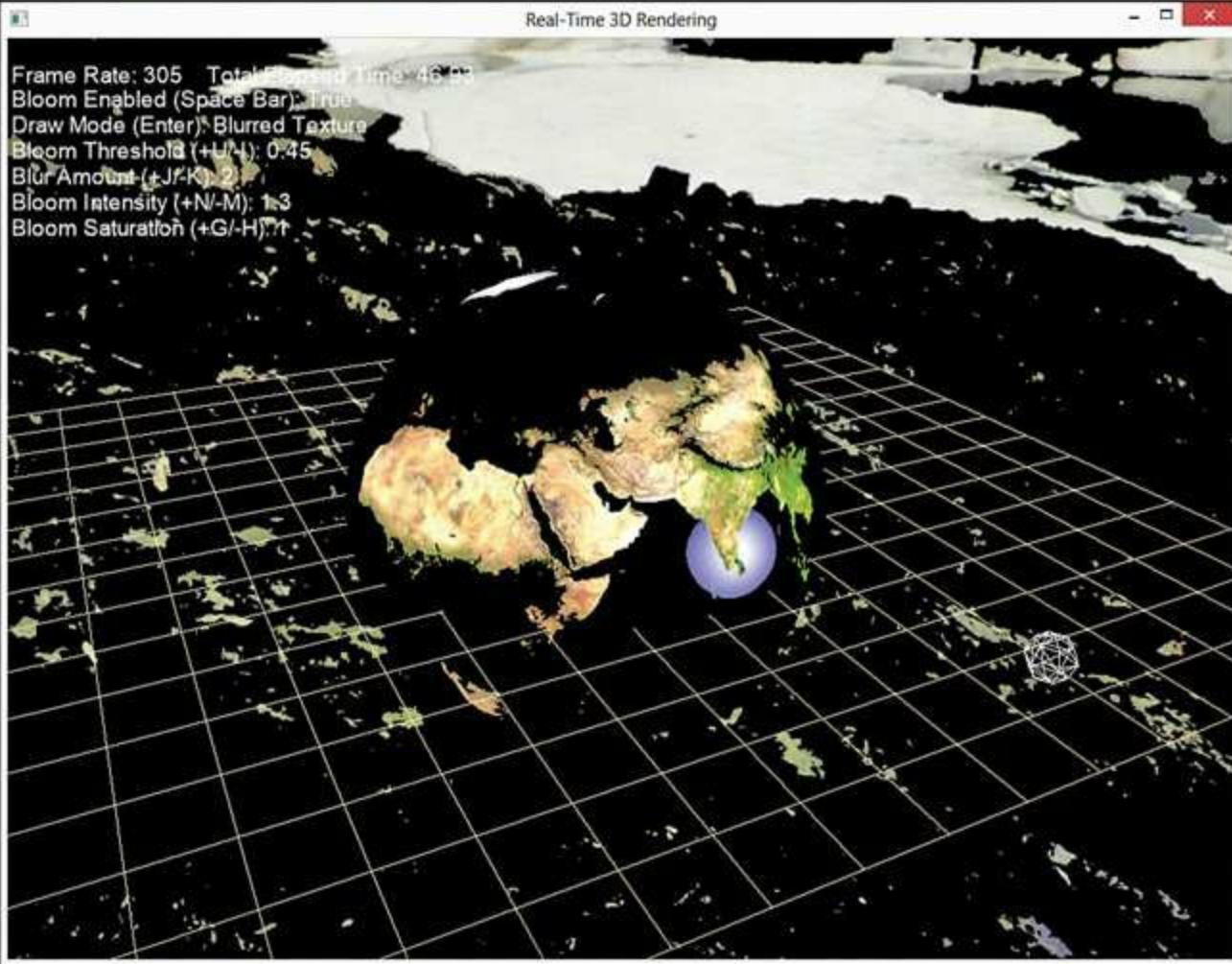
## Real-Time 3D Rendering

Frame Rate: 299 Total Elapsed Time: 25.91  
Bloom Enabled (Space Bar): True  
Draw Mode (Enter): Extracted Texture  
Bloom Threshold (+U/-I): 0.45  
Blur Amount (+J/-K): 2  
Bloom Intensity (+N/-M): 1.3  
Bloom Saturation (+G/-H): 1



## Real-Time 3D Rendering

Frame Rate: 305 Total Elapsed Time: 46.93  
Bloom Enabled (Space Bar): True  
Draw Mode (Enter): Blurred Texture  
Bloom Threshold (+U/-I): 0.45  
Blur Amount (+J/-K): 2  
Bloom Intensity (+N/-M): 1.3  
Bloom Saturation (+G/-H): 1



**Figure 18.7** Glow maps extracted from a scene using the original method (top) and an alternate method (bottom). (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

The glow map is blurred by the Gaussian blurring component to give the glow a “bleed over” effect as the light intrudes into other pixels. This blurred glow map is combined with the original scene texture, but you can further modulate it with intensity and saturation values. The demo application on the companion website enables you to interact with these values. Experimentation will produce some interesting results.

## Distortion Mapping

The final post-processing technique we cover in this chapter is **distortion mapping**. Distortion mapping is akin to the displacement mapping effect in [Chapter 9, “Normal Mapping and Displacement Mapping.”](#) But where displacement mapping alters an object’s vertices, distortion mapping alters its pixels. More specifically, you sample a texture (a distortion map) for horizontal and vertical offsets and apply these offsets to the UV for lookup into the scene texture.

## A Full-Screen Distortion Shader

[Listing 18.13](#) presents the pixel shader for a full-screen post-processing distortion effect.

### Listing 18.13 A Full-Screen Distortion Mapping Pixel Shader

[Click here to view code image](#)

```
cbuffer CBufferPerObject
{
    float DisplacementScale = 1.0f;
}

Texture2D SceneTexture;
Texture2D DistortionMap;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = CLAMP;
    AddressV = CLAMP;
};

float4 displacement_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy - 0.5;
    OUT = SceneTexture.Sample(TrilinearSampler,
```

```

IN.TextureCoordinate + (DisplacementScale * displacement));
return OUT;
}

```

This shader samples two channels (x and y) for horizontal and vertical displacement. These are 8-bit channels mapped from [0, 255] to floating point values [0.0, 1.0]. However, the pixel displacement can be positive or negative, so you must scale the range. The specific scale depends on how your displacement maps are created, but this example scales the range to [-0.5, 0.5]. From an artist's perspective, that means that the channel value 127 yields (almost) no displacement. I say *almost* because  $127 / 255 \approx 0.49804$ , so there's an error of  $0.5 - 0.49804 = .00196$ . You can compensate for this error by adding this value into the displacement calculation. For example:

[Click here to view code image](#)

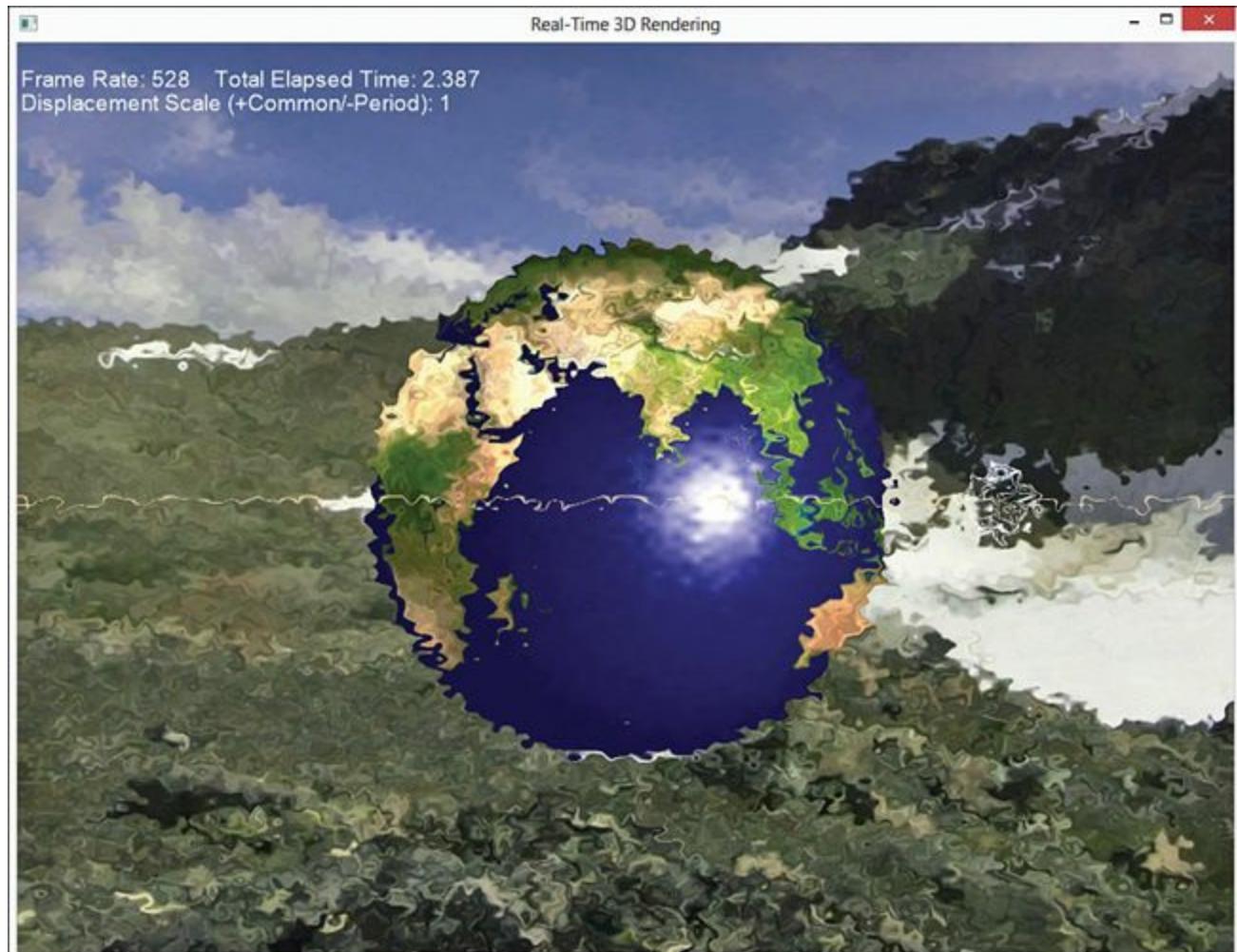
```

static const float ZeroCorrection = 0.5f / 255.0f;

float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy - 0.5 + ZeroCorrection;

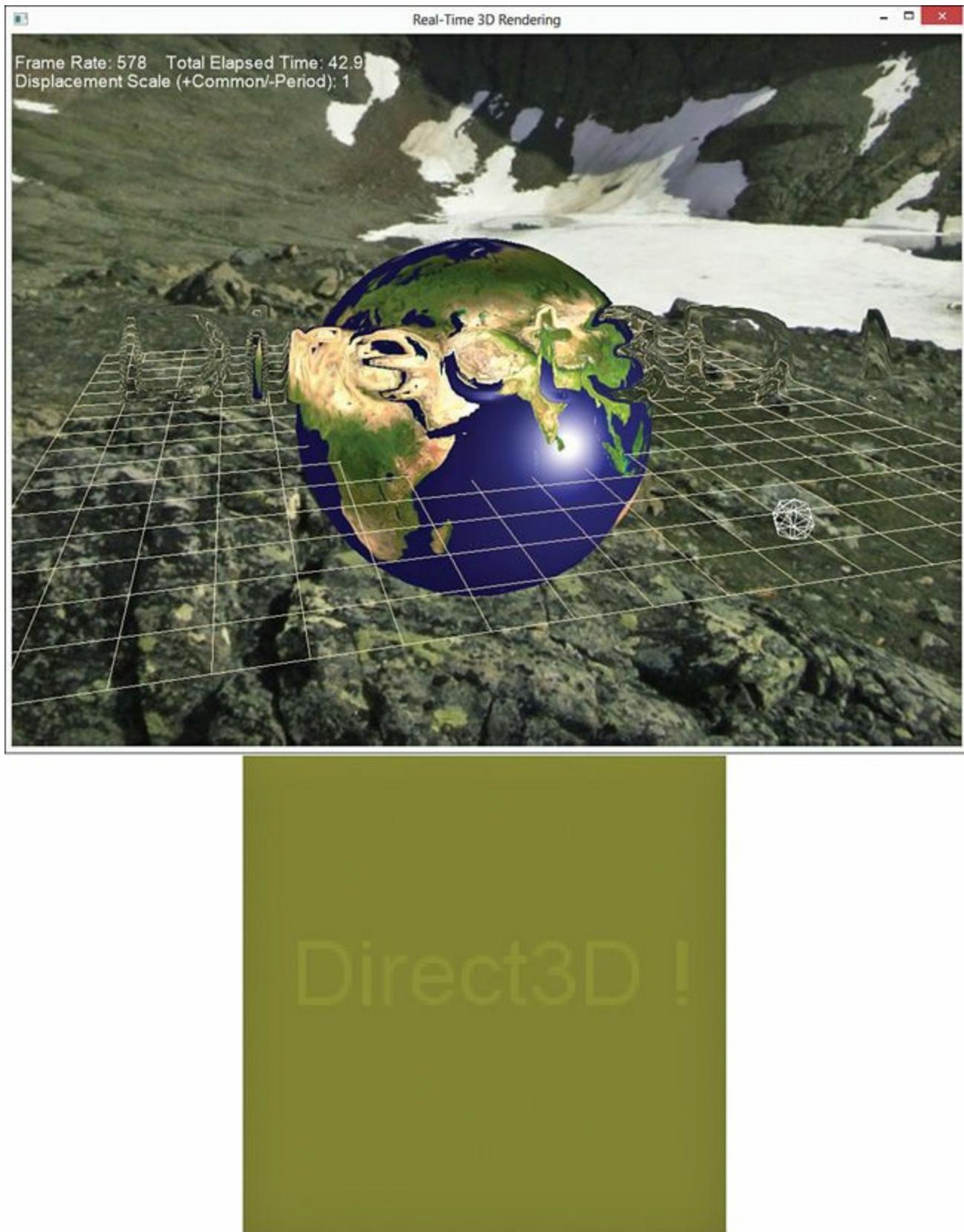
```

Rendering with this shader is normal post-processing—identical to the color filtering shaders. You render your scene to an off-screen render target and then render a full-screen quad to the back buffer using the distortion mapping shader. [Figure 18.8](#) shows the output of the displacement shader using a distortion map made to look like warped glass.



**Figure 18.8** Output of the full-screen distortion mapping shader using a distortion map resembling warped glass. (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

The variation in the warped glass distortion map is so subtle that you wouldn't be able to easily distinguish the perturbations (even with a magnified image). Thus, [Figure 18.9](#) shows the output of the displacement shader (top) using a distortion map (bottom) containing the text “Direct3D !”. Notice the yellowish appearance of the distortion map; this is because only the red and green channels are populated.

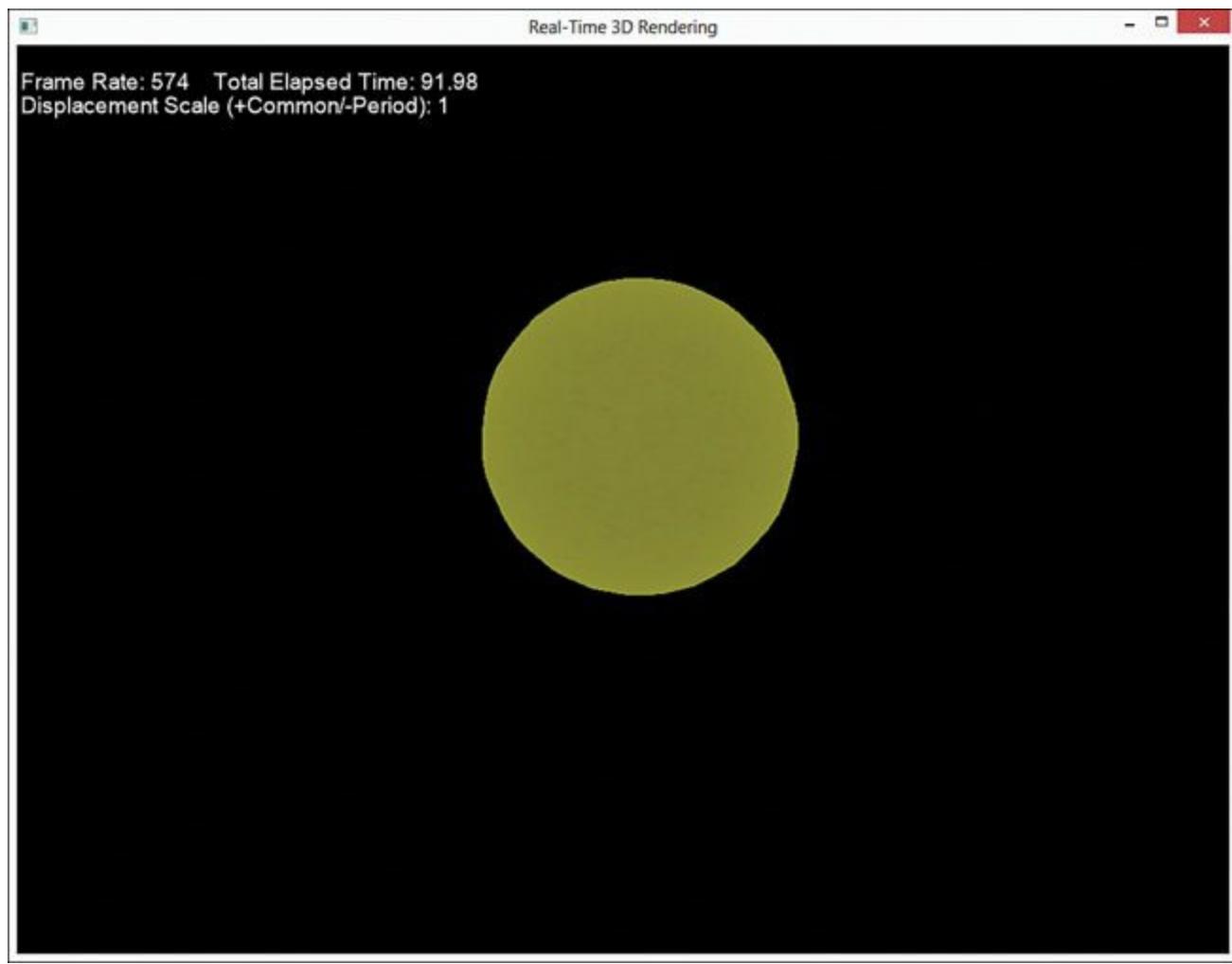


**Figure 18.9** Output of the full-screen distortion mapping shader (top) and associated distortion map (bottom). (*Skybox texture by Emil Persson. Earth texture from Reto Stöckli, NASA Earth Observatory.*)

## A Masking Distortion Shader

The full-screen distortion mapping shader can produce some interesting effects—even more interesting if you animate the displacement offsets (accomplished by modulating the offsets by some continuously changing value, such as time). However, you might not want to distort the entire scene. For example, you could simulate the heat haze above a fire (or hot pavement) and want to isolate the effect to that specific location.

One way to accomplish this is to create a run-time **distortion mask**, a full-screen distortion map with a cutout of the area you want to be distorted. [Figure 18.10](#) shows a distortion mask created using the warped glass distortion map and a sphere.



**Figure 18.10** A distortion mask made with a sphere.

This distortion mask is created with an off-screen render target by first clearing the target to black pixels and then rendering objects onto the target. However, instead of rendering colors from an object's color texture, you look up offsets from a distortion map. Thus, the created distortion mask is made up of either black pixels or distortion offsets. After the mask is created, you perform a post-processing pass between the scene texture and the distortion mask. In this step, you first sample from the distortion mask. If its value is black, you sample the scene texture without displacing its UV. Otherwise, you displace the UV with the offsets sampled from the mask. [Listing 18.14](#) presents the

complete source code for this effect.

## Listing 18.14 A Masking Distortion Mapping Effect

[Click here to view code image](#)

```
***** Resources *****
static const float ZeroCorrection = 0.5f / 255.0f;

cbuffer CBufferPerObjectCutout
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

cbuffer CBufferPerObjectComposite
{
    float DisplacementScale = 1.0f;
}

Texture2D SceneTexture;
Texture2D DistortionMap;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

***** Data Structures *****
struct VS_INPUT
{
    float4 Position : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

***** Cutout *****
VS_OUTPUT distortion_cutout_vertex_shader(VS_INPUT IN)
{
```

```

VS_OUTPUT OUT = (VS_OUTPUT)0;

OUT.Position = mul(IN.Position, WorldViewProjection);
OUT.TextureCoordinate = IN.TextureCoordinate;

return OUT;
}

float4 distortion_cutout_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy;

    return float4(displacement.xy, 0, 1);
}

/***************** Distortion Post Processing *****/
VS_OUTPUT distortion_vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}

float4 distortion_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy;
    if (displacement.x == 0 && displacement.y == 0)
    {
        OUT = SceneTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    }
    else
    {
        displacement -= 0.5f + ZeroCorrection;
        OUT = SceneTexture.Sample(TrilinearSampler,
IN.TextureCoordinate + (DisplacementScale * displacement));
    }

    return OUT;
}

```

```

}

float4 no_distortion_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return SceneTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
}

/***** Techniques *****/
technique11 distortion_cutout
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
distortion_cutout_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
distortion_cutout_pixel_shader()));
    }
}

technique11 distortion
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
distortion_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
distortion_pixel_shader()));
    }
}

technique11 no_distortion
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
distortion_composite_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
no_distortion_pixel_shader()));
    }
}

```

---

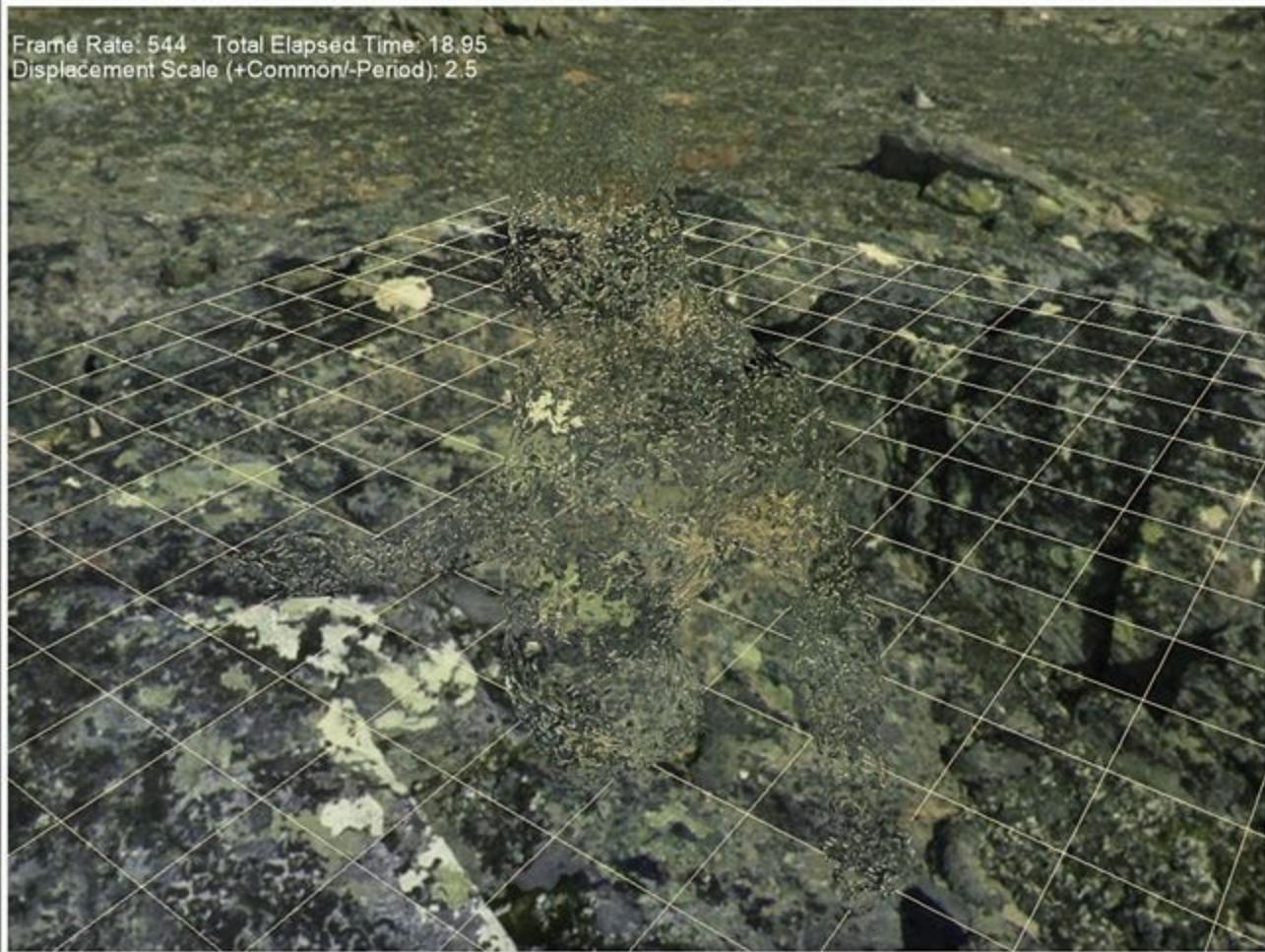
This shader uses three techniques:

- **distortion\_cutout:** References vertex and pixel shaders for creating the distortion mask. The vertex shader expects the incoming vertex positions to be in object space and, therefore, transforms them. The pixel shader outputs the x and y channels from the distortion map using the regular texture coordinates of the vertices.
- **distortion:** Defines the post-processing step between the scene texture and the distortion mask. The vertex shader expects the incoming vertex positions to be in screen space and does not transform them (as with all the post-processing effects). The pixel shader outputs the sampled scene texture either with or without a displacement offset, depending on the values of the offsets.
- **no\_distortion:** Outputs the scene texture with no distortion.

Rendering the masking distortion shader requires two render targets, one to render the un-distorted background objects and one to create the distortion mask. You can find a full demonstration application on the companion website. [Figure 18.11](#) shows the output of the masking distortion shader (top) using a model of a humanoid torso to create the distortion mask (bottom).

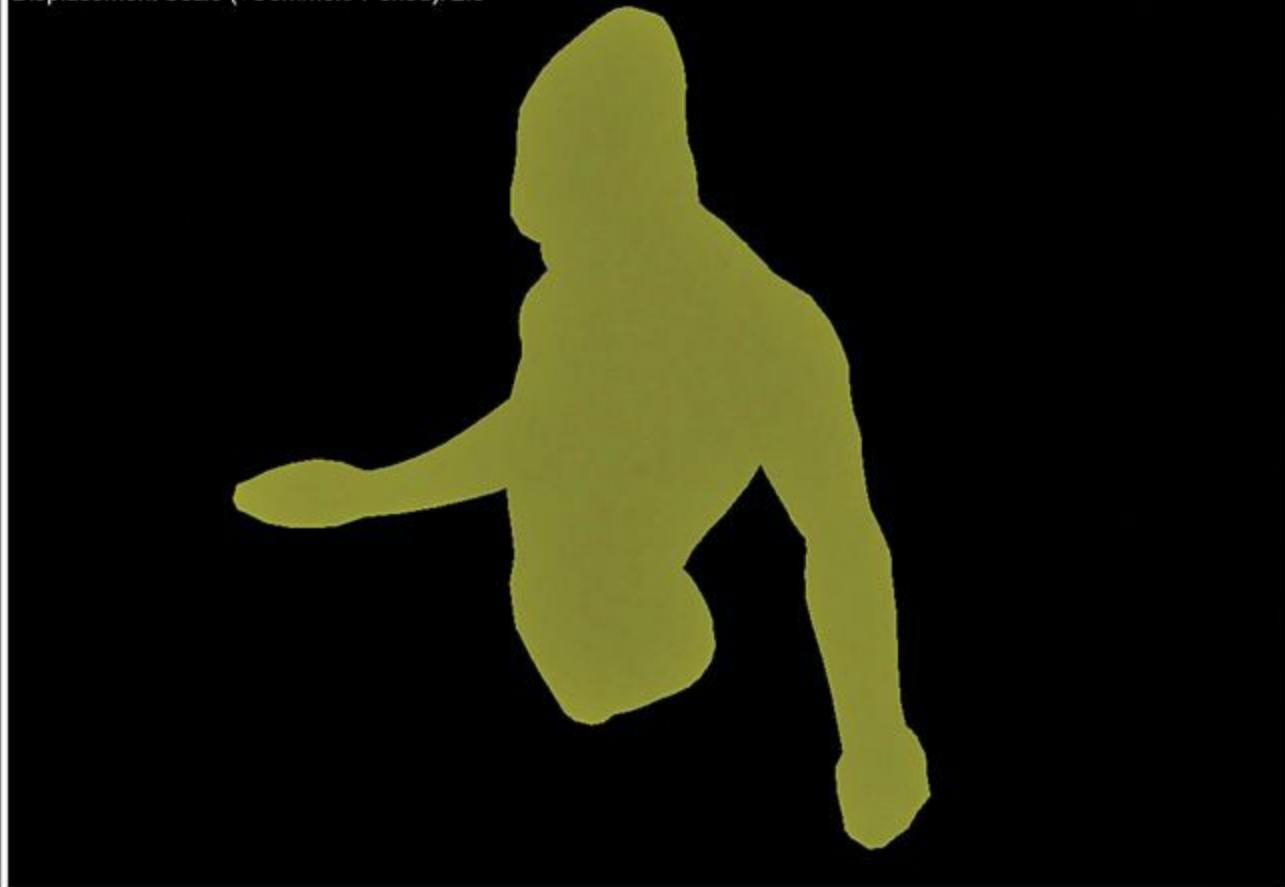
Real-Time 3D Rendering

Frame Rate: 544 Total Elapsed Time: 18.95  
Displacement Scale (+Common/-Period): 2.5



Real-Time 3D Rendering

Frame Rate: 547 Total Elapsed Time: 45.84  
Displacement Scale (+Common/-Period): 2.5



**Figure 18.11** Output of the distortion masking shader (top) and associated distortion mask (bottom). (*Skybox texture by Emil Persson. 3D model by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

## Summary

This chapter introduced the topic of post-processing, graphics techniques applied *after* the scene is rendered. You authored a number of post-processing effects for color filtering, Gaussian blurring, bloom/glow, and distortion mapping. You integrated these effects into your C++ rendering framework with full-screen render target and quad components, and you explored demonstration applications to exercise these effects.

In the next chapter, you learn techniques for projective texture mapping and shadow mapping.

## Exercises

1. Experiment with all the effects and demo applications from this chapter. Vary the shader inputs, and observe the results.
2. Create your own distortion maps for the post-processing distortion shader, and use them with the associated demo application.
3. Animate the masking distortion shader to simulate heat haze above a fire. *Hint:* Use `GameTime::TotalGameTime()` as input into the shader.

# Chapter 19. Shadow Mapping

In this chapter, you learn how to render shadows. You learn about depth maps (2D textures that store depths instead of colors) and projective texture mapping (think the Batman signal). You also examine common problems with rendering shadows and determine how to address them.

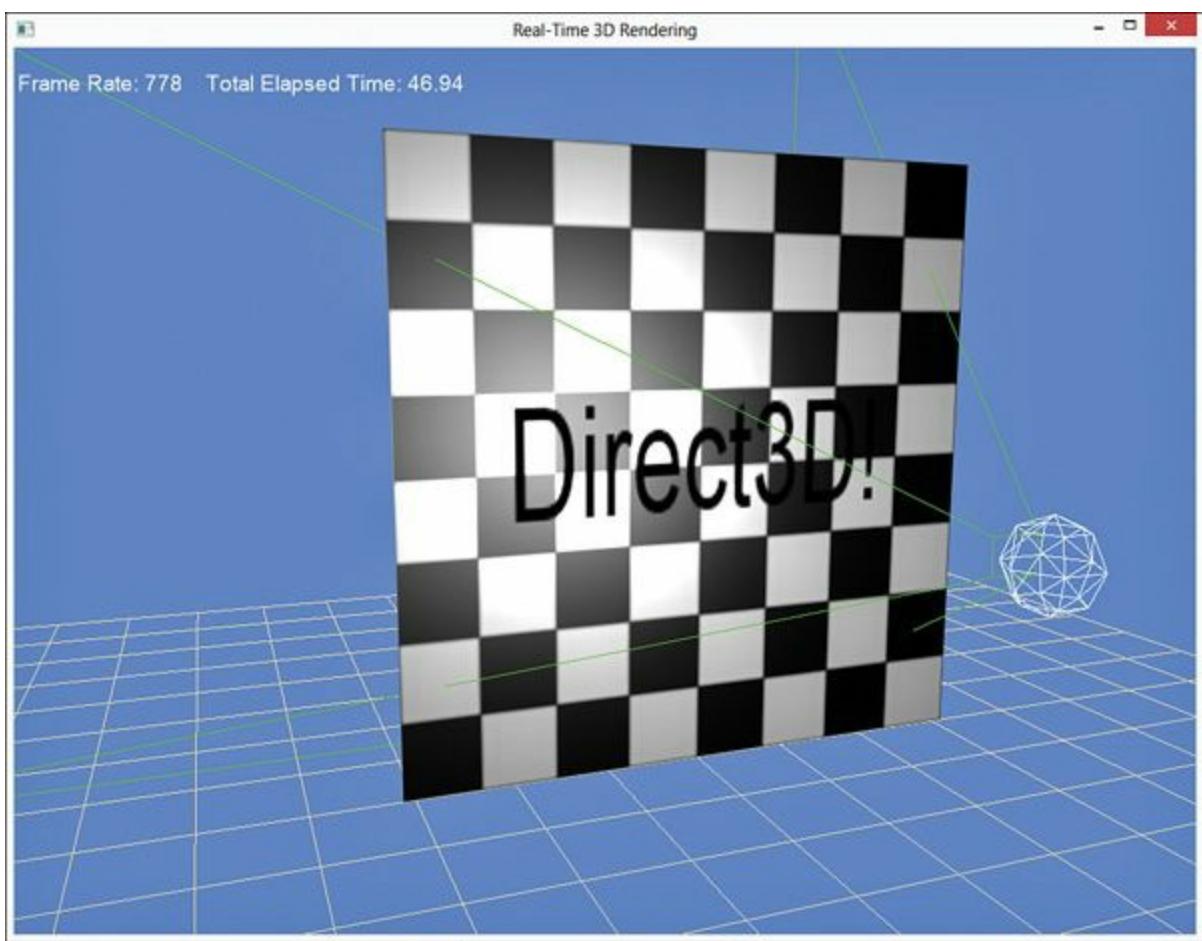
## Motivation

Shadows are part of our visual experience, but unless you're making shadow puppets, you don't likely even notice the myriad shadows that surround you. Remove them, however, and something will feel off. That's a conundrum for video games and simulations because realistic shadows are difficult to produce and computationally expensive. They are subtle and almost unconsciously perceived, but leave them out, and your users will notice that something about the scene is just not quite right. In this chapter, we discuss **shadow mapping**, a common technique for producing shadows. Along the way, we introduce **projective texture mapping**, a technique for projecting a 2D texture onto arbitrary geometry. We also discuss the various shortcomings of projective texture mapping and shadow mapping, and explore ways to overcome them.

## Projective Texture Mapping

Shadow mapping involves producing a **depth map**, a 2D texture that stores the depths of objects nearest the light source, and mapping that texture onto the geometry you want to accept a shadow (the shadow's **receiver**). Thus, a first step to understanding shadow mapping is to discuss projective texture mapping. Projective texture mapping is analogous to a slide or movie projector. The **projector** is defined by a frustum, in the same way as a virtual camera, and is used to generate texture coordinates to sample the projected image.

[Figure 19.1](#) shows the output of a projective texture mapping shader with a texture of the text "Direct3D!" projected onto a plane colored with a checkerboard pattern. The green lines depict the frustum of the projector, and the white sphere represents a point light in the scene.



Direct3D!

**Figure 19.1** Output of the projective texture mapping shader (top) with the projected texture (bottom).

Three helper classes are used for projective texture mapping: Projector, Frustum, and RenderableFrustum; see [Figure 19.2](#) for the class diagrams. We do not list these classes here, but you can find the full source code on the companion website.

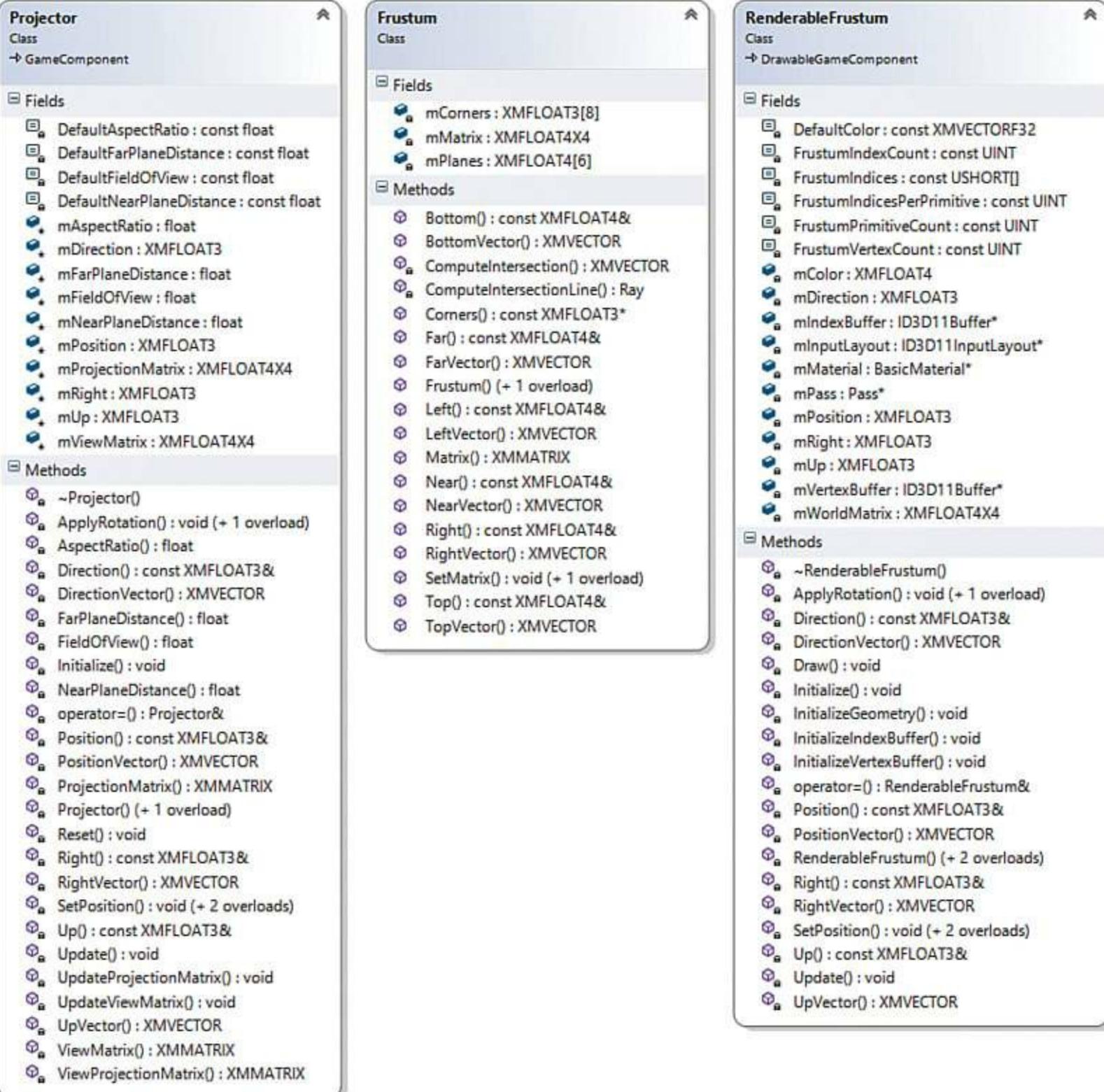


Figure 19.2 Class diagrams for projective texture-mapping helper classes.

## Projective Texture Coordinates

The `Projector` class is nearly identical to the `Camera` class, and its output is the `ViewProjection` matrix. This matrix transforms an object into projection space from the perspective of the projector instead of from the perspective of the camera. You can use the transformed position (the x and y components) as the lookup into the projected texture. However, the coordinates will be in **normalized device coordinate space** (NDC space, or screen space) whose range is  $[-1, 1]$ , and a texture is mapped with a range  $[0, 1]$ . Therefore, you must scale the projective texture coordinates as follows:

$$u = 0.5x + 0.5$$

$$v = -0.5y + 0.5$$

These equations can be composed in matrix form as:

$$Scaling_{Texture} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix}$$

If you concatenate the object's world matrix, the projector's view and projection matrices, and the texture scaling matrix, you can transform an object from local space to projective texture space with a single matrix multiplication. We call this construct a projective texture matrix.

## A Projective Texture-Mapping Shader

[Listing 19.1](#) presents a shader for projective texture mapping. It uses the point light shader as its foundation, with variables for a single point light, ambient light, and specular highlights. New are variables for the `ProjectiveTextureMatrix` and the `ProjectedTexture`.

### **Listing 19.1** An Initial Projective Texture-Mapping Shader

[Click here to view code image](#)

---

```
#include "include\\Common.fhx"

/****************** Resources *****/
static const float4 ColorWhite = { 1, 1, 1, 1 };

cbuffer CBufferPerFrame
{
    float4 AmbientColor = { 1.0f, 1.0f, 1.0f, 0.0f };
    float4 LightColor = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 LightPosition = { 0.0f, 0.0f, 0.0f };
    float LightRadius = 10.0f;
    float3 CameraPosition;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
    float4 SpecularColor : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
};

    float SpecularPower : SPECULARPOWER = 25.0f;

    float4x4 ProjectiveTextureMatrix;
}
```

```
Texture2D ColorTexture;
Texture2D ProjectedTexture;

SamplerState ProjectedTextureSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
};

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState BackFaceCulling
{
    CullMode = BACK;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
    float4 ProjectedTextureCoordinate : TEXCOORD3;
};

/****** Vertex Shader *****/

VS_OUTPUT project_texture_vertex_shader(VS_INPUT IN)
{
```

```

VS_OUTPUT OUT = (VS_OUTPUT)0;

OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
OUT.TextureCoordinate = IN.TextureCoordinate;
OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);
float3 lightDirection = LightPosition - OUT.WorldPosition;
OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));

OUT.ProjectedImageCoordinate = mul(IN.ObjectPosition,
ProjectiveTextureMatrix);

return OUT;
}

```

```

***** Pixel Shaders *****/
float4 project_texture_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
-IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;
}
```

```

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    IN.ProjectedTextureCoordinate.xy /= 
IN.ProjectedTextureCoordinate.w;
    float3 projectedColor = ProjectedTexture.
Sample(ProjectedTextureSampler,
IN.ProjectedTextureCoordinate.xy).rgb;

    OUT.rgb *= projectedColor;

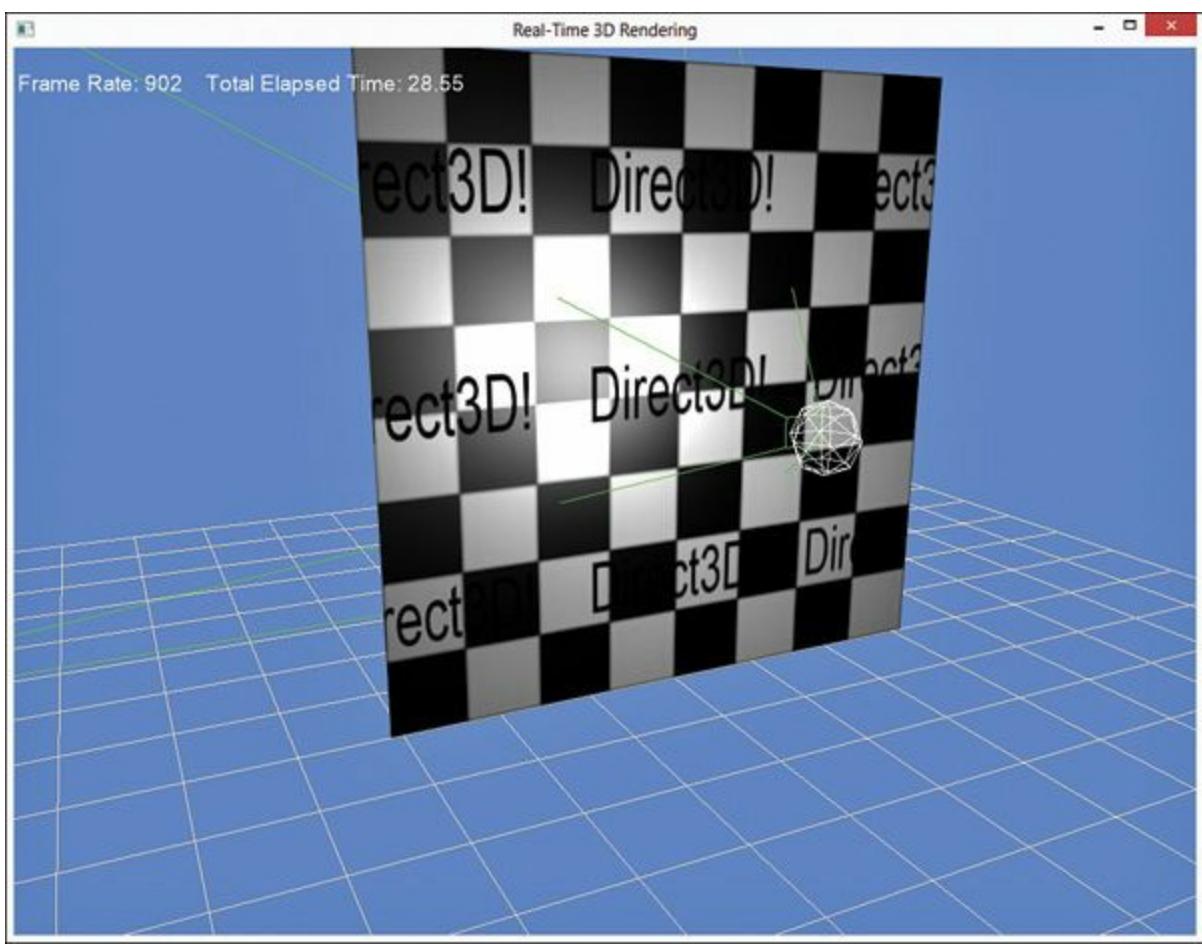
    return OUT;
}

/***** Techniques *****/
technique11 project_texture
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
project_texture_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
project_texture_pixel_shader()));

        SetRasterizerState(BackFaceCulling);
    }
}

```

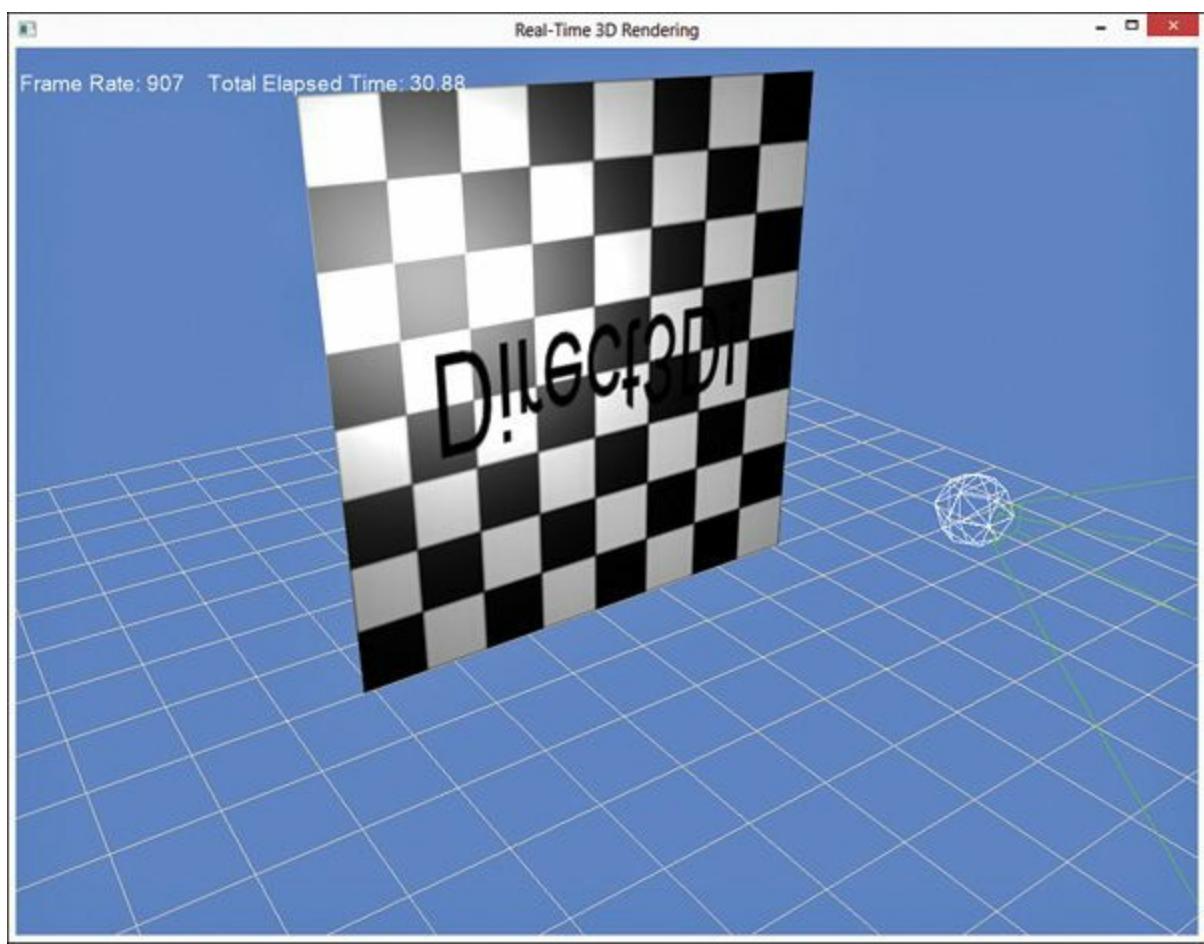
The vertex shader has only one line that is specific to projective texture mapping: the transformation of the vertex from local space into projective texture space. The resulting vector is stored in the `VS_OUTPUT.ProjectedTextureCoordinate` member for pass-through to the pixel shader. The pixel shader divides the `x` and `y` components of this vector by its `w` component to transform the position into NDC space. This step is called the **homogeneous divide** or **perspective divide**, and it is performed automatically for the position passed using the `SV_POSITION` semantic. After this final transformation, the projected texture can be sampled and applied to the final color. Note the `ProjectedTextureSampler` object that's used for sampling the projected texture. It specifies trilinear filtering and a texture addressing mode of `BORDER` with a white border color. Projective texture coordinates can extend beyond the range of `[0, 1]`, and a white border provides the multiplicative identity for any such coordinates. You aren't limited to using border addressing mode and should experiment. [Figure 19.3](#) shows the output of the shader when wrapping texture addresses.



**Figure 19.3** Output of the projective texture-mapping shader wrapping texture addresses.

## Reverse Projection

We must address a few issues with the shader in [Listing 19.1](#). The first is that the projector appears to emit from either side of the frustum to create a *reverse projection*. [Figure 19.4](#) shows this effect.



**Figure 19.4** Reverse projection.

To correct this, verify that the projected texture coordinate's w component is greater than or equal to 0. For example:

[Click here to view code image](#)

```
if (IN.ProjectedTextureCoordinate.w >= 0.0f)
{
    IN.ProjectedTextureCoordinate.xy /= 
    IN.ProjectedTextureCoordinate.w;
    float3 projectedColor = ProjectedTexture.
    Sample(ProjectedTextureSampler,
    IN.ProjectedTextureCoordinate.xy).rgb;

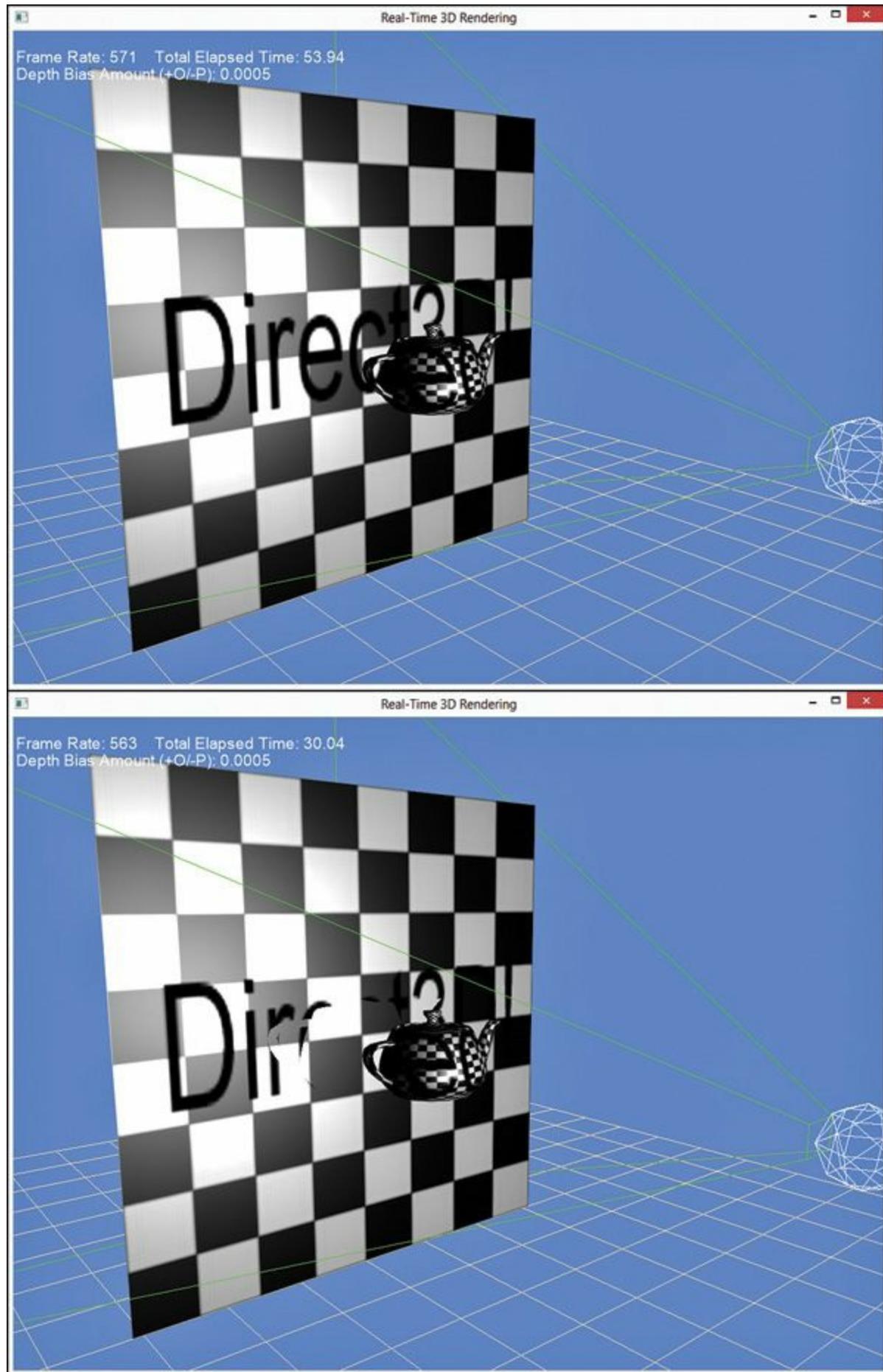
    OUT.rgb *= projectedColor;
}
```

In the demo on the companion website, this is implemented in a separate pixel shader and technique. In this way, you need not branch if reverse projection isn't a concern.

## Occlusion

A second problem is that the shader in [Listing 19.1](#) doesn't address objects that occlude other objects (or themselves). Consider the images in [Figure 19.5](#). Both images include a teapot placed in front of the plane, and each object is rendered with the same projected texture. In the top image, the texture is incorrectly projected onto the plane for areas that would otherwise be occluded by the teapot. In the

bottom image, a depth map has been created for the scene and used for occlusion testing. Such occlusion testing applies not only to the plane, but also to the teapot. For instance, if the teapot was rotated so that the handle was occluded by the pot (with respect to the projector), the handle would not (incorrectly) receive the projected texture.



**Figure 19.5** Projecting a texture without (top) and with (bottom) occlusion testing.

## Creating a Depth Map

You create a depth map (a texture that stores distances from the projector or light source, instead of colors) using the render-to-target feature discussed in the last chapter. [Listing 19.2](#) presents the declaration of the `DepthMap` class, which encapsulates the functionality required for building a depth map.

### **Listing 19.2** Declaration of the `DepthMap` Class

[Click here to view code image](#)

---

```
class DepthMap : public RenderTarget
{
    RTTI_DECLARATIONS(DepthMap, RenderTarget)

public:
    DepthMap(Game& game, UINT width, UINT height);
    ~DepthMap();

    ID3D11ShaderResourceView* OutputTexture() const;
    ID3D11DepthStencilView* DepthStencilView() const;

    virtual void Begin() override;
    virtual void End() override;

private:
    DepthMap();
    DepthMap(const DepthMap& rhs);
    DepthMap& operator=(const DepthMap& rhs);

    Game* mGame;
    ID3D11DepthStencilView* mDepthStencilView;
    ID3D11ShaderResourceView* mOutputTexture;
    D3D11_VIEWPORT mViewPort;
};
```

---

The `DepthMap` class contains members and accessors for a depth-stencil view and a shader resource view, but not for a render target view. Unlike the render-to-target work from the last chapter, you need not create a depth map through a pixel shader/render target view. Your depth map can be generated through the vertex shader with a depth-stencil view, and you can disable the pixel shader stage (which has a positive performance impact) by setting the render target view to NULL. The `DepthMap` class also contains a member for a viewport, which has the same dimensions as the depth map. This enables you to create a depth map that is smaller than your game's resolution, thereby reducing memory overhead and improving performance. [Listing 19.3](#) presents the implementation of the `DepthMap` class.

## Listing 19.3 Implementation of the DepthMap Class

[Click here to view code image](#)

```
#include "DepthMap.h"
#include "Game.h"
#include "GameException.h"

namespace Library
{
    RTTI_DEFINITIONS(DepthMap)

    DepthMap::DepthMap(Game& game, UINT width, UINT height)
        : RenderTarget(), mGame(&game),
        mDepthStencilView(nullptr),
        mOutputTexture(nullptr), mViewport()
    {
        D3D11_TEXTURE2D_DESC textureDesc;
        ZeroMemory(&textureDesc, sizeof(textureDesc));
        textureDesc.Width = width;
        textureDesc.Height = height;
        textureDesc.MipLevels = 1;
        textureDesc.ArraySize = 1;
        textureDesc.Format = DXGI_FORMAT_R24G8_TYPELESS;
        textureDesc.SampleDesc.Count = 1;
        textureDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL |
        D3D11_BIND_SHADER_RESOURCE;

        HRESULT hr;
        ID3D11Texture2D* texture = nullptr;
        if (FAILED(hr = game.Direct3DDevice() ->CreateTexture2D(
        (&textureDesc, nullptr, &texture)))
        {
            throw GameException("IDXGIDevice::CreateTexture2D() failed.", hr);
        }

        D3D11_SHADER_RESOURCE_VIEW_DESC resourceViewDesc;
        ZeroMemory(&resourceViewDesc, sizeof(resourceViewDesc));
        resourceViewDesc.Format =
        DXGI_FORMAT_R24_UNORM_X8_TYPELESS;
        resourceViewDesc.ViewDimension =
        D3D_SRV_DIMENSION_TEXTURE2D;
        resourceViewDesc.Texture2D.MipLevels = 1;

        if (FAILED(hr = game.Direct3DDevice()
        ->CreateShaderResourceView(texture, &resourceViewDesc,
        
```

```

&mOutputTexture)))
{
    ReleaseObject(texture);
    throw
GameException("IDXGIDevice::CreateShaderResource
View() failed.", hr);
}

D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc;
ZeroMemory(&depthStencilViewDesc, sizeof
(depthStencilViewDesc));
    depthStencilViewDesc.Format =
DXGI_FORMAT_D24_UNORM_S8_UINT;
    depthStencilViewDesc.ViewDimension =
D3D11_DSV_DIMENSION_TEXTURE2D;
    depthStencilViewDesc.Texture2D.MipSlice = 0;

    if (FAILED(hr = game.Direct3DDevice()-
>CreateDepthStencilView
(texture, &depthStencilViewDesc, &mDepthStencilView)))
{
    ReleaseObject(texture);
    throw
GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
}

ReleaseObject(texture);

mViewport.TopLeftX = 0.0f;
mViewport.TopLeftY = 0.0f;
mViewport.Width = static_cast<float>(width);
mViewport.Height = static_cast<float>(height);
mViewport.MinDepth = 0.0f;
mViewport.MaxDepth = 1.0f;
}

DepthMap::~DepthMap()
{
    ReleaseObject(mOutputTexture);
    ReleaseObject(mDepthStencilView);
}

ID3D11ShaderResourceView* DepthMap::OutputTexture() const
{
    return mOutputTexture;
}

```

```

ID3D11DepthStencilView* DepthMap::DepthStencilView() const
{
    return mDepthStencilView;
}

void DepthMap::Begin()
{
    static ID3D11RenderTargetView* nullRenderTargetView =
nullptr;
    RenderTarget::Begin(mGame->Direct3DDeviceContext(), 1,
&nullRenderTargetView, mDepthStencilView, mViewport);
}

void DepthMap::End()
{
    RenderTarget::End(mGame->Direct3DDeviceContext());
}

```

---

Most of this implementation resides within the `DepthMap` constructor. First, a 2D texture is created to store the actual depth map data. Note the `DXGI_FORMAT_R24G8_TYPELESS` format specified for the texture. This indicates that the texture will be split into 24-bit and 8-bit components. The `TYPELESS` identifier denotes that the depth-stencil and the shader resource views will treat the texture differently. The depth-stencil view is bound to the output-merger stage, and the shader resource view is supplied as input into a shader (the projective texture mapping shader, for example). After the shader resource and depth-stencil views are created, the texture object can be released (because the two views maintain references to the object). Finally, a viewport is created.

The `DepthMap::Begin()` and `DepthMap::End()` methods perform the same function as their counterparts in the `FullScreenRenderTarget` class from the last chapter: They bind the depth-stencil view to the output-merger stage and restore the previous render targets, respectively. However, these methods now invoke members from a base `RenderTarget` class that implements a render target stack. `RenderTarget::Begin()` pushes the specified render targets onto the stack before binding them to the output-merger stage, and vice versa for `RenderTarget::End()`. This allows nested render-targets. [Listings 19.4](#) and [19.5](#) present the declaration and implementation of the `RenderTarget` base class.

## **Listing 19.4** Declaration of the `RenderTarget` Class

[Click here to view code image](#)

---

```

class RenderTarget : public RTTI
{
    RTTI_DECLARATIONS(RenderTarget, RTTI)

public:

```

```

RenderTarget();
virtual ~RenderTarget();

virtual void Begin() = 0;
virtual void End() = 0;

protected:
    typedef struct _RenderTargetData
{
    UINT ViewCount;
    ID3D11RenderTargetView** RenderTargetViews;
    ID3D11DepthStencilView* DepthStencilView;
    D3D11_VIEWPORT Viewport;

    _RenderTargetData(UINT viewCount,
ID3D11RenderTargetView** renderTargetViews,
ID3D11DepthStencilView* depthStencilView, const D3D11_VIEWPORT&
viewport)
        : ViewCount(viewCount),
    RenderTargetViews(renderTarget
Views), DepthStencilView(depthStencilView), Viewport(viewport) {
}
    } RenderTargetData;

    void Begin(ID3D11DeviceContext* deviceContext, UINT
viewCount,
ID3D11RenderTargetView** renderTargetViews,
ID3D11DepthStencilView*
depthStencilView, const D3D11_VIEWPORT& viewport);
    void End(ID3D11DeviceContext* deviceContext);

private:
    RenderTarget(const RenderTarget& rhs);
    RenderTarget& operator=(const RenderTarget& rhs);

    static std::stack<RenderTargetData> sRenderTargetStack;
};

```

---

## Listing 19.5 Implementation of the RenderTarget Class

[Click here to view code image](#)

---

```

#include "RenderTarget.h"
#include "Game.h"

namespace Library

```

```
{  
    RTTI_DEFINITIONS(RenderTarget)  
  
    std::stack<RenderTarget::RenderTargetData> RenderTarget::  
sRenderTargetStack;  
  
    RenderTarget::RenderTarget()  
    {  
    }  
  
    RenderTarget::~RenderTarget()  
    {  
    }  
  
    void RenderTarget::Begin(ID3D11DeviceContext* deviceContext,  
    UINT viewCount, ID3D11RenderTargetView** renderTargetViews,  
    ID3D11DepthStencilView* depthStencilView, const D3D11_VIEWPORT&  
viewport)  
    {  
        sRenderTargetStack.push(RenderTargetData(viewCount,  
renderTargetViews, depthStencilView, viewport));  
        deviceContext->OMSetRenderTargets(viewCount,  
renderTargetViews,  
depthStencilView);  
        deviceContext->RSSetViewports(1, &viewport);  
    }  
  
    void RenderTarget::End(ID3D11DeviceContext* deviceContext)  
    {  
        sRenderTargetStack.pop();  
  
        if (sRenderTargetStack.size() > 0)  
        {  
            RenderTargetData renderTargetData =  
sRenderTargetStack.  
top();  
            deviceContext->OMSetRenderTargets(renderTargetData.  
ViewCount, renderTargetData.RenderTargetViews, renderTargetData.  
DepthStencilView);  
            deviceContext->RSSetViewports(1, &renderTargetData.  
Viewport);  
        }  
        else  
        {  
            static ID3D11RenderTargetView* nullRenderTargetView  
=
```

```

nullptr;
    deviceContext->OMSetRenderTargets(1,
&nullRenderTargetView,
nullptr);
}
}
}

```

---

Note that, with the introduction of the `RenderTarget` class, the base Game class becomes a `RenderTarget`. You can find the updated source code on the companion website.

Using the `DepthMap` class is simple: You merely bind the depth map to the output-merger stage and render the scene (or a portion of the scene) from the perspective of the projector. For example:

[Click here to view code image](#)

```

mDepthMap->Begin();

ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_
TRIANGLELIST);

direct3DDeviceContext->ClearDepthStencilView(mDepthMap
->DepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL,
1.0f, 0);
Pass* pass = mDepthMapMaterial->CurrentTechnique()-
>Passes().at(0);
ID3D11InputLayout* inputLayout = mDepthMapMaterial-
>InputLayouts().
at(pass);
direct3DDeviceContext->IASetInputLayout(inputLayout);

UINT stride = mDepthMapMaterial->VertexSize();
UINT offset = 0;
direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mModelPositionVertexBuffer, &stride, &offset);
direct3DDeviceContext->IASetIndexBuffer(mModelIndexBuffer,
DXGI_FORMAT
R32_UINT, 0);

XMMATRIX modelWorldMatrix = XMLoadFloat4x4(&mModelWorldMatrix);
mDepthMapMaterial->WorldLightViewProjection() <<
modelWorldMatrix *
mProjector->ViewMatrix() * mProjector->ProjectionMatrix();

pass->Apply(0, direct3DDeviceContext);
direct3DDeviceContext->DrawIndexed(mModelIndexCount, 0, 0);

```

```
mDepthMap->End();
```

The DepthMapMaterial class, referenced in the previous code snippet, supports the Depth Map.fx shader in [Listing 19.6](#).

## **Listing 19.6** The DepthMap.fx Shader

[Click here to view code image](#)

---

```
cbuffer CBufferPerObject
{
    float4x4 WorldLightViewProjection;
}

float4 create_depthmap_vertex_shader(float4 ObjectPosition : POSITION)
: SV_Position
{
    return mul(ObjectPosition, WorldLightViewProjection);
}

technique11 create_depthmap
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
create_depthmap_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(NULL);
    }
}
```

---

This effect has only a single variable, WorldLightViewProjection, and accepts just an ObjectPosition as input to the vertex shader. Also notice how the create\_depthmap technique sets the pixel shader to NULL.

## **A Projective Texture-Mapping Shader with Occlusion Testing**

With the capability to create a depth map, you can now update the projective texture mapping shader to support occlusion testing. [Listing 19.7](#) presents the updated shader, abbreviated for brevity.

## **Listing 19.7** An Updated Projective Texture-Mapping Shader with Occlusion Testing

[Click here to view code image](#)

---

```
cbuffer CBufferPerFrame
{
```

```

/* ... */
float DepthBias = 0.005;
}

Texture2D ColorTexture;
Texture2D ProjectedTexture;
Texture2D DepthMap;

SamplerState DepthMapSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
};

float4 project_texture_w_depthmap_pixel_shader(VS_OUTPUT IN) :
SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;
}

```

```

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

if (IN.ProjectedTextureCoordinate.w >= 0.0f)
{
    IN.ProjectedTextureCoordinate.xyz /= IN.ProjectedTexture
Coordinate.w;
    float pixelDepth = IN.ProjectedTextureCoordinate.z;
    float sampledDepth = DepthMap.Sample(DepthMapSampler,
IN.ProjectedTextureCoordinate.xy).x + DepthBias;

    float3 projectedColor = (pixelDepth > sampledDepth ?
ColorWhite : ProjectedTexture.Sample(ProjectedTextureSampler,
IN.ProjectedTextureCoordinate.xy).rgb);
    OUT.rgb *= projectedColor;
}

return OUT;
}

***** Techniques *****

technique11 project_texture_w_depthmap
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
project_texture_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
project_texture_w_depthmap_pixel_shader()));

        SetRasterizerState(BackFaceCulling);
    }
}

```

The shader now includes variables for the `DepthMap` and a `DepthBias` (described shortly), along with a `DepthMapSampler`. Unlike the projected texture (which uses trilinear filtering to reduce rendering artifacts), the depth map sampler uses point filtering to find the nearest depth value to the requested UV. A texel in the depth map represents a specific location in the scene. Although you can sample multiple depths and interpolate the notion of which of them is occluding an object and which is not (which you do in the next section), you should not interpolate the depth itself.

Examine the pixel shader and notice that the homogeneous divide is performed on the x, y, and z components of the `ProjectedTextureCoordinate` input member. The z component represents the pixel depth of the rendered object from the perspective of the projector. This value is compared against the sampled depth to determine whether the projected texture should be applied to the final

color. If the pixel depth is greater than the sampled depth, an object is occluding the pixel from the perspective of the projector. Notice that the `DepthBias` variable modifies the sampled depth. This is a first peek at a problem you'll encounter when implementing shadow mapping: **shadow acne**. You can see this in [Figure 19.6](#), where the shader is supplied with a `DepthBias` of 0. ([Figure 19.6](#) also shows the depth map in the lower-left corner.)



**Figure 19.6** Output of the projective texture mapping shader, using a depth map and no depth bias.

Shadow acne is an error with self-shadowing that occurs because the shadow map is limited in its resolution and its depths are quantized. When comparing the actual depth against the sampled depth, the results vary. Thus, some of the pixels indicate that they should receive the projected texture, and some do not. Precision errors can also cause shadow acne. A simple fix is to bias the depth by some fixed amount. Or instead of using a fixed-depth bias, you might employ **slope-scaled depth biasing**, a technique presented in the discussion of shadow mapping.

Another approach to eliminating shadow acne is to cull front-facing geometry when rendering the depth map.

## Shadow Mapping

You now have all the pieces necessary to implement shadow mapping. Indeed, shadow mapping follows practically the same process as projective texture mapping with occlusion testing. First, you render the scene to a depth map from the perspective of a light. Next, you render the scene from the perspective of the camera, using the depth map as input to determine whether an object is “in shadow.” [Listing 19.8](#) presents a first pass at a shadow-mapping shader.

### [Listing 19.8](#) An initial shadow-mapping shader

[Click here to view code image](#)

```
#include "include\\Common.fxh"

***** Resources *****/
static const float4 ColorWhite = { 1, 1, 1, 1 };
static const float3 ColorBlack = { 0, 0, 0 };
static const float DepthBias = 0.005;

cbuffer CBufferPerFrame
{
    float4 AmbientColor = { 1.0f, 1.0f, 1.0f, 0.0f };
    float4 LightColor = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 LightPosition = { 0.0f, 0.0f, 0.0f };
    float LightRadius = 10.0f;
    float3 CameraPosition;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
    float4 SpecularColor : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
    float SpecularPower : SPECULARPOWER = 25.0f;
    float4x4 ProjectiveTextureMatrix;
}

Texture2D ColorTexture;
Texture2D ShadowMap;

SamplerState ShadowMapSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
};

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

```
RasterizerState BackFaceCulling
{
    CullMode = BACK;
};

/****** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
    float4 ShadowTextureCoordinate : TEXCOORD3;
};

/****** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.TextureCoordinate = IN.TextureCoordinate;
    OUT.Normal = normalize(mul(float4(IN.Normal, 0),
World).xyz);

    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));

    OUT.ShadowTextureCoordinate = mul(IN.ObjectPosition,
ProjectiveTextureMatrix);

    return OUT;
}

/****** Pixel Shader *****/
```

```

float4 shadow_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    if (IN.ShadowTextureCoordinate.w >= 0.0f)
    {
        IN.ShadowTextureCoordinate.xyz /=
IN.ShadowTextureCoordinate.w;
        float pixelDepth = IN.ShadowTextureCoordinate.z;
        float sampledDepth = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy).x + DepthBias;

            // Shadow applied in a boolean fashion -- either in
shadow or not
        float3 shadow = (pixelDepth > sampledDepth ? ColorBlack
:
ColorWhite.rgb);
        diffuse *= shadow;
        specular *= shadow;
    }
}

```

```

        OUT.rgb = ambient + diffuse + specular;
        OUT.a = 1.0f;

    return OUT;
}

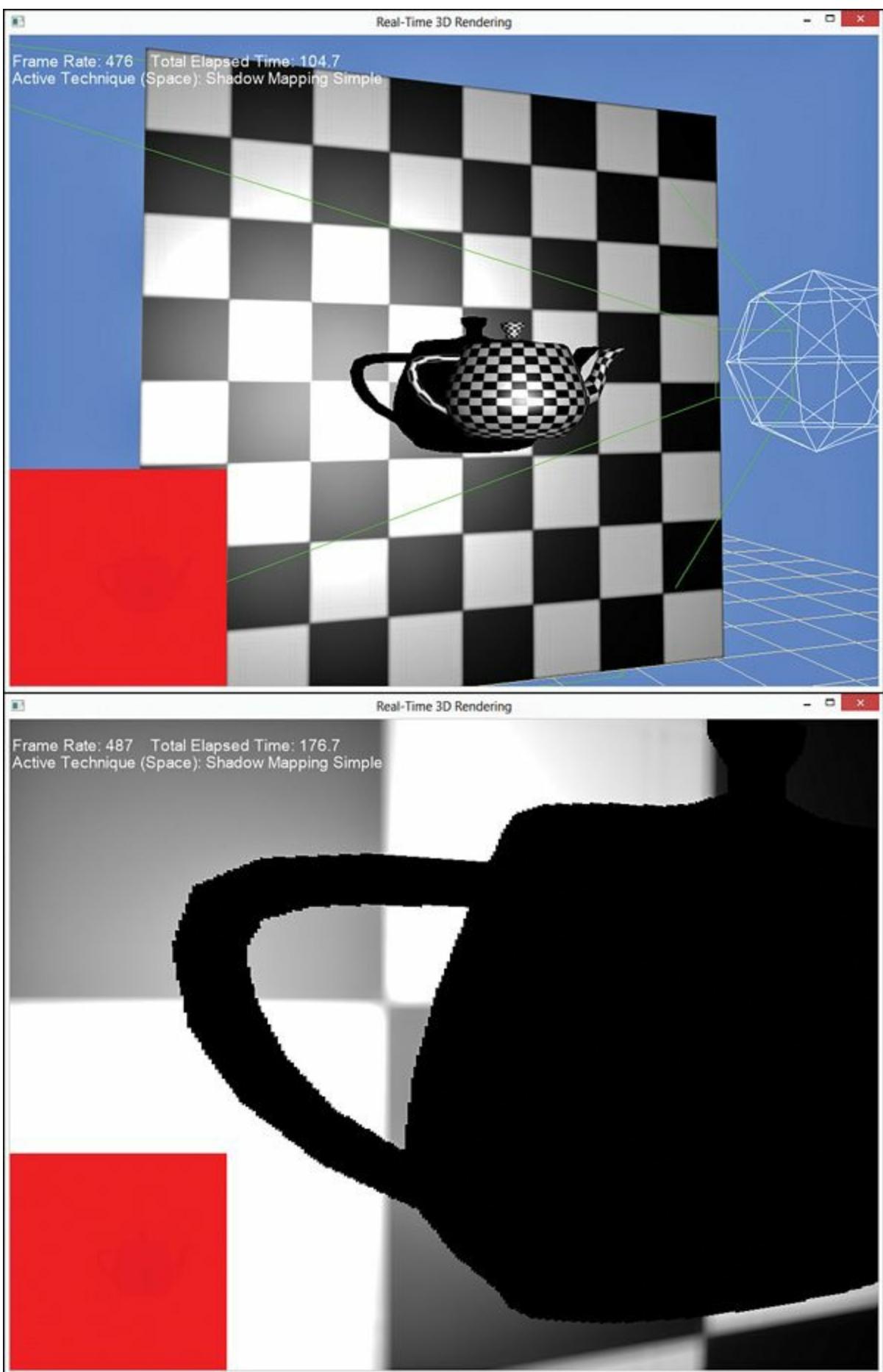
//***** Techniques *****/
technique11 shadow_mapping
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
shadow_pixel_shader()));

        SetRasterizerState(BackFaceCulling);
    }
}

```

---

This shader is very similar to the projective texture-mapping shader with occlusion testing. The depth map variable is now named `ShadowMap`, along with its `ShadowMapSampler`. But you see the real modifications in the pixel shader. The shadow map is sampled and biased and compared against the pixel depth. If the pixel depth is greater than the sampled depth, then it is in shadow and its diffuse and specular components are modulated by a shadow color (black, in this example), although not the ambient term. Notice that this is an all-or-nothing technique; the pixel is either completely in shadow or not. This produces hard-edged shadows, as in [Figure 19.7](#).



**Figure 19.7** Output of the shadow mapping shader (top), zoomed in (bottom) to demonstrate hard shadow edges.

## Percentage Closer Filtering

You can soften the edges of the shadow by sampling the depth map multiple times to determine how much of the sampled area is in shadow. This technique is known as **percentage closer filtering** (PCF). Traditionally, PCF samples a  $2 \times 2$  grid surrounding the requested UV. Each sample is tested against the object's pixel depth to determine whether the sample is in shadow, and the results are interpolated. If three samples are in shadow but one is not, the pixel can be described as being 75 percent in shadow. The UVs are computed according to the depth map's size:

$$TexelSize = 1 / ShadowMapSize$$

$$Sample_1 = (u, v)$$

$$Sample_2 = (u + TexelSize.x, v)$$

$$Sample_3 = (u, v + TexelSize.y)$$

$$Sample_4 = (u + TexelSize.x, v + TexelSize.y)$$

[Listing 19.9](#) presents the pixel shader for shadow mapping with PCF.

### **Listing 19.9** A Shadow-Mapping Shader with Manual PCF

[Click here to view code image](#)

---

```
cbuffer CBufferPerFrame
{
    /* ... */
    float2 ShadowMapSize = { 1024.0f, 1024.0f };
}

float4 shadow_manual_pcf_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
```

```

IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

        float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
        float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
        float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    if (IN.ShadowTextureCoordinate.w >= 0.0f)
    {
        IN.ShadowTextureCoordinate.xyz /=

IN.ShadowTextureCoordinate.w;

        float2 texelSize = 1.0f / ShadowMapSize;
        float sampledDepth1 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy).x + DepthBias;
        float sampledDepth2 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy + float2(texelSize.x, 0)).x +
DepthBias;
        float sampledDepth3 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy + float2(0, texelSize.y)).x +
DepthBias;
        float sampledDepth4 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy + float2(texelSize.x,
texelSize.y)).x +
DepthBias;

        float pixelDepth = IN.ShadowTextureCoordinate.z;
        float shadowFactor1 = (pixelDepth > sampledDepth1 ? 0.0f
:
1.0f);
        float shadowFactor2 = (pixelDepth > sampledDepth2 ? 0.0f
:
1.0f);
        float shadowFactor3 = (pixelDepth > sampledDepth3 ? 0.0f
:
1.0f);
        float shadowFactor4 = (pixelDepth > sampledDepth4 ? 0.0f
:
1.0f);

        float2 lerpValues = frac(IN.ShadowTextureCoordinate.xy *

```

```

ShadowMapSize);

    float shadow = lerp(lerp(shadowFactor1, shadowFactor2,
lerpValues.x), lerp(shadowFactor3, shadowFactor4, lerpValues.x),
lerpValues.y);
        diffuse *= shadow;
        specular *= shadow;
}

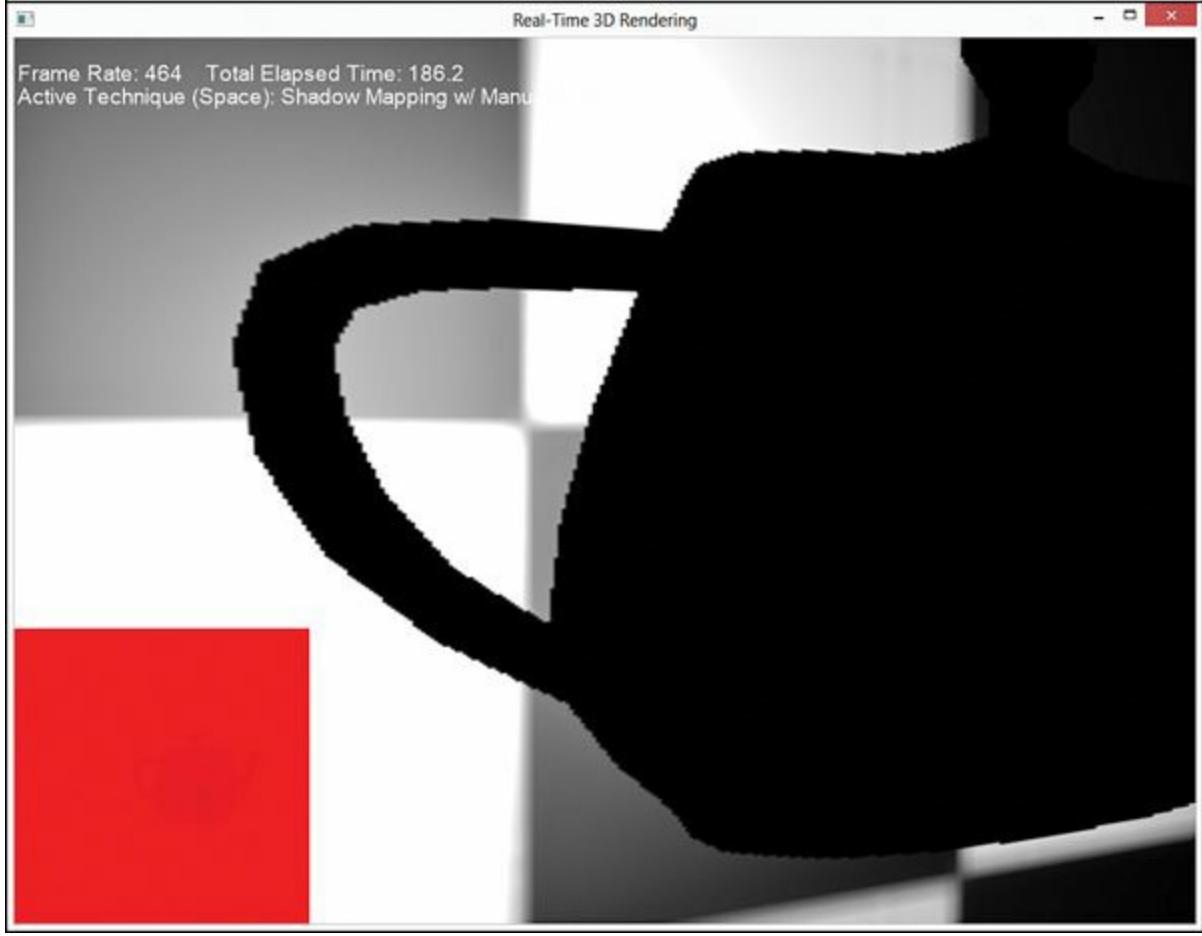
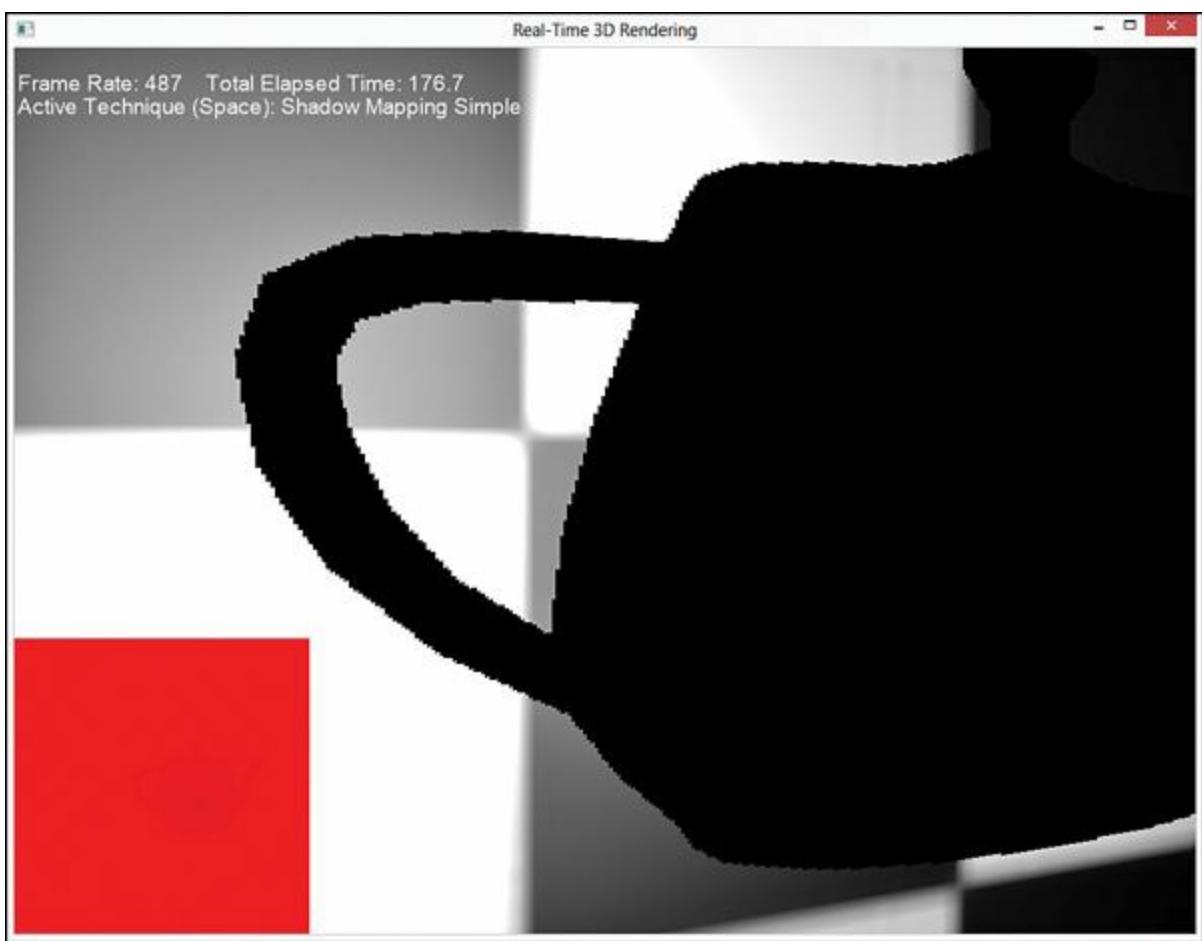
OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}

```

---

After sampling and biasing the four depths, their “in-shadow factors” are interpolated using the HLSL `lerp()` function. The interpolation value (how much should be pulled from each source) is determined with the HLSL `frac()` function, which yields the fractional part of a floating-point number. [Figure 19.8](#) shows the results of PCF compared with the original shadow mapping shader.



**Figure 19.8** Shadow edges without (top) and with (bottom) percentage closer filtering.

Sampling the depth map repeatedly is expensive, and the more samples you take, the more expensive the shader. Thankfully, Direct3D 11 intrinsically supports PCF through the texture object's

`SampleCmp()` function, which improves performance over manual PCF. Thus, the previous shader can be rewritten to match [Listing 19.10](#).

## **Listing 19.10 A Shadow-Mapping Shader with Intrinsic PCF**

[Click here to view code image](#)

---

```
SamplerComparisonState PcfShadowMapSampler
{
    Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
    ComparisonFunc = LESS_EQUAL;
};

float4 shadow_pcf_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    IN.ShadowTextureCoordinate.xyz /=
```

```

IN.ShadowTextureCoordinate.w;
    float pixelDepth = IN.ShadowTextureCoordinate.z;

    float shadow =
ShadowMap.SampleCmpLevelZero(PcfShadowMapSampler,
IN.ShadowTextureCoordinate.xy, pixelDepth).x;
    diffuse *= shadow;
    specular *= shadow;

    OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}

```

---

Gone from this shader are the manual offset calculations, in-shadow comparisons, and linear interpolations. The `SampleCmpLevelZero()` function handles all these. This function samples mip-level zero (the only mipmap you create for the depth map) with  $2 \times 2$  PCF filtering and compares the sampled values against the function's third argument (the pixel depth). This function requires a `SamplerComparisonState` object instead of a normal `SamplerObject`, with a filter of `COMPARISION_MIN_MAG_LINEAR_MIP_POINT`.

The output is the same, although you must supply the depth bias through a `RasterizerState` object when you create the depth map. This leads us to the final topic in this chapter, **slope-scaled depth biasing**.

## Slope-Scaled Depth Biasing

To provide a fixed depth bias for intrinsic PCF filtering, modify your `DepthMap.fx` shader to match [Listing 19.11](#).

### Listing 19.11 Setting the Bias When Creating the Depth Map

[Click here to view code image](#)

---

```

RasterizerState DepthBias
{
    DepthBias = 84000;
};

float4 create_depthmap_vertex_shader(float4 ObjectPosition :
POSITION)
: SV_Position
{
    return mul(ObjectPosition, WorldLightViewProjection);
}

technique11 create_depthmap_w_bias

```

```

}

pass p0
{
    SetVertexShader(CompileShader(vs_5_0,
create_depthmap_vertex_shader()));
    SetGeometryShader(NULL);
    SetPixelShader(NULL);

    SetRasterizerState(DepthBias);
}
}

```

---

The `DepthBias` member of the `RasterizerState` object provides a fixed offset for values as they are written to the depth buffer. This number is used to calculate the final bias as follows:

$$Bias_{Final} = DepthBias * r$$

where,  $r$  is the smallest value that can be represented by the floating-point format of the depth buffer (for a 24-bit depth map  $r = 1/2^{24}$ ). Thus, a fixed-depth bias of 0.005 (the default value in the previous iterations of the shadow mapping shader) can be calculated as  $0.005 \approx 84000 * 1/2^{24}$ . The problem with a fixed-depth bias is that your geometry isn't uniform (with respect to its orientation to the light source). Specifically, as the slope of the triangle increases, so should its bias. Direct3D hardware can intrinsically measure the slope of the triangle and use the

`RasterizerState.SlopeScaledDepthBias` property to calculate the bias dynamically. The `SlopeScaledDepthBias` is modulated by the `MaxDepthSlope`, whose value is the maximum of the horizontal and vertical slopes of the depth value at a specific location. Thus, the final bias calculation can be redefined as:

$$Bias_{Final} = DepthBias * r + SlopeScaledDepthBias * MaxDepthSlope$$

Because this value is calculated dynamically, you might want to clamp the bias through the `RasterizerState.DepthBiasClamp` property. This prevents very large depth values when a polygon is viewed at a sharp angle.

### Note

This bias calculation is used when the depth buffer has a UNORM format. Visit the MSDN Direct3D documentation for the depth bias calculation when the depth buffer has a floating-point format.

You might be tempted to skip slope-scaled depth biasing and just choose a large enough fixed bias to accommodate most of your geometry. However, an overly large bias leads to an effect called **peter panning**, where the shadow appears disconnected from the associated object (the name comes from the children's book character whose shadow became detached). These values are scene dependent, and you need to experiment to find optimal values. The demo available on the companion website

enables you to change these values and visualize the results interactively.

## Summary

In this chapter, you explored shadow mapping, a common technique for producing shadows. As an intermediate step, you developed shaders for projective texture mapping, a technique for projecting a 2D texture onto arbitrary geometry. You learned about creating depth maps and occlusion testing, and you applied that work to implement a shadow mapping shader. You also examined various issues with shadow mapping, including *shadow acne*, *hard edges*, and *peter panning*, and you discovered ways to address these problems.

Shadow mapping isn't the only approach for producing shadows, and more techniques can help with overcoming shadow artifacts. In particular, you might want to investigate **cascaded shadow maps** and **variance shadow maps**. Microsoft has a good article on the subject titled "Common Techniques to Improve Shadow Depth Maps," available online at MSDN. A link to this article is available on the book's companion website.

## Exercises

1. Experiment with all the effects and demo applications from this chapter. Vary the shader inputs, and observe the results.
2. Vary the size of your depth/shadow map, and observe the results. The `ShadowMappingDemo` application (available on the companion website) uses a  $1024 \times 1024$  depth map. What happens when you go to  $2048 \times 2048$  or  $4096 \times 4096$ ? What about  $256 \times 256$ ?
3. While perspective projection (what we've used for the projector in this chapter) is common for simulating shadows for point and spotlights, directional light shadows are typically implemented through orthographic projection. Implement an orthographic projector and integrate it with your shadow-mapping shader.

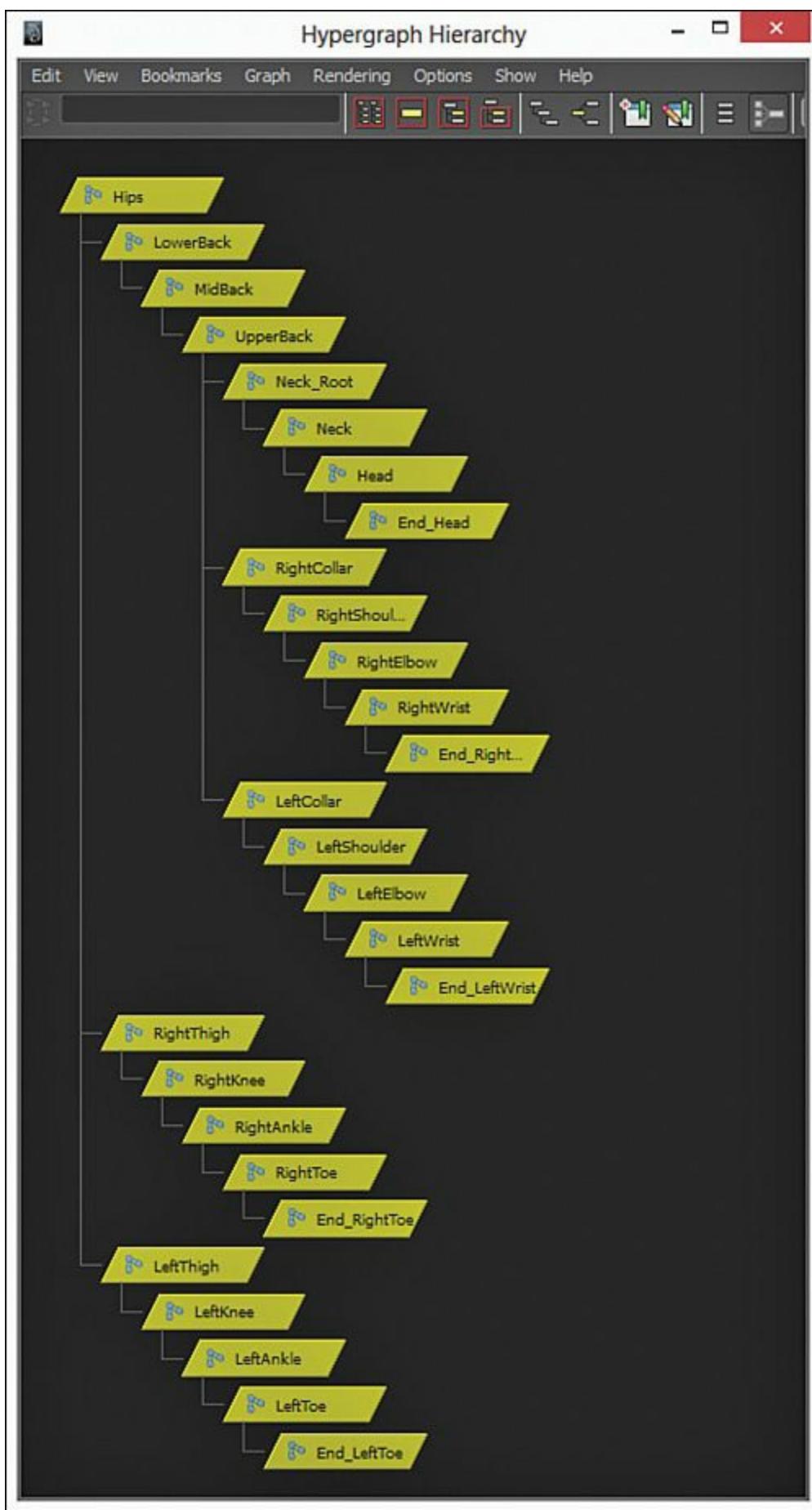
# Chapter 20. Skeletal Animation

Skeletal animation gets its name from the hierarchical set of interconnected transforms (bones) associated with a model’s mesh (the skeleton’s skin). When these transforms are modified, over time, the mesh is animated. In this chapter, you explore skeletal animation and develop the systems to animate your models.

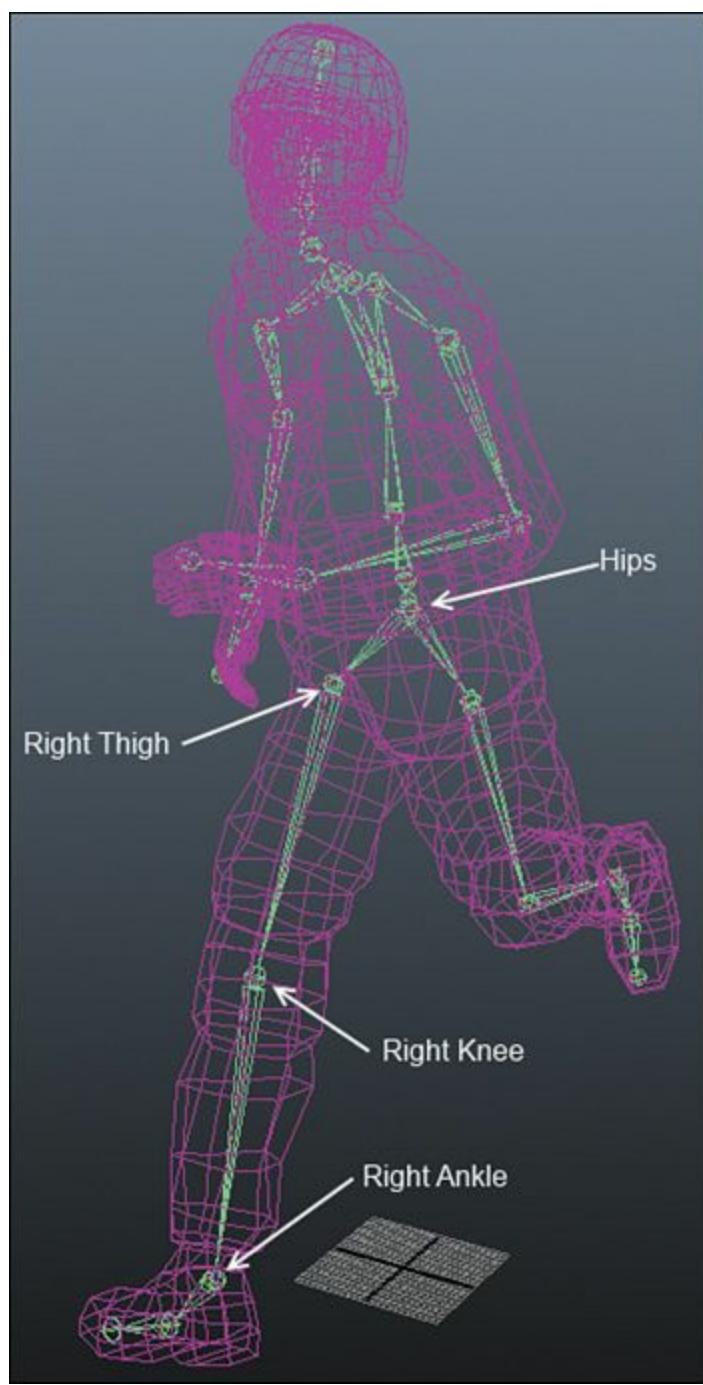
## Hierarchical Transformations

In most of the demonstrations thus far, you have applied transformations—scale, rotation, and translation—to orient your objects in world space. These transformations modify the position of each vertex in an associated model. That’s the basis of animation: If you modify these transformations over time, your object will animate. However, most people wouldn’t consider an object moving or rotating as a whole to be true animation. Animation is typically defined as subelements of an object changing position with respect to the entire model.

Consider the example of a human model with a running animation. The model itself could be constructed as just a single mesh, but to make the model run, the vertices in the arms, hips, thighs, lower legs, and feet each need to be transformed separately. However, just as with a real human body, the arms are connected to shoulders, thighs are connected to hips, lower legs are connected to thighs, and so on. And when one part of the body moves, its associated parts move as well. Thus, these transformations must be applied according to a hierarchy, with the final positions of the vertices associated with each *node* determined by the node’s **local transformation**—and the transformations of its entire ancestry. This hierarchy is commonly referred to as a **skeleton**, and each node in the hierarchy is a **bone**. A bone is just a transformation, and that transformation impacts the vertices associated with the bone and all the vertices down the hierarchy. [Figure 20.1](#) shows an example of the skeleton of a model of a human soldier. At the root of the skeleton is a bone labeled *Hips* with child bones for *LowerBack*, *RightThigh*, and *LeftThigh*; these bones have children of their own, and so on down the hierarchy. [Figure 20.2](#) shows this skeleton surrounded by a mesh, with a few of the bones labeled. Building the skeleton is a process known as **rigging** and is performed by an artist.



**Figure 20.1** The skeletal hierarchy of a human soldier. (*Animated model provided by Brian Salisbury, Florida Interactive Entertainment Academy.*)



**Figure 20.2** The human soldier skeleton surrounded by a mesh. (*Animated model provided by Brian Salisbury, Florida Interactive Entertainment Academy.*)

## Skinning

A skeleton is mapped to a mesh in a process known as **skinning**. This task is performed by an artist and involves associating vertices to specific bones. A vertex can be associated with more than one bone (typically up to four), with each bone influencing the vertex by a given amount (for example, at a joint). Thus, the final position of the vertex is derived from a weighted average of the transformations of the associated bones.

The term *skinning* also applies to how the vertices are transformed at runtime. **CPU skinning** indicates that the vertices are transformed on the CPU, while **GPU skinning** pushes this work onto the GPU. GPU skinning is typically much faster than CPU skinning, although it generally limits the number of bones that can be used. [Listing 20.1](#) presents a GPU skinning shader.

## Listing 20.1 The SkinnedModel.fx Shader

[Click here to view code image](#)

```
#include "include\\Common.fhx"

#define MaxBones 60

/********************* Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor = { 1.0f, 1.0f, 1.0f, 0.0f };
    float4 LightColor = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 LightPosition = { 0.0f, 0.0f, 0.0f };
    float LightRadius = 10.0f;
    float3 CameraPosition;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
    float4 SpecularColor : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
    float SpecularPower : SPECULARPOWER = 25.0f;
}

cbuffer CBufferSkinning
{
    float4x4 BoneTransforms [MaxBones];
}

Texture2D ColorTexture;

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/****************** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
```



```

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,
SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

```

```

***** Techniques *****/

```

```

technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}
```

}

---

This is just a point light shader that supports skinned models. All the skinning-specific work is performed in the vertex shader, which makes use of the new `BoneTransforms` shader variable and the `BoneIndices` and `BoneWeights` shader inputs. The `BoneTransforms` variable is an array of matrices that contains the transformation of each bone in the model's skeleton. The `BoneIndices` member of the `VS_INPUT` structure is an array of four unsigned integers that index into the `BoneTransforms` array. Each `BoneWeight` element applies the weighted average for vertices that are influenced by more than one bone.

The vertex shader first composes a `skinTransform` matrix with this code:

[Click here to view code image](#)

```
float4x4 skinTransform = (float4x4)0;
skinTransform += BoneTransforms [IN.BoneIndices.x] *
IN.BoneWeights.x;
skinTransform += BoneTransforms [IN.BoneIndices.y] *
IN.BoneWeights.y;
skinTransform += BoneTransforms [IN.BoneIndices.z] *
IN.BoneWeights.z;
skinTransform += BoneTransforms [IN.BoneIndices.w] *
IN.BoneWeights.w;
```

If a vertex isn't influenced by a bone, its weight is 0 and this thereby eliminates the bone's contribution to the `skinTransform` matrix. The CPU-side application should guarantee that the bone weights sum to 1.

Next, the vertex position is multiplied by the `skinTransform` matrix (likewise for the surface normal, tangent, and binormal, if applicable). The bone matrices are still within the object's local coordinate system, thus the normal `WorldViewProjection` matrix is applied to move the vertex from animated local space to homogenous clip space before the next shader stage.

Note that the pixel shader is identical to the original point light pixel shader. Indeed, skinned models can be supported for any of the lighting techniques we have presented in this book. All that is required are the additional bone-specific shader variables/inputs we've just discussed.

## Importing Animated Models

Before you can invoke the skinned model shader, you need to import the animation data for a particular model. This is easier said than done and requires additional plumbing. Thankfully, the Open Asset Import Library (see [Chapter 15, “Models”](#)) supports animated models, and you can employ it to at least get animation data from various file formats. However, you have more work to do to get the data into a format your rendering engine can use and to isolate the runtime code base from the Open Asset Import Library (with the idea that transforming data into runtime format is best done as part of a build-time process).

The first class to create is the `SceneNode` class (see [Listing 20.2](#)).

### Listing 20.2 Declaration of the `SceneNode` Class

[Click here to view code image](#)

---

```
class SceneNode : public RTTI
{
    RTTI_DECLARATIONS(SceneNode, RTTI)

public:
    const std::string& Name() const;
    SceneNode* Parent();
    std::vector<SceneNode*>& Children();
    const XMFLOAT4X4& Transform() const;
    XMMATRIX TransformMatrix() const;

    void SetParent(SceneNode* parent);
    void SetTransform(XMFLOAT4X4& transform);
    void SetTransform(CXMMATRIX transform);

    SceneNode(const std::string& name);
    SceneNode(const std::string& name, const XMFLOAT4X4&
transform);

protected:
    std::string mName;
    SceneNode* mParent;
    std::vector<SceneNode*> mChildren;
    XMFLOAT4X4 mTransform;

private:
    SceneNode();
    SceneNode(const SceneNode& rhs);
    SceneNode& operator=(const SceneNode& rhs);
};
```

---

The SceneNode class establishes the hierarchical transformation structure used in skeletal animation. Each SceneNode has a name, a transformation, a parent, and a collection of children. If the parent is NULL, the node represents the root of the hierarchy.

The Bone class derives from SceneNode; [Listing 20.3](#) gives its declaration.

### **Listing 20.3** Declaration of the Bone Class

[Click here to view code image](#)

---

```
class Bone : public SceneNode
{
    RTTI_DECLARATIONS(Bone, SceneNode)
```

```

public:
    UINT Index() const;
    void SetIndex(UINT index);

    const XMFLOAT4X4& OffsetTransform() const;
    XMMATRIX OffsetTransformMatrix() const;

    Bone(const std::string& name, UINT index, const XMFLOAT4X4&
offsetTransform);

private:
    Bone();
    Bone(const Bone& rhs);
    Bone& operator=(const Bone& rhs);

    UINT mIndex; // Index into the model's bone
    container
    XMFLOAT4X4 mOffsetTransform; // Transforms from mesh space
    to bone
    space
};

```

A `Bone` object represents a particular element within the model's skeleton. Because a `Bone` object is also a `SceneNode` object, it shares the same hierarchical structure. The `Bone` and `SceneNode` classes are separate to allow transformations within the hierarchy that impact the skeleton but have no vertices associated with them. This might seem odd at first, but it is actually very common. The `Bone` class extends `SceneNode` to include an index into the model's skeleton and an **offset transform**. The index is used to associate vertices to the shader's array of bone transformations (the `BoneIndices` shader input).

The offset transform moves an object from mesh space to bone space. This transformation matrix is necessary because the vertices associated with this bone are specified in mesh space (that is, local/model space), but the bone transformations are **in bone space** (the bone's own coordinate system, relative to its parent's bone). Therefore, to manipulate a vertex by a bone transformation, you must first move it from mesh space to bone space using the offset transform.

With the `Bone` class defined, you can augment the `Model` class to contain a set of `Bones`. More specifically, the `Model` class should include the following data members:

[Click here to view code image](#)

```

std::vector<Bone*> mBones;
std::map<std::string, UINT> mBoneIndexMapping;
SceneNode* mRootNode;

```

The `mBones` data member is the collection of bones without respect to their hierarchy. The `mBoneIndexMapping` associates a bone's name with its index into the `mBones` collection. This same mapping identifies the `Bone::mIndex` member, which is the shader's lookup into the bone transformations array. These separate constructs are a side effect of using the Open Asset Import

Library and its approach to storing bone data. These members could potentially be removed if you processed the animation data at build time.

The `mRootNode` member represents the root node within the transformation hierarchy and is populated after all the bones within the model have been discovered. The Open Asset Import Library stores bone data per mesh, so the actual discovery of the bone data is done through an augmented Mesh class. In particular, the Mesh class is now responsible for finding two pieces of information: the bones themselves and the bone vertex weights. [Listing 20.4](#) shows the declaration of the `BoneVertexWeight` class.

## Listing 20.4 Declaration of the `BoneVertexWeight` Class

[Click here to view code image](#)

```
class BoneVertexWeights
{
public:
    typedef struct _VertexWeight
    {
        float Weight;
        UINT BoneIndex;

        _VertexWeight(float weight, UINT boneIndex)
            : Weight(weight), BoneIndex(boneIndex) { }
    } VertexWeight;

    const std::vector<VertexWeight>& Weights();
    void AddWeight(float weight, UINT boneIndex);
    static const UINT MaxBoneWeightsPerVertex = 4;
};

private:
    std::vector<VertexWeight> mWeights;
};
```

An instance of the `BoneVertexWeights` class is associated with every vertex within the mesh through the member `Mesh::mBoneWeights` (of type `std::vector<BoneVertexWeights>`). The `Mesh` class constructor (where the processing of the Open Asset Import Library mesh structure is performed) should be modified to include the code in [Listing 20.5](#).

## Listing 20.5 Bone Processing with the `Mesh` Class Constructor

[Click here to view code image](#)

```
if (mesh.HasBones())
```

```

{
    mBoneWeights.resize(mesh.mNumVertices);

    for (UINT i = 0; i < mesh.mNumBones; i++)
    {
        aiBone* meshBone = mesh.mBones[i];

        // Look up the bone in the model's hierarchy, or add it
if not
found.
        UINT boneIndex = 0U;
        std::string boneName = meshBone->mName.C_Str();
        auto boneMappingIterator = mModel.mBoneIndexMapping.
find(boneName);
        if (boneMappingIterator != mModel.mBoneIndexMapping.end())
        {
            boneIndex = boneMappingIterator->second;
        }
        else
        {
            boneIndex = mModel.mBones.size();
            XMATRIX offsetMatrix = XMLoadFloat4x4(&(XMFLOAT4X4
reinterpret_cast<const float*>(meshBone->mOffsetMatrix[0])));
            XMFLOAT4X4 offset;
            XMStoreFloat4x4(&offset,
XMMatrixTranspose(offsetMatrix));

            Bone* modelBone = new Bone(boneName, boneIndex,
offset);
            mModel.mBones.push_back(modelBone);
            mModel.mBoneIndexMapping[boneName] = boneIndex;
        }

        for (UINT i = 0; i < meshBone->mNumWeights; i++)
        {
            aiVertexWeight vertexWeight = meshBone->mWeights[i];
            mBoneWeights[vertexWeight.mVertexId].
AddWeight(vertexWeight.mWeight, boneIndex);
        }
    }
}

```

The Open Asset Import Library stores its bone data within an array of `aiBone` objects. An `aiBone` object stores the name of the bone, its offset transform, and its list of vertex weights. The code in [Listing 20.5](#) iterates through the list of `aiBone` objects and attempts to find each bone within the model's `mBoneIndexMapping` container. If it doesn't find a bone, it's added to the model (after

its offset matrix is transposed because the transform is stored in column-major order and is needed in row-major order). This step is necessary because the bones are stored in the model but are spread out across any number of `aiMesh` objects. Finally, the vertex weights are copied to the `Mesh::mBoneWeights` collection using the `aiVertexWeight::mVertexId` member. Most of the bone data is found within the mesh, but the skeletal hierarchy is part of the `aiScene` object and is processed within the `Model` class constructor. More specifically, the `aiScene` object has an `mRootNode` member of type `aiNode`. The `aiNode` structure is analogous to our `SceneNode` class, and the `aiScene::mRootNode` member represents the root of the transformation hierarchy. Processing these nodes is done through the recursive `Model::BuildSkeleton()` method, presented in [Listing 20.6](#).

## Listing 20.6 Processing the Skeletal Hierarchy

[Click here to view code image](#)

```
SceneNode* Model::BuildSkeleton(aiNode& node, SceneNode*& parentSceneNode)
{
    SceneNode* sceneNode = nullptr;

    auto boneMapping =
mBoneIndexMapping.find(node.mName.C_Str());
    if (boneMapping == mBoneIndexMapping.end())
    {
        sceneNode = new SceneNode(node.mName.C_Str());
    }
    else
    {
        sceneNode = mBones[boneMapping->second];
    }

    XMATRIX transform = XMLoadFloat4x4(&
(XMFLOAT4X4(reinterpret_cast
<const float*>(node.mTransformation[0]))));
    sceneNode->SetTransform(XMMatrixTranspose(transform));
    sceneNode->SetParent(parentSceneNode);

    for (UINT i = 0; i < node.mNumChildren; i++)
    {
        SceneNode* childSceneNode = BuildSkeleton(*
(node.mChildren[i]),
sceneNode);
        sceneNode->Children().push_back(childSceneNode);
    }

    return sceneNode;
}
```

}

---

This method is invoked with the `aiSceneNode::mRootNode` member and is recursively invoked for each of the node's children. Note that the hierarchy doesn't necessarily consist entirely of `Bone` objects. If the node name matches a `Bone` in the model, then the bone is used. Otherwise, the node represents a transformation that doesn't have any associated vertices but nonetheless impacts the transformation hierarchy. In that case, a `SceneNode` class is added to the hierarchy. You must not skip these nodes.

At this point, you've imported all the data required to represent the skeletal structure of the model and its skinning information. The final step is to import the animations that dictate how the skeleton is transformed over time. Animations are composed as a set of **keyframes**. A keyframe represents a moment in time and records the transformation of a bone at that moment. An animation might be created to match the game's expected frame rate. For example, a 1-second animation might contain 60 keyframes if the game is expected to run at 60 frames per second. But if an animation has fewer keyframes than your game's frame rate, you can interpolate between keyframes to produce a smooth animation.

The Open Asset Import Library stores animations within the `aiScene::mAnimations` data member, an array of `aiAnimation` objects. Each `aiAnimation` instance contains the name of the animation, its duration (in ticks), and the number of ticks per second. It also stores sets of keyframes, one for each of the bones involved in the animation. The `AnimationClip` class is an analogous data structure (which removes the public dependency on the Open Asset Import Library). [Listing 20.7](#) gives its declaration.

## Listing 20.7 Declaration of the AnimationClip Class

[Click here to view code image](#)

---

```
class AnimationClip
{
    friend class Model;

public:
    ~AnimationClip();

    const std::string& Name() const;
    float Duration() const;
    float TicksPerSecond() const;
    const std::vector<BoneAnimation*>& BoneAnimations() const;
    const std::map<Bone*, BoneAnimation*>&
    BoneAnimationsByBone()
    const;
    const UINT KeyframeCount() const;

    UINT GetTransform(float time, Bone& bone, XMFLOAT4X4& transform)
```

```

const;
    void GetTransforms(float time, std::vector<XMFLOAT4X4>&
boneTransforms) const;
    void GetTransformAtKeyframe(UINT keyframe, Bone& bone,
XMFLOAT4X4&
transform) const;
    void GetTransformsAtKeyframe(UINT keyframe,
std::vector<XMFLOAT4X4>& boneTransforms) const;
    void GetInterpolatedTransform(float time, Bone& bone,
XMFLOAT4X4&
transform) const;
    void GetInterpolatedTransforms(float time,
std::vector<XMFLOAT4X4>& boneTransforms) const;

private:
    AnimationClip(Model& model, aiAnimation& animation);

    AnimationClip();
    AnimationClip(const AnimationClip& rhs);
    AnimationClip& operator=(const AnimationClip& rhs);

    std::string mName;
    float mDuration;
    float mTicksPerSecond;
    std::vector<BoneAnimation*> mBoneAnimations;
    std::map<Bone*, BoneAnimation*> mBoneAnimationsByBone;
    UINT mKeyframeCount;
};

```

---

The sets of keyframes, one for each bone in the animation, are stored in the mBoneAnimations container. The declaration of the BoneAnimation class is presented shortly and simply contains a collection of keyframes for a given bone. This data is processed within the AnimationClip's private constructor, through the implementation in [Listing 20.8](#).

## **Listing 20.8** Processing Animation Data

[Click here to view code image](#)

---

```

AnimationClip::AnimationClip(Model& model, aiAnimation&
animation)
    : mName(animation.mName.C_Str()),
      mDuration(static_cast<float>(animation.mDuration)),
      mTicksPerSecond(static_cast<float>
(animation.mTicksPerSecond)),
      mBoneAnimations(), mBoneAnimationsByBone(),

```

```

mKeyframeCount(0)
{
    assert(animation.mNumChannels > 0);

    if (mTicksPerSecond <= 0.0f)
    {
        mTicksPerSecond = 1.0f;
    }

    for (UINT i = 0; i < animation.mNumChannels; i++)
    {
        BoneAnimation* boneAnimation = new BoneAnimation(model,
* (animation.mChannels[i]));
        mBoneAnimations.push_back(boneAnimation);

        assert(mBoneAnimationsByBone.find(&(boneAnimation-
>GetBone())))
== mBoneAnimationsByBone.end());
        mBoneAnimationsByBone [&(boneAnimation->GetBone())] =
boneAnimation;
    }

    for (BoneAnimation* boneAnimation : mBoneAnimations)
    {
        if (boneAnimation->Keyframes().size() > mKeyframeCount)
        {
            mKeyframeCount = boneAnimation->Keyframes().size();
        }
    }
}

```

The BoneAnimation objects are also referenced in the mBoneAnimationsByBone map, which facilitates quick lookup when retrieving a specific bone's animation data. Also notice the calculation of the mKeyframeCount member. This data member is provided to allow the retrieval of bone transformations (by sequence position instead of time), up to a maximum keyframe. However, each BoneAnimation instance can contain a different number of keyframes. By choosing the largest keyframe count in the set of BoneAnimation objects, you allow retrieval of all available keyframes. You clamp at the last keyframe for BoneAnimation objects with fewer keyframes than the sequence number specified.

Retrieving bone transformations is the job of the AnimationClip::GetTransform\*() and GetTransforms\*() methods. The singular versions of these methods get the transformations of specific bones. The plural versions collect the transformations of all the bones, according to the retrieval parameters. These methods relay calls to the BoneAnimation class, in [Listing 20.9](#).

## **Listing 20.9** Declaration of the BoneAnimation Class

[Click here to view code image](#)

---

```
class BoneAnimation
{
    friend class AnimationClip;

public:
    ~BoneAnimation();

    Bone& GetBone();
    const std::vector<Keyframe*> Keyframes() const;

    UINT GetTransform(float time, XMFLOAT4X4& transform) const;
    void GetTransformAtKeyframe(UINT keyframeIndex, XMFLOAT4X4& transform) const;
    void GetInterpolatedTransform(float time, XMFLOAT4X4& transform)
const;

private:
    BoneAnimation(Model& model, aiNodeAnim& nodeAnim);

    BoneAnimation();
    BoneAnimation(const BoneAnimation& rhs);
    BoneAnimation& operator=(const BoneAnimation& rhs);

    UINT FindKeyframeIndex(float time) const;

    Model* mModel;
    Bone* mBone;
    std::vector<Keyframe*> mKeyframes;
};
```

---

The keyframe data is processed within the `BoneAnimation` constructor (see [Listing 20.10](#)).

## Listing 20.10 Processing Keyframe Data

[Click here to view code image](#)

---

```
BoneAnimation::BoneAnimation(Model& model, aiNodeAnim& nodeAnim)
    : mModel(&model), mBone(nullptr), mKeyframes()
{
    UINT boneIndex = model.BoneIndexMapping().at(nodeAnim.
mNodeName.C_Str());
    mBone = model.Bones().at(boneIndex);

    assert(nodeAnim.mNumPositionKeys ==
```

```

nodeAnim.mNumRotationKeys);
    assert(nodeAnim.mNumPositionKeys ==
nodeAnim.mNumScalingKeys);

    for (UINT i = 0; i < nodeAnim.mNumPositionKeys; i++)
{
    aiVectorKey positionKey = nodeAnim.mPositionKeys[i];
    aiQuatKey rotationKey = nodeAnim.mRotationKeys[i];
    aiVectorKey scaleKey = nodeAnim.mScalingKeys[i];

    assert(positionKey.mTime == rotationKey.mTime);
    assert(positionKey.mTime == scaleKey.mTime);

    Keyframe* keyframe = new Keyframe(static
_cast<float>(positionKey.mTime), XMFLOAT3(positionKey.mValue.x,
positionKey.mValue.y, positionKey.mValue.z),
XMFLOAT4(rotationKey.
mValue.x, rotationKey.mValue.y, rotationKey.mValue.z,
rotationKey.
mValue.w), XMFLOAT3(scaleKey.mValue.x, scaleKey.mValue.y,
scaleKey.
mValue.z));
    mKeyframes.push_back(keyframe);
}
}

```

---

This code merely copies data from `aiNodeAnim` structures to `Keyframe` instances. [Listing 20.11](#) gives the declaration of the `Keyframe` class.

## **Listing 20.11 Declaration of the Keyframe Class**

[Click here to view code image](#)

---

```

class Keyframe
{
    friend class BoneAnimation;

public:
    float Time() const;
    const XMVECTOR Translation() const;
    const XMVECTOR RotationQuaternion() const;
    const XMVECTOR Scale() const;

    XMVECTOR TranslationVector() const;
    XMVECTOR RotationQuaternionVector() const;
    XMVECTOR ScaleVector() const;

```

```

XMMATRIX Transform() const;

private:
    Keyframe(float time, const XMFLOAT3& translation, const
XMFLOAT4& rotationQuaternion, const XMFLOAT3& scale);

    Keyframe();
    Keyframe(const Keyframe& rhs);
    Keyframe& operator=(const Keyframe& rhs);

    float mTime;
    XMFLOAT3 mTranslation;
    XMFLOAT4 mRotationQuaternion;
    XMFLOAT3 mScale;
};


```

---

Each keyframe consists of a translation, rotation, and scale for a specific time within the animation. The rotation is represented as a **quaternion**, a four-dimensional extension of complex numbers that make three-dimensional rotation convenient. We've been using Euler angles throughout the book to describe rotation in 3D space. Such rotations are represented as a combination of a unit vector (the axis of rotation) and an angle ( $\theta$ ). Quaternions provide an easy way to encode these axis-angle representations in four numbers, which can be applied to a position vector. Quaternions also avoid **gimbal lock**, the loss of a degree of freedom that occurs when two Euler angles become parallel.

The implementation of the `Keyframe` class is very simple: It just holds data and exposes it through public accessors. However, the `Keyframe::Transform()` method deserves a look because it's the method that combines the translation, rotation, and scale into a transformation matrix.

[Click here to view code image](#)

```

XMMATRIX Keyframe::Transform() const
{
    XMVECTOR rotationOrigin =
XMLoadFloat4(&Vector4Helper::Zero);

    return XMMatrixAffineTransformation(ScaleVector(),
rotationOrigin,
                                         RotationQuaternionVector(),
TranslationVector());
}


```

This method is employed within the `BoneAnimation` class and its `GetTransform*` () methods. You can query uninterpolated bone transformations at a specific time through `BoneAnimation::GetTransform()`. This method works by finding the latest keyframe whose time position is less than the specified time. The `BoneAnimation::GetInterpolatedTransform()` method queries the two keyframes surrounding a time position and interpolates the translation, rotation, and scale of the frames. The `BoneAnimation::GetTransformAtKeyframe()` method returns the transformation at the

specified keyframe, or clamps at the last keyframe if the index specified is larger than the bone's keyframe set (as discussed previously, not all bones necessarily have the same number of keyframes for an animation). [Listing 20.12](#) presents the implementation for each of these methods.

## Listing 20.12 Bone Transformation Retrieval

[Click here to view code image](#)

---

```
UINT BoneAnimation::GetTransform(float time, XMFLOAT4X4&
transform)
const
{
    UINT keyframeIndex = FindKeyframeIndex(time);
    Keyframe* keyframe = mKeyframes[keyframeIndex];

    XMStoreFloat4x4(&transform, keyframe->Transform());

    return keyframeIndex;
}

void BoneAnimation::GetTransformAtKeyframe(UINT keyframeIndex,
XMFLOAT4X4& transform) const
{
    // Clamp the keyframe
    if (keyframeIndex >= mKeyframes.size() )
    {
        keyframeIndex = mKeyframes.size() - 1;
    }

    Keyframe* keyframe = mKeyframes[keyframeIndex];

    XMStoreFloat4x4(&transform, keyframe->Transform());
}

void BoneAnimation::GetInterpolatedTransform(float time,
XMFLOAT4X4&
transform) const
{
    Keyframe* firstKeyframe = mKeyframes.front();
    Keyframe* lastKeyframe = mKeyframes.back();

    if (time <= firstKeyframe->Time())
    {
        // Specified time is before the start time of the
        animation, so
        return the first keyframe
        XMStoreFloat4x4(&transform, firstKeyframe->Transform());
    }
}
```

```

}

else if (time >= lastKeyframe->Time())
{
    // Specified time is after the end time of the
animation, so
return the last keyframe
    XMStoreFloat4x4(&transform, lastKeyframe->Transform());
}
else
{
    // Interpolate the transform between keyframes
    UINT keyframeIndex = FindKeyframeIndex(time);
    Keyframe* keyframeOne = mKeyframes[keyframeIndex];
    Keyframe* keyframeTwo = mKeyframes[keyframeIndex + 1];

        XMVECTOR translationOne = keyframeOne-
>TranslationVector();
        XMVECTOR rotationQuaternionOne =
keyframeOne->RotationQuaternionVector();
        XMVECTOR scaleOne = keyframeOne->ScaleVector();

        XMVECTOR translationTwo = keyframeTwo-
>TranslationVector();
        XMVECTOR rotationQuaternionTwo =
keyframeTwo->RotationQuaternionVector();
        XMVECTOR scaleTwo = keyframeTwo->ScaleVector();

        float lerpValue = ((time - keyframeOne-
>Time()) / (keyframeTwo->Time() - keyframeOne->Time()));
        XMVECTOR translation = XMVectorLerp(translationOne,
translationTwo, lerpValue);
        XMVECTOR rotationQuaternion = XMQuaternionSlerp
(rotationQuaternionOne, rotationQuaternionTwo, lerpValue);
        XMVECTOR scale = XMVectorLerp(scaleOne, scaleTwo,
lerpValue);

        XMVECTOR rotationOrigin =
XMLoadFloat4(&Vector4Helper::Zero);
        XMStoreFloat4x4(&transform,
XMMatrixAffineTransformation(scale,
rotationOrigin, rotationQuaternion, translation));
    }
}

UINT BoneAnimation::FindKeyframeIndex(float time) const
{
    Keyframe* firstKeyframe = mKeyframes.front();

```

```

if (time <= firstKeyframe->Time())
{
    return 0;
}

Keyframe* lastKeyframe = mKeyframes.back();
if (time >= lastKeyframe->Time())
{
    return mKeyframes.size() - 1;
}

UINT keyframeIndex = 1;

for (; keyframeIndex < mKeyframes.size() - 1 && time >=
mKeyframes[keyframeIndex]->Time(); keyframeIndex++);
}

return keyframeIndex - 1;
}

```

---

The BoneAnimation::GetInterpolatedTransform() method deserves the closest examination. Here, the two surrounding keyframes are selected and their data are retrieved. This equation determines the interpolation value:

$$lerpValue = (time - keyframe_1.Time) / (keyframe_2.Time - keyframe_1.Time)$$

The following scenario demonstrates this calculation:

$$\begin{aligned}
time &= 0.1 \\
keyframe_1.Time &= 0 \\
keyframe_2.Time &= 0.5 \\
lerpValue &= (0.1 - 0) / (0.5 - 0) = .1 / .5 = 0.2
\end{aligned}$$

As we've discussed in previous chapters, linear interpolation is calculated as:

$$lerp(x,y,s) = x * (1 - s) + (y * s)$$

Thus, in the previous scenario, a lerp value of 0.2 implies that 80 percent of computed value comes from key frame 1, and 20 percent comes from keyframe 2.

The BoneAnimation::GetInterpolatedTransform() method calculates the interpolated translation, rotation, and scale, and uses these values to compute the bone transformation.

## Animation Rendering

With all the data imported and the supporting infrastructure in place, you are now ready to render an animation. Rendering an animated model works based on the following steps:

1. Advance the current time.

2. Update the transformations of each bone in the skeleton.

3. Send the updated bone transformations to the skinned model shader.

4. Execute the draw call(s) for the skinned model.

Most of these steps are encapsulated within an AnimationPlayer component (see [Listing 20.13](#)).

### **Listing 20.13 Declaration of the AnimationPlayer Class**

[Click here to view code image](#)

---

```
class AnimationPlayer : GameComponent
{
    RTTI_DECLARATIONS(AnimationPlayer, GameComponent)

public:
    AnimationPlayer(Game& game, Model& model, bool
interpolationEnabled
= true);

    const Model& GetModel() const;
    const AnimationClip* CurrentClip() const;
    float CurrentTime() const;
    UINT CurrentKeyframe() const;
    const std::vector<XMFLOAT4X4>& BoneTransforms() const;

    bool InterpolationEnabled() const;
    bool IsPlayingClip() const;
    bool IsClipLooped() const;

    void SetInterpolationEnabled(bool interpolationEnabled);

    void StartClip(AnimationClip& clip);
    void PauseClip();
    void ResumeClip();
    virtual void Update(const GameTime& gameTime) override;
    void SetCurrentKeyFrame(UINT keyframe);

private:
    AnimationPlayer();
    AnimationPlayer(const AnimationPlayer& rhs);
    AnimationPlayer& operator=(const AnimationPlayer& rhs);

    void GetBindPose(SceneNode& sceneNode);
    void GetPose(float time, SceneNode& sceneNode);
    void GetPoseAtKeyframe(UINT keyframe, SceneNode& sceneNode);
    void GetInterpolatedPose(float time, SceneNode& sceneNode);
```

```

Model* mModel;
AnimationClip* mCurrentClip;
float mCurrentTime;
UINT mCurrentKeyframe;
std::map<SceneNode*, XMFLOAT4X4> mToRootTransforms;
std::vector<XMFLOAT4X4> mFinalTransforms;
XMFLOAT4X4 mInverseRootTransform;
bool mInterpolationEnabled;
bool mIsPlayingClip;
bool mIsClipLooped;
};


```

---

The AnimationPlayer class is associated with a model and has an active AnimationClip specified with a call to StartClip(). This method also resets the mCurrentTime and mCurrentKeyFrame members and sets mIsPlayingClip to true. The full implementation of StartClip() is presented here:

[Click here to view code image](#)

```

void AnimationPlayer::StartClip(AnimationClip& clip)
{
    mCurrentClip = &clip;
    mCurrentTime = 0.0f;
    mCurrentKeyframe = 0;
    mIsPlayingClip = true;

    XMATRIX inverseRootTransform = XMMatrixInverse
    (&XMMatrixDeterminant(mModel->RootNode()->TransformMatrix()), ,
    mModel->RootNode()->TransformMatrix());
    XMStoreFloat4x4(&mInverseRootTransform,
    inverseRootTransform);
    GetBindPose(* (mModel->RootNode()) );
}


```

The last few lines of StartClip() store the inverse root transformation and invoke the GetBindPose() method. The inverse of the model's root node transformation is the last transform you apply to each bone, so it's computed once and saved for subsequent use. The GetBindPose() method is a recursive call invoked against the model's root node and initializes the model into its **bind pose**. The bind pose is the initial position of a model, before any animation is applied. More specifically, the bind pose is the result of the initial transformations for each bone in a skinned model. [Listing 20.14](#) shows a bottom-up implementation of GetBindPose().

## Listing 20.14 Retrieving a Skinned Model's Bind Pose (Bottom-up)

[Click here to view code image](#)

---

```

void AnimationPlayer::GetBindPose(SceneNode& sceneNode)
{


```

```

XMMATRIX toRootTransform = sceneNode.TransformMatrix();

SceneNode* parentNode = sceneNode.Parent();
while (parentNode != nullptr)
{
    toRootTransform = toRootTransform *
parentNode->TransformMatrix();
    parentNode = parentNode->Parent();
}

Bone* bone = sceneNode.As<Bone>();
if (bone != nullptr)
{
    XMStoreFloat4x4(&(mFinalTransforms[bone->Index()]),
bone->OffsetTransformMatrix() * toRootTransform * XMLoadFloat4x4
(&mInverseRootTransform));
}

for (SceneNode* childNode : sceneNode.Children())
{
    GetBindPose(*childNode);
}
}

```

---

This implementation first retrieves the transformation matrix for the scene node. This is a relative transform (relative to the parent node) but is used to initialize the `toRootTransform` variable. The *to-root* transform changes the coordinate space of the node from its own bone space to the model space of the root node. If the incoming scene node is the root node (that is, it has no parent), the to-root transform is the relative *to-parent* transform. Otherwise, the to-root transform is combined with the transform for each ancestor in the hierarchy. The final transform (what's sent to the skinned model shader) is composed from the offset transform, the to-root transform, and the inverse root transform. Recall that the vertices associated with the bone are originally in model space and that the offset transform moves them to bone space so that the skeleton can transform them.

This version of `GetBindPose()` is a bottom-up approach because you are traversing the hierarchy from the current node up for each node in the skeleton. [Listing 20.15](#) presents the same method using a top-down approach, which sacrifices some memory to eliminate duplicate multiplications.

## **Listing 20.15** Retrieving a Skinned Model's Bind Pose (Top-down)

[Click here to view code image](#)

---

```

void AnimationPlayer::GetBindPose(SceneNode& sceneNode)
{
    XMMATRIX toParentTransform = sceneNode.TransformMatrix();
    XMMATRIX toRootTransform = (sceneNode.Parent() != nullptr ?
toParentTransform * XMLoadFloat4x4(&

```

```

(mToRootTransforms.at(sceneNode.
Parent())) : toParentTransform);
XMStoreFloat4x4(&(mToRootTransforms[&sceneNode]), 
toRootTransform);

Bone* bone = sceneNode.As<Bone>();
if (bone != nullptr)
{
    XMStoreFloat4x4(&(mFinalTransforms[bone->Index()]),
bone->OffsetTransformMatrix() * toRootTransform * XMLoadFloat4x4
(&mInverseRootTransform));
}

for (SceneNode* childNode : sceneNode.Children())
{
    GetBindPose(*childNode);
}
}

```

---

After the animation clip is initialized, the animation can be automatically advanced as time passes. This is accomplished through the overridden `Update()` method, derived from the `GameComponent` class. [Listing 20.16](#) presents this method.

## **Listing 20.16** Advancing an Animation Automatically

[Click here to view code image](#)

---

```

void AnimationPlayer::Update(const GameTime& gameTime)
{
    if (mIsPlayingClip)
    {
        assert(mCurrentClip != nullptr);

        mCurrentTime += static_cast<float>
(gameTime.ElapsedGameTime())
* mCurrentClip->TicksPerSecond();
        if (mCurrentTime >= mCurrentClip->Duration())
        {
            if (mIsClipLooped)
            {
                mCurrentTime = 0.0f;
            }
            else
            {
                mIsPlayingClip = false;
                return;
            }
        }
    }
}

```

```

        }

        if (mInterpolationEnabled)
        {
            GetInterpolatedPose (mcurrentTime, * (mModel-
>RootNode ()) );
        }
        else
        {
            GetPose (mcurrentTime, * (mModel->RootNode ()) );
        }
    }
}

```

---

The `Update()` method first advances time according to the elapsed frame time and the clip's ticks-per-second property (recall that the animation clip's duration is stored in ticks, not seconds). If the clip is looped, it will reset the time after the animation has finished; otherwise, the clip stops. If time is left within the playing clip, the final bone transformations are updated through the `GetPose()` or `GetInterpolatedPose()` methods. [Listing 20.17](#) presents the `GetInterpolatedPose()` method. `GetPose()` is identical, except that it invokes `AnimationClip::GetTransform()` instead of `AnimationClip::GetInterpolatedTransform()` to update the `toParentTransform` matrix.

## **Listing 20.17** Retrieving the Current Pose

[Click here to view code image](#)

---

```

void AnimationPlayer::GetInterpolatedPose (float time, SceneNode&
sceneNode)
{
    XMFLOAT4X4 toParentTransform;
    Bone* bone = sceneNode.As<Bone> ();
    if (bone != nullptr)
    {
        mCurrentClip->GetInterpolatedTransform (time, *bone,
toParentTransform);
    }
    else
    {
        toParentTransform = sceneNode.Transform ();
    }

    XMATRIX toRootTransform = (sceneNode.Parent () != nullptr ?
XMLoadFloat4x4 (&toParentTransform) * XMLoadFloat4x4 (&
(mToRootTransforms.at (sceneNode.Parent ()))) : XMLoadFloat4x4
(&toParentTransform));

```

```

XMStoreFloat4x4(&(mToRootTransforms[&sceneNode]),  

toRootTransform);

if (bone != nullptr)
{
    XMStoreFloat4x4(&(mFinalTransforms[bone->Index()]),  

bone->OffsetTransformMatrix() * toRootTransform * XMLoadFloat4x4  

(&mInverseRootTransform));
}

for (SceneNode* childNode : sceneNode.Children())
{
    GetInterpolatedPose(time, *childNode);
}
}

```

---

You can also manually advance from frame to frame with the `GetPoseAtKeyframe()` method, which is invoked through the `SetCurrentKeyFrame()` method:

[Click here to view code image](#)

```

void AnimationPlayer::SetCurrentKeyFrame(UINT keyframe)
{
    mCurrentKeyframe = keyframe;
    GetPoseAtKeyframe(mCurrentKeyframe, *(mModel->RootNode()));
}

```

The final transforms (what you pass to the skinned model shader) are exposed through the `AnimationPlayer::BoneTransforms()` accessor. The `AnimationPlayer` class also exposes accessors for the associated model, current animation clip, current time, and current keyframe, as well as methods for pausing and resuming the clip. Visit the book's companion website for the complete implementation.

Using the `AnimationPlayer` is a matter of instantiating the class, starting an animation clip, and either manually advancing the keyframes or updating the `AnimationPlayer` component. This also implies that you have instantiated a `Model` object with a file containing a skeletal hierarchy and animations. The book's companion website contains a sample model in COLLADA (.dae) format. You must also create a material class to interface with the skinned model shader. This material has just one animation-specific variable (`BoneTransforms`) and creates vertex buffers that include the bone weights and indices. The material's input layout should have the following elements:

[Click here to view code image](#)

```

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
D3D11_INPUT
PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0,
D3D11_APPEND_ALIGNED_

```

```

ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
D3D11_APPEND_ALIGNED_
ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "BONEINDICES", 0, DXGI_FORMAT_R32G32B32A32_UINT, 0,
D3D11_APPEND_
ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "WEIGHTS", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
D3D11_APPEND_
ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

```

Note the format of the `BoneIndices` and `Weights` elements— $4 \times 32$ -bit unsigned integers for the indices and  $4 \times 32$ -bit floating point values for the weights.

Drawing the skinned model is identical to drawing any other models, with the exception that you pass the bone transforms to the shader. For example:

[Click here to view code image](#)

```
mMaterial->BoneTransforms () << mAnimationPlayer-
>BoneTransforms ();
```

Here, the `mMaterial` object refers to an instance of the `SkinnedModelMaterial` class. An interactive demo is available on the companion website. This demo enables you to vary the options of the animation player and manually step through keyframes. [Figure 20.3](#) shows the animation demo rendering the running soldier depicted at the beginning of this chapter.



**Figure 20.3** Output of the animation demo. (*Animated model provided by Brian Salisbury, Florida Interactive Entertainment Academy.*)

## Summary

In this chapter, you explored skeletal animation. You developed the systems to import animation data from the Open Asset Import Library, as well as the components to retrieve and interpolate between keyframes. You also implemented a shader to transform the skinned model's vertices on the GPU.

## Exercises

1. From within the debugger, walk through the code used to import an animated model, to better understand all the moving parts (pun intended). Import a variety of animated models to compare the idiosyncrasies between file formats (which the Open Asset Import Library might not successfully abstract).
2. Explore the animation demo found on the book's companion website. Vary the options provided to automatically and manually advance keyframes with and without interpolation, and observe the results.
3. Integrate the animation shader with additional lighting models (directional lighting and spot lighting, for example).

# Chapter 21. Geometry and Tessellation Shaders

In this chapter, you examine two relatively new additions to the Direct3D graphics pipeline: geometry and tessellation shaders. These systems enable you to create vertices dynamically and even change the topology of a surface on the hardware. You learn how these systems work and implement a variety of interesting effects.

## Motivation: Geometry Shaders

Geometry shaders have the capability to add and remove geometry from the graphics pipeline. That capability is absent from the pipeline stages we've discussed thus far and allows for some interesting applications. For example, lower-fidelity geometry can be sent to the pipeline, and the geometry shader can conjure up more vertices. Conversely, the geometry shader can excise entire primitives, or portions of them, from further processing.

## Processing Primitives

The vertex shader operates on individual vertices, and the pixel shader operates on individual pixels, but the geometry shader operates on entire primitives. Recall that three basic primitive types exist: points, lines, and triangles (although they can be organized as lists and strips and can include adjacency data—see [Chapter 1, “Introducing DirectX,”](#) for a refresher). When programming a geometry shader, you must specify which type of primitive is to be processed using the keywords: point, line, triangle, lineadj, and triangleadj. That's one of a few syntactical differences between geometry shaders and vertex or pixel shaders. The following code presents an example declaration of a geometry shader.

[Click here to view code image](#)

```
[maxvertexcount (3)]
void geometry_shader(point VS_OUTPUT IN[1], inout
TriangleStream<GS_
OUTPUT> triStream) { // shader body }
```

The maxvertexcount attribute specifies the maximum number of vertices that can be output by the geometry shader (three, in this example). The shader's first parameter defines the type of primitive (point), the structure of the input data (VS\_OUTPUT), and how many vertices will be processed on each invocation of the geometry shader ([1]). The VS\_OUTPUT data type is just the naming convention we've adopted for vertex shader output, and it can be any HLSL data type. Because the geometry shader stage is fed by the vertex shader stage (without active tessellation shaders), it makes sense that the output of the vertex shader becomes the input to the geometry shader. And with the inclusion of the geometry shader, the output of the geometry shader (by convention, GS\_OUTPUT) becomes the input to the pixel shader.

The [n] array-style syntax, of the geometry shader's first parameter, varies with the specified primitive type according to [Table 21.1](#).

Primitive Type	Number of Elements
point	1 (one vertex per point)
line	2 (two vertices per line)
triangle	3 (three vertices per triangle)
lineadj	4 (two vertices in the line, plus two adjacent vertices)
triangleadj	6 (three vertices in the triangle, plus three adjacent vertices)

**Table 21.1** The Array Size of Geometry Shader Input, According to Primitive Type

The second parameter of the geometry shader is a `StreamOutputObject<T>` type and is always prefaced with the `inout` modifier. The stream-output object is a templated data type that comes in three flavors: `PointStream`, `LineStream`, and `TriangleStream`, corresponding to the primitive type you want to output. The `PointStream` type is analogous to the point list topology (it outputs just a set of points), but the `LineStream` and `TriangleStream` types output line and triangle strips, respectively.

The basic behavior of a geometry shader is to process incoming primitives and append vertices to the stream-output object. You append vertices with the `StreamOutputObject<T>::Append()` method. Extending the previous example, we define our geometry shader as accepting point primitives and outputting a triangle stream with a maximum of three vertices (just one triangle in the stream for each invocation of the geometry shader). The following code demonstrates the basic outline of such a shader:

[Click here to view code image](#)

```

struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct GS_OUTPUT
{
    float4 Position : SV_Position;
};

[maxvertexcount(3)]
void geometry_shader(point VS_OUTPUT IN[1], inout
TriangleStream<GS_
OUTPUT> triStream)
{
    GS_OUTPUT OUT = (GS_OUTPUT)0;

    for (int i = 0; i < 3; i++)
    {
        OUT.Position = // Some modification of the input point,
followed

```

```

by transformation into homogeneous-clip space
triStream.Append(OUT);
}
}

```

## A Point Sprite Shader

A common application of geometry shaders is point sprite expansion. A **point sprite** enables you to render a textured quad (or other shape) while specifying just a single vertex (a point). The point represents the center of the quad, and the geometry shader constructs the four surrounding vertices to form the quad. The quads could share a fixed size or could specify a custom size per-vertex. Furthermore, such a system could allow for a single texture (the same image mapped to every quad) or an array of textures (where the index into the array is specified per vertex). [Listing 21.1](#) presents a point sprite shader with a fixed texture and per-vertex quad sizes. This shader **billboards** the quad; it positions the vertices so that the quad always faces the camera. Two billboarding techniques are common: **spherical billboarding**, which orients the object toward the camera, regardless of axis; and **cylindrical billboarding**, which constrains the rotation about a single axis (the y-axis, for example). Cylindrical billboarding is useful for simulating trees, where, for example, each tree should be “planted” on the ground but always oriented toward the camera to give the illusion of a 3D tree. The shader in [Listing 21.1](#) uses spherical billboarding.

### [Listing 21.1](#) A Spherical Billboard Point Sprite Shader

[Click here to view code image](#)

---

```

/********************* Resources *****/
static const float2 QuadUVs[4] = { float2(0.0f, 1.0f), // v0,
                                  lower-left
                                  float2(0.0f, 0.0f), // v1,
                                  upper-left
                                  float2(1.0f, 0.0f), // v2,
                                  upper-right
                                  float2(1.0f, 1.0f) // v3,
                                  lower-right
                                };
cbuffer CBufferPerFrame
{
    float3 CameraPosition : CAMERAPOSITION;
    float3 CameraUp;
}

cbuffer CBufferPerObject
{
    float4x4 ViewProjection;
}
```

```
Texture2D ColorTexture;

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/************************************************************************************************ Data Structures *****/
struct VS_INPUT
{
    float4 Position : POSITION;
    float2 Size : SIZE;
};

struct VS_OUTPUT
{
    float4 Position : POSITION;
    float2 Size : SIZE;
};

struct GS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/************************************************************************************************ Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.Size = IN.Size;

    return OUT;
}

/************************************************************************************************ Geometry Shader *****/
[maxvertexcount(6)]
void geometry_shader(point VS_OUTPUT IN[1], inout TriangleStream<GS_
```

```

OUTPUT> triStream)
{
    GS_OUTPUT OUT = (GS_OUTPUT)0;

    float2 halfSize = IN[0].Size / 2.0f;
    float3 direction = CameraPosition - IN[0].Position.xyz;
    float3 right = cross(normalize(direction), CameraUp);

    float3 offsetX = halfSize.x * right;
    float3 offsetY = halfSize.y * CameraUp;

    float4 vertices[4];
    vertices[0] = float4(IN[0].Position.xyz + offsetX - offsetY,
1.0f);
    // lower-left
    vertices[1] = float4(IN[0].Position.xyz + offsetX + offsetY,
1.0f);
    // upper-left
    vertices[2] = float4(IN[0].Position.xyz - offsetX + offsetY,
1.0f);
    // upper-right
    vertices[3] = float4(IN[0].Position.xyz - offsetX - offsetY,
1.0f);
    // lower-right

    // tri: 0, 1, 2
    OUT.Position = mul(vertices[0], ViewProjection);
    OUT.TextureCoordinate = QuadUVs[0];
    triStream.Append(OUT);

    OUT.Position = mul(vertices[1], ViewProjection);
    OUT.TextureCoordinate = QuadUVs[1];
    triStream.Append(OUT);

    OUT.Position = mul(vertices[2], ViewProjection);
    OUT.TextureCoordinate = QuadUVs[2];
    triStream.Append(OUT);
    triStream.RestartStrip();

    // tri: 0, 2, 3
    OUT.Position = mul(vertices[0], ViewProjection);
    OUT.TextureCoordinate = QuadUVs[0];
    triStream.Append(OUT);

    OUT.Position = mul(vertices[2], ViewProjection);
    OUT.TextureCoordinate = QuadUVs[2];
    triStream.Append(OUT);
}

```

```

        OUT.Position = mul(vertices[3], ViewProjection);
        OUT.TextureCoordinate = QuadUVs[3];
        triStream.Append(OUT);
    }

***** Pixel Shader *****

float4 pixel_shader(GS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
}

***** Techniques *****

technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(CompileShader(gs_5_0,
geometry_shader()));
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}

```

---

No lighting model is applied in this example, in order to focus on the geometry shader. The pixel shader just samples the color texture. The vertex shader, too, is extremely simple—it merely passes through the position of the vertex and size of the quad to construct. The work is primarily done in the geometry shader, which takes in a single point, constructs four, and then outputs six total vertices (two triangles in a list, duplicating two of the constructed points). Because the incoming point represents the center of the quad, you build up the four surrounding vertex positions at half the size horizontally and vertically along the plane of quad. You compute the vector representing the angle of the plane (with regard to the camera) by first calculating the direction vector from the center point to the camera (this can also be used as the surface normal). In this example, the center points are already in world space and, therefore, do not need to be transformed before calculation against the camera position. The orthogonal surface vector (named `right`) is created from the cross product of the direction vector and the camera's up vector. The four vertices are positioned along the `right` vector using the horizontal and vertical offsets.

Finally, the vertices of the two triangles are appended to the stream-output object with their positions transformed to homogeneous clip space and their texture coordinates set. Note the invocation of the `StreamOutputObject<T>::RestartStrip()` method. Recall that `LineStream` and `TriangleStream` types output strips, but you can emulate a list by restarting the strip between objects. You can rewrite this shader to output a triangle strip using the code in [Listing 21.2](#).

## Listing 21.2 Updated Point Sprite Geometry Shader Using Triangle Strips

[Click here to view code image](#)

```
static const float2 QuadStripUVs[4] = { float2(0.0f, 1.0f), //  
    v0,  
    lower-left  
    float2(0.0f, 0.0f), //  
    v1,  
    upper-left  
    float2(1.0f, 1.0f), //  
    v2,  
    lower-right  
    float2(1.0f, 0.0f) //  
    v3,  
    upper-right  
};  
  
[maxvertexcount(4)]  
void geometry_shader(point VS_OUTPUT IN[1], inout  
TriangleStream<GS_  
OUTPUT> triStream)  
{  
    GS_OUTPUT OUT = (GS_OUTPUT)0;  
  
    float2 halfSize = IN[0].Size / 2.0f;  
    float3 direction = CameraPosition - IN[0].Position.xyz;  
    float3 right = cross(normalize(direction), CameraUp);  
  
    float3 offsetX = halfSize.x * right;  
    float3 offsetY = halfSize.y * CameraUp;  
    float4 vertices[4];  
    vertices[0] = float4(IN[0].Position.xyz + offsetX - offsetY,  
1.0f);  
    // lower-left  
    vertices[1] = float4(IN[0].Position.xyz + offsetX + offsetY,  
1.0f);  
    // upper-left  
    vertices[2] = float4(IN[0].Position.xyz - offsetX - offsetY,  
1.0f);  
    // lower-right  
    vertices[3] = float4(IN[0].Position.xyz - offsetX + offsetY,  
1.0f);  
    // upper-right  
  
    [unroll]  
    for (int i = 0; i < 4; i++)  
    {  
        OUT.Position = mul(vertices[i], ViewProjection);  
        OUT.TextureCoordinate = QuadStripUVs[i];  
    }  
}
```

```
    triStream.Append(OUT) ;  
}  
}
```

---

Note how the vertex order has changed between the two iterations to accommodate the triangle strip. Using the geometry shader from the CPU side requires no special coding constructs. In the geometry shader demo (full source code available online), a random set of points and quad sizes is created and rendered with the code in [Listing 21.3](#).

### Listing 21.3 Initialization and Rendering for the Geometry Shader Demo

[Click here to view code image](#)

---

```
void GeometryShaderDemo::Initialize()  
{  
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());  
  
    // Initialize the material  
    mEffect = new Effect(*mGame);  
    mEffect->LoadCompiledEffect(L"Content\\Effects\\PointSprite.cso");  
    mMaterial = new PointSpriteMaterial();  
    mMaterial->Initialize(*mEffect);  
  
    mPass = mMaterial->CurrentTechnique()->Passes().at(0);  
    mInputLayout = mMaterial->InputLayouts().at(mPass);  
  
    UINT maxRandomPoints = 100;  
    float maxDistance = 10;  
    float minSize = 2;  
    float maxSize = 2;  
  
    std::random_device randomDevice;  
    std::default_random_engine randomGenerator(randomDevice());  
    std::uniform_real_distribution<float> distanceDistribution  
        (-maxDistance, maxDistance);  
    std::uniform_real_distribution<float>  
        sizeDistribution(minSize,  
        maxSize);  
  
    // Randomly generate points  
    std::vector<VertexPositionSize> vertices;  
    vertices.reserve(maxRandomPoints);  
    for (UINT i = 0; i < maxRandomPoints; i++)  
    {  
        float x = distanceDistribution(randomGenerator);  
        float y = distanceDistribution(randomGenerator);  
        float z = distanceDistribution(randomGenerator);  
        float size = sizeDistribution(minSize, maxSize);  
        vertices.push_back({x, y, z, size});  
    }  
}
```

```

        float y = distanceDistribution(randomGenerator);
        float z = distanceDistribution(randomGenerator);

        float size = sizeDistribution(randomGenerator);

        vertices.push_back(VertexPositionSize(XMFLOAT4(x, y, z,
1.0f),
XMFLOAT2(size, size)));
    }

    mVertexCount = vertices.size();
    mMaterial->CreateVertexBuffer(mGame->Direct3DDevice(),
&vertices[0], mVertexCount, &mVertexBuffer);

    std::wstring textureName =
L"Content\\Textures\\BookCover.png";
    HRESULT hr = DirectX::CreateWICTextureFromFile(mGame->
Direct3DDevice(), mGame->Direct3DDeviceContext(),
textureName.c_str(),
nullptr, &mColorTexture);
    if (FAILED(hr))
    {
        throw GameException("CreateWICTextureFromFile()
failed.", hr);
    }
}

void GeometryShaderDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_POINTLIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = mMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mVertexBuffer,
&stride, &offset);

    mMaterial->ViewProjection() << mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mMaterial->CameraPosition() << mCamera->PositionVector();
    mMaterial->CameraUp() << mCamera->UpVector();
    mMaterial->ColorTexture() << mColorTexture;
}

```

```

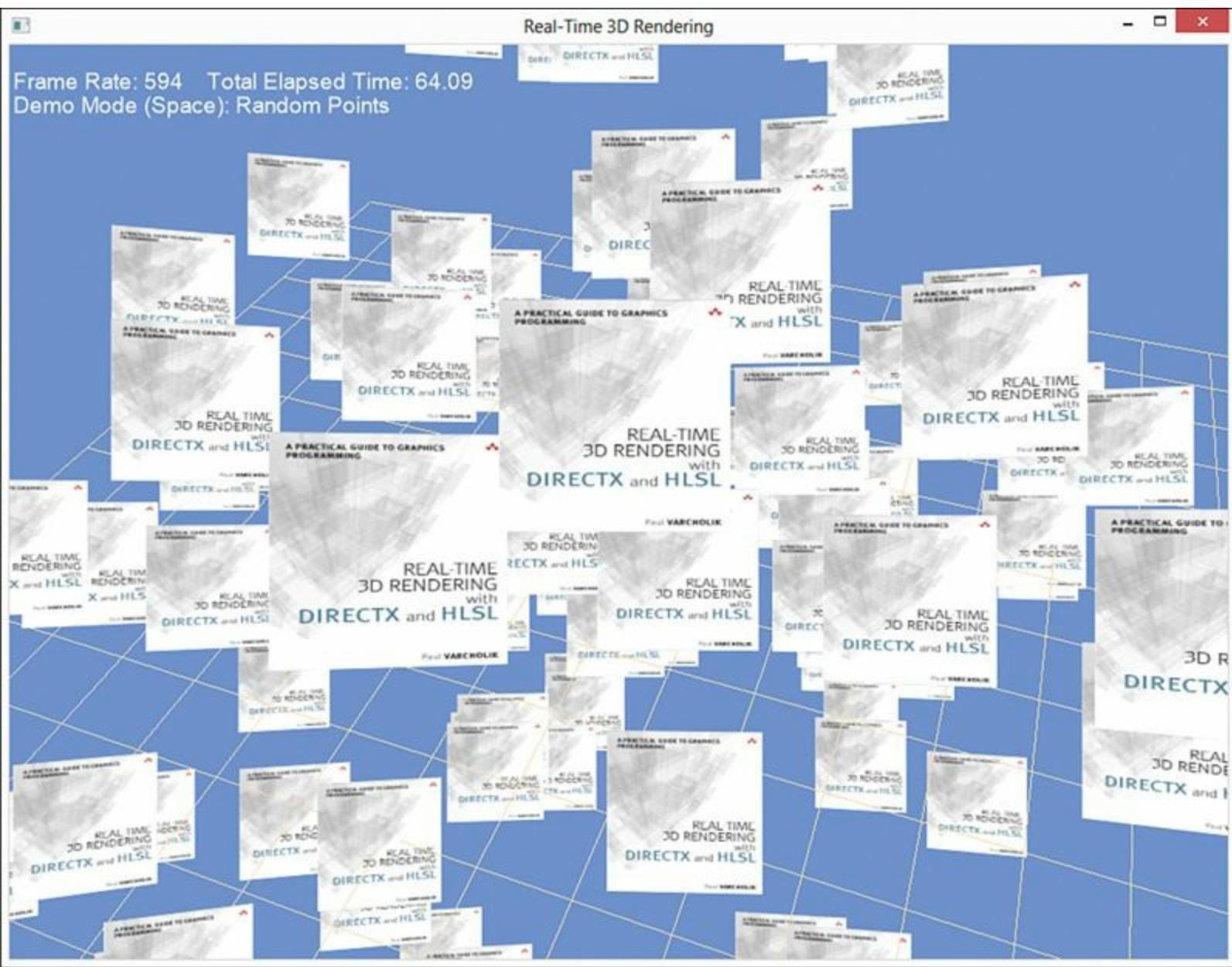
mPass->Apply(0, direct3DDeviceContext);

direct3DDeviceContext->Draw(mVertexCount, 0);

direct3DDeviceContext->GSSetShader(nullptr, nullptr, 0);
}

```

[Figure 21.1](#) shows the output of this application.



**Figure 21.1** Output of the geometry shader demo.

## Primitive IDs

Each primitive coming through the geometry shader can be tagged with a unique identifier (unique per draw call, but not between calls). This is done through the `SV_PrimitiveID` semantic attached to an unsigned integer. You can use it for some interesting effects. [Listing 21.4](#) uses the primitive ID to generate a quad size instead of passing in the size per vertex.

## **Listing 21.4** Example of SV\_PrimitiveID Usage

[Click here to view code image](#)

---

```
[maxvertexcount(4)]
void geometry_shader_nosize(point VS_NOSIZE_OUTPUT IN[1], uint
primitiveID : SV_PrimitiveID, inout TriangleStream<GS_OUTPUT>
triStream)
{
    GS_OUTPUT OUT = (GS_OUTPUT)0;

    float size = primitiveID + 1;

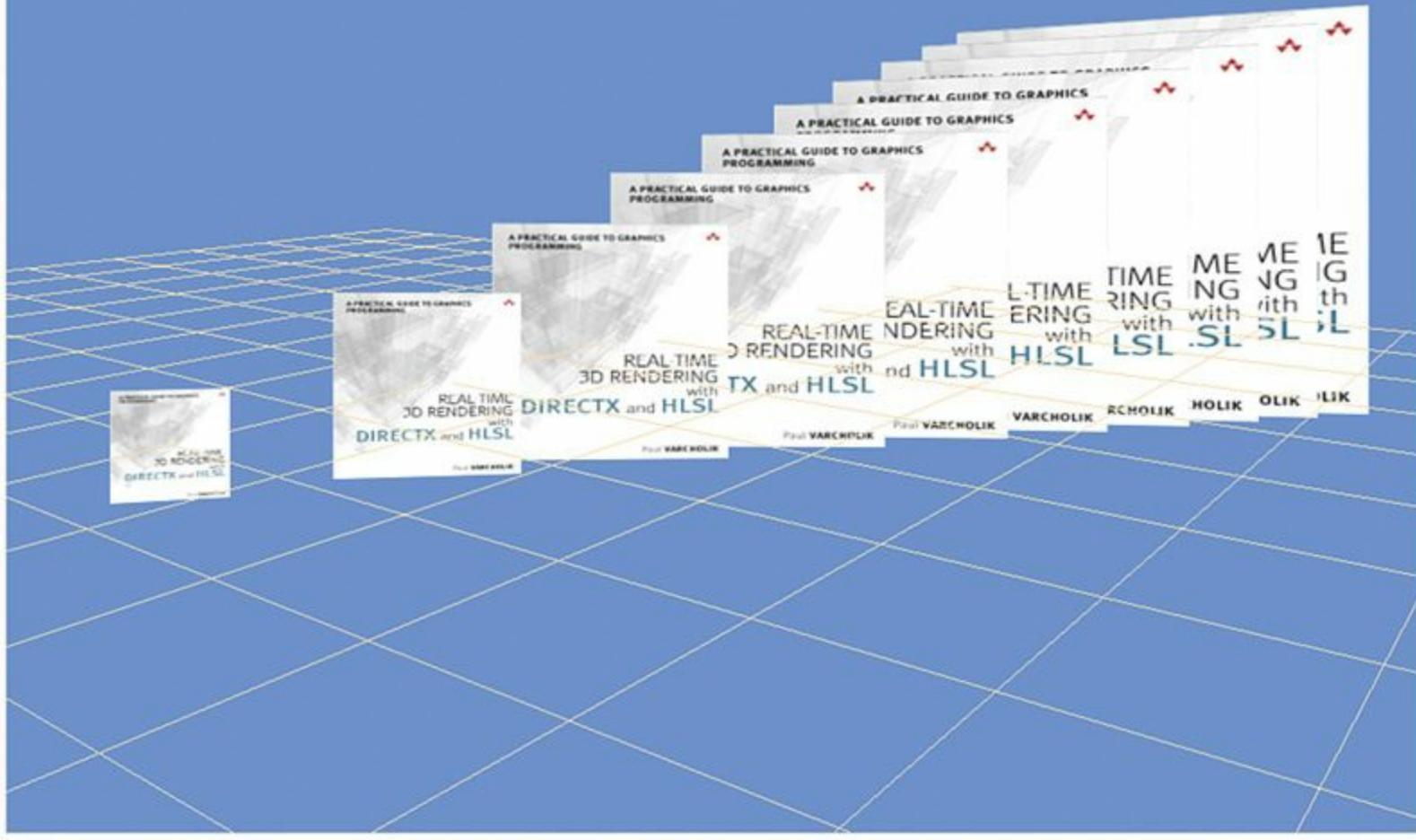
    float2 halfSize = size / 2.0f;

    // Remaining code identical to previous iteration, and
    removed for
    brevity
}
```

---

If you update the vertex buffer creation code (to produce points at regular intervals along the x-axis), this shader creates the output in [Figure 21.2](#).

Frame Rate: 596 Total Elapsed Time: 154.5  
Demo Mode (Space): Fixed Points



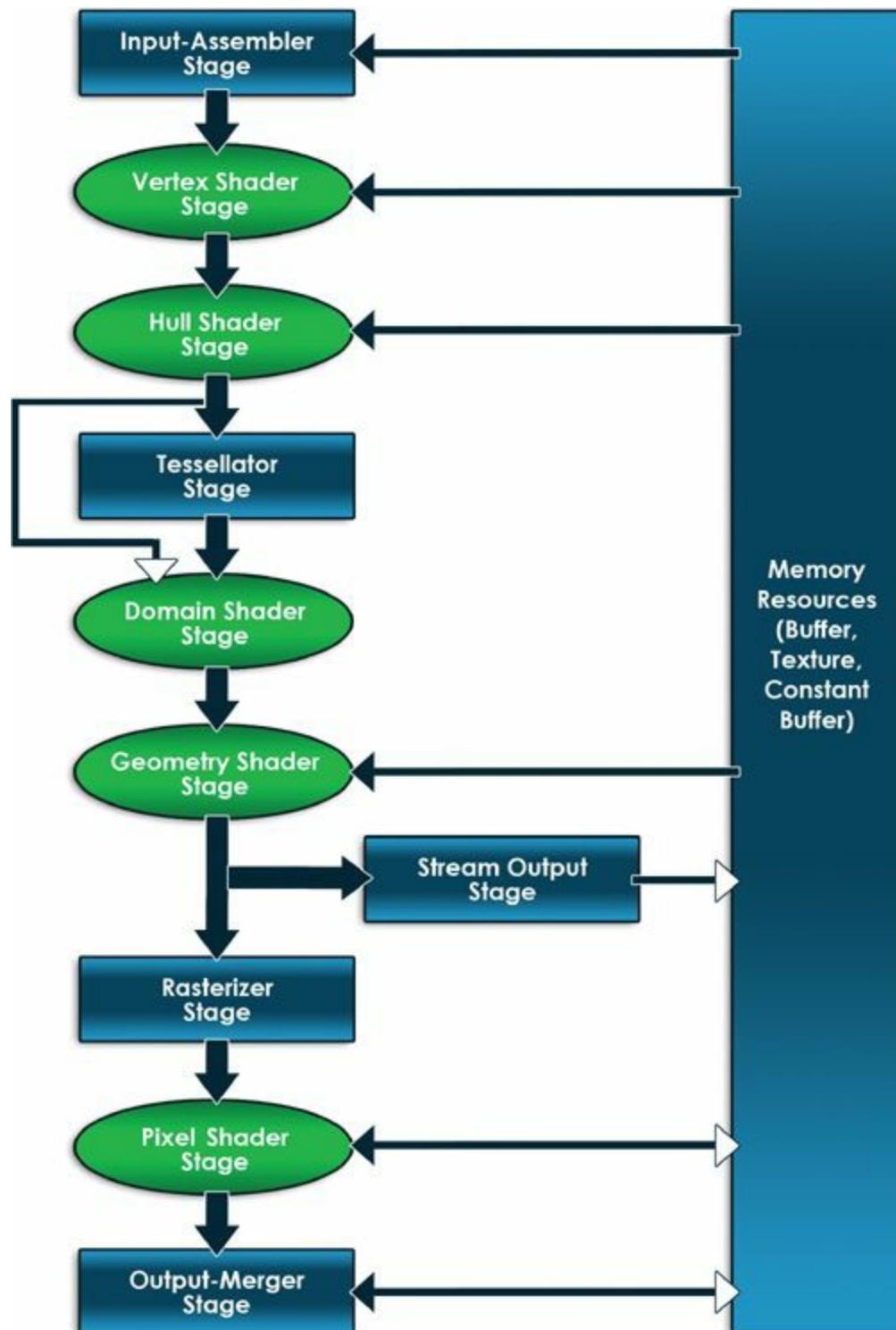
**Figure 21.2** Output of the geometry shader demo, with sizes based on the primitive ID.

The vertex shader also supports an ID, using the `SV_VertexID` semantic, with analogous behavior. These IDs need not be used strictly within their corresponding shader stages. You could pass an ID through an output variable from one shader stage to the next.

## Motivation: Tessellation Shaders

Direct3D 11 introduced hardware tessellation. Tessellation refers to the subdivision of surfaces—adding geometry to a surface to increase its fidelity. This allows low-poly models to be transmitted to the GPU without sacrificing the quality of high-poly rendering. The throughput of the bus between the CPU and the GPU is often a bottleneck, so reducing the amount of data sent over the bus is a good idea. Furthermore, without hardware tessellation, 3D models are often created at multiple resolutions (such as low, medium, and high) with a particular level of detail (LOD) rendered with respect to some metric (such as the distance of the object to the camera). This means additional work by an artist or an additional step in the content pipeline (to automatically produce the LODs), as well as more storage and processing requirements for the model. Hardware tessellation makes dynamic LODs possible using just one, low-poly version of the model.

Before the tessellation pipeline stages were added, the geometry shader was used for surface subdivision, but now there are dedicated tessellation stages: the hull-shader stage, the tessellation stage, and the domain-shader stage. Recall the graphics pipeline in [Figure 21.3](#) and the tessellation stages between the vertex and geometry shader stages. The next sections discuss each of these stages and then provide demonstration applications.



**Figure 21.3** The Direct3D 11 graphics pipeline.

## The Hull Shader Stage

With tessellation enabled, the vertex shader processes control point patch lists instead of the traditional point, line, or triangle primitives. A **patch** is a surface whose shape is defined by its associated control points. Direct3D 11 supports patches with 1 to 32 control points. A patch with

three control points is merely a triangle, and a quad has four control points. Patches with more than four control points refer to surfaces constructed from Bezier curves. The hull shader stage accepts these input control points and then outputs a set of control points, **tessellation factors**, and any additional data required for future stages. Tessellation factors specify the number of subdivisions for each patch. [Listing 21.5](#) presents an example hull shader for triangle patches.

## **Listing 21.5** A Hull Shader for Triangle Patches

[Click here to view code image](#)

---

```
struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[3] : SV_TessFactor;
    float InsideFactor : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("constant_hull_shader")]
HS_OUTPUT hull_shader(InputPatch<VS_OUTPUT, 3> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;

    return OUT;
}
```

---

This listing resembles other HLSL shaders, but with syntax that's specific to hull shaders. First, note that the initial parameter to the hull shader is of type `InputPatch<T, n>`, where `T` is the output type from the vertex shader and `n` is the number of control points in the input patch. The second parameter is the unique identifier of each *output* control point. The hull shader can redefine the patch topology, with either greater or fewer output control points than input points. The hull shader is

invoked once per output control point, and the **outputcontrolpoints** attribute specifies the total number of output points. In this example, each hull shader output is defined by the `HS_OUTPUT` structure.

A number of attributes are associated with the hull shader, starting with the **domain** attribute. This attribute identifies the tessellation primitive to be processed by the hull shader. It can be **tri** (triangles), **quad** (rectangles), or **isoline** (rectangles made of sets of lines with multiple control points). The **partitioning** attribute specifies how the fractional components of tessellation factors are applied. A partitioning value of **integer** or **pow2** rounds fractional or non-power-of-two tessellation factors to the next acceptable value, respectively. A value of **fractional-even** or **fractional-odd** does not replace the fractional component, and it facilitates a smoother transition between tessellation factors. The **outputtopology** attribute defines the primitive type produced by the tessellator and can be **point**, **line**, **triangle\_cw**, or **triangle\_ccw**. Finally, the **patchconstantfunc** attribute defines the function for computing patch constant data (tessellation factors), the so-called constant hull shader. [Listing 21.6](#) gives an example of a constant hull shader.

## Listing 21.6 A Constant Hull Shader

[Click here to view code image](#)

---

```
cbuffer CBufferPerFrame
{
    float TessellationEdgeFactors[3];
    float TessellationInsideFactor;
}

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[3] : SV_TessFactor;
    float InsideFactor : SV_InsideTessFactor;
};

HS_CONSTANT_OUTPUT constant_hull_shader(InputPatch<VS_OUTPUT, 3>
patch)
{
    HS_CONSTANT_OUTPUT OUT = (HS_CONSTANT_OUTPUT) 0;

    [unroll]
    for (int i = 0; i < 3; i++)
    {
        OUT.EdgeFactors[i] = TessellationEdgeFactors[i];
    }

    OUT.InsideFactor = TessellationInsideFactor;

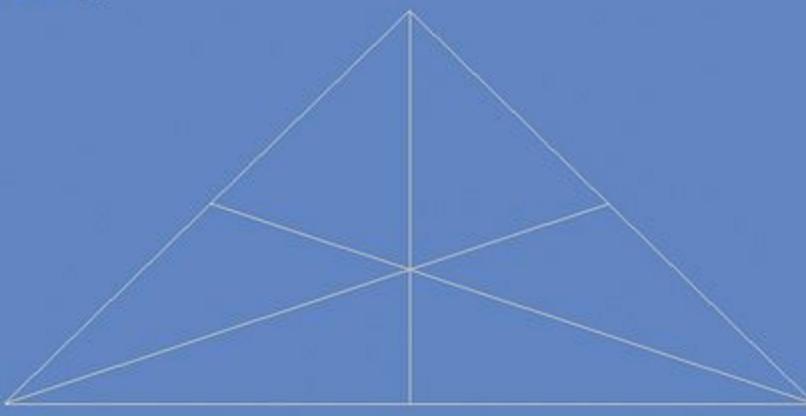
    return OUT;
}
```

The constant hull shader is invoked once per patch and must output values marked with the `SV_TessFactor` and `SV_InsideTessFactor` semantics. The `SV_TessFactor` semantic denotes **edge** tessellation factors—the number of subdivisions for each edge in the patch. The variable with the `SV_InsideTessFactor` semantic specifies the number of **inside** subdivisions. The number of edge and inside tessellation factors must match the patch topology. For example, triangle patches have three edge factors and one inside factor. Quad patches have four edge factors and two inside factors (for horizontal and vertical subdivisions). Each tessellation factor has the range  $[0, 64]$ , and the values need not be uniformly applied. If any tessellation factor is 0, the patch is discarded from further processing. A tessellation factor of 1 implies no tessellation. The code in [Listing 21.6](#) applies tessellation factors that are input from the application. These are used in the first full demonstration of tessellation, where you vary the edge and inside tessellation factors for triangle and quad patches.

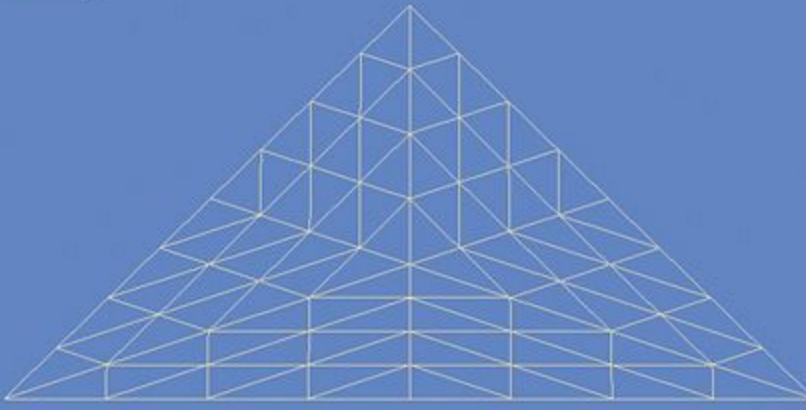
## The Tessellation Stage

The next stage is the tessellation stage, and it is not programmable. This stage performs the actual surface subdivision and is controlled by the output from the hull shader stage. [Figures 21.4](#) and [21.5](#) show a tessellated triangle and quad with different edge and inside tessellation factors.

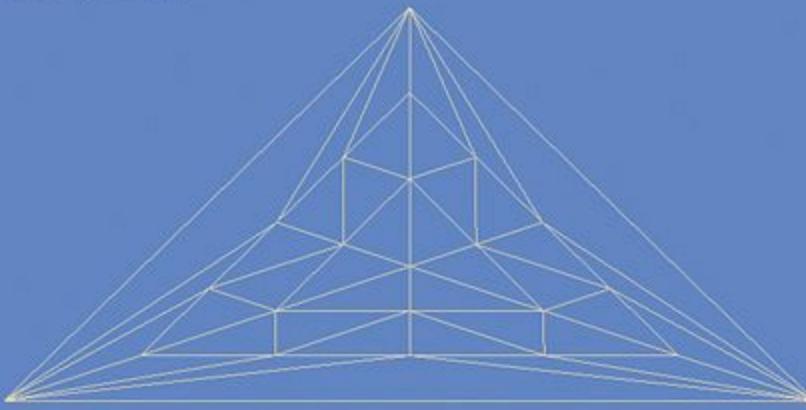
Tessellation Edge Factors: (+Up/-Down)[2, 2, 2]  
Tessellation Inside Factor: [2]



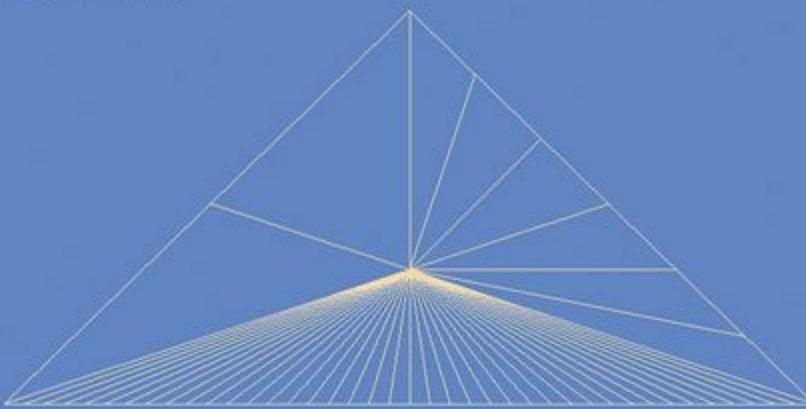
Tessellation Edge Factors: (+Up/-Down)[8, 8, 8]  
Tessellation Inside Factor: [8]



Tessellation Edge Factors: (+U/-H, +I/-J, +O/-K)[1, 1, 1]  
Tessellation Inside Factor: (+V/-B) [6]

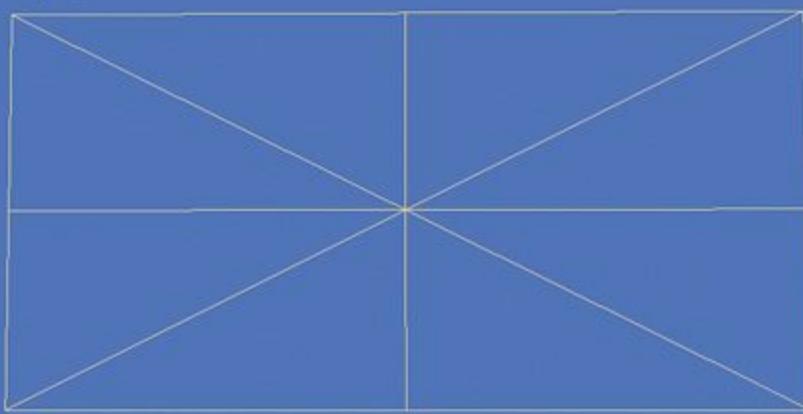


Tessellation Edge Factors: (+U/-H, +I/-J, +O/-K)[6, 40, 2]  
Tessellation Inside Factor: (+V/-B) [1]

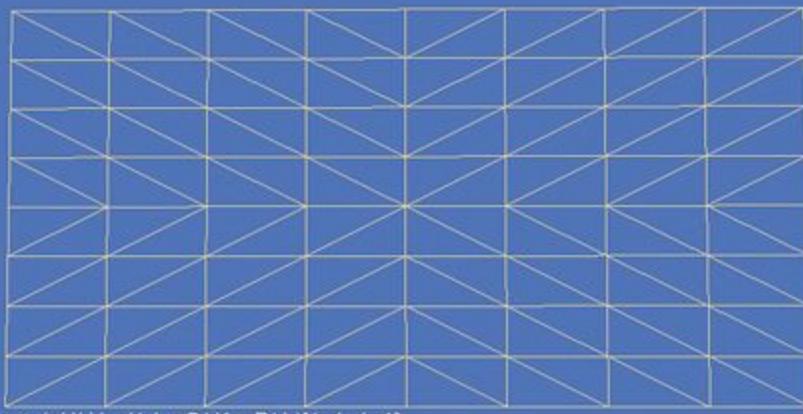


**Figure 21.4** A tessellated triangle with various tessellation factors.

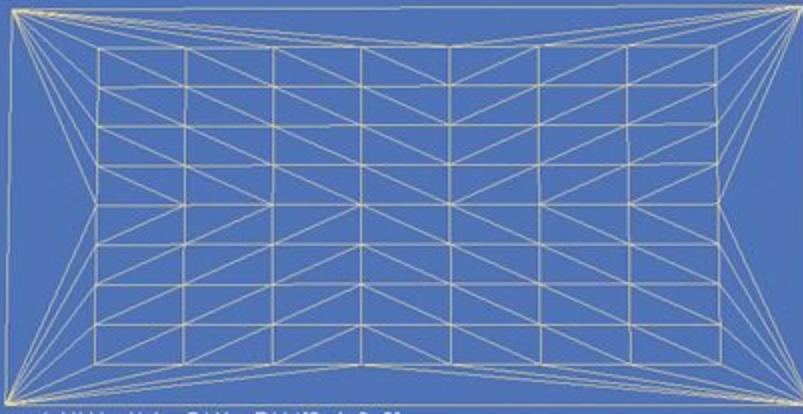
Tessellation Edge Factors: (+Up/-Down)[2, 2, 2, 2]  
Tessellation Inside Factor: [2, 2]



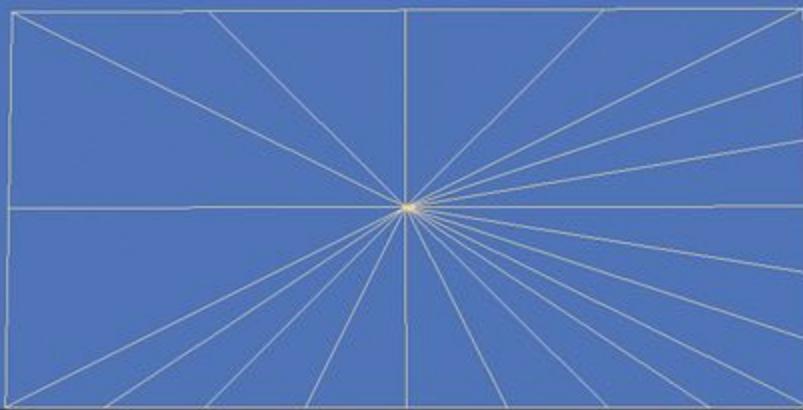
Tessellation Edge Factors: (+Up/-Down)[8, 8, 8, 8]  
Tessellation Inside Factor: [8, 8]



Tessellation Edge Factors: (+U/-H, +I/-J, +O/-K, +P/-L)[1, 1, 1, 1]  
Tessellation Inside Factor: (+V/-B, +N/-M) [9, 10]



Tessellation Edge Factors: (+U/-H, +I/-J, +O/-K, +P/-L)[2, 4, 6, 8]  
Tessellation Inside Factor: (+V/-B, +N/-M) [1, 1]



**Figure 21.5** A tessellated quad with various tessellation factors.

## The Domain Shader Stage

The domain shader is invoked for each vertex coming from the tessellation stage and outputs vertices in homogeneous clip space. The data from the domain shader is passed to the geometry shader (if one exists) or to the pixel shader (if the geometry shader stage is disabled). Thus, one of the outputs from the domain shader is typically marked with the `SV_Position` semantic, just as you would have done for the vertex shader output without tessellation. [Listing 21.7](#) presents an example domain shader.

## Listing 21.7 A Domain Shader

[Click here to view code image](#)

---

```

struct DS_OUTPUT
{
    float4 Position : SV_Position;
};

[domain("tri")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float3 uvw : SV_DomainLocation, const OutputPatch<HS_OUTPUT, 3> patch)
{
    DS_OUTPUT OUT = (DS_OUTPUT)0;

    float3 objectPosition = uvw.x * patch[0].ObjectPosition.xyz
+ uvw.y
* patch[1].ObjectPosition.xyz + uvw.z *
patch[2].ObjectPosition.xyz;

    OUT.Position = mul(float4(objectPosition, 1.0f),
WorldViewProjection);

    return OUT;
}

```

---

As with the hull shader, the **domain** attribute specifies the patch topology. The first parameter to the domain shader is the output data type from the constant hull shader, and the hull shader output is available in the third parameter of type `OutputPatch<T, n>`. The second parameter describes the position of the vertex in patch space. For triangle patches, these are **barycentric coordinates**. In brief, barycentric coordinates allow any point within a triangle to be written as a weighted sum of the three triangle vertices. Specifically:

$$Position_{tri} = u * VertexPos_0 + v * VertexPos_1 + w * VertexPos_2$$

For quads, this parameter is two-dimensional (uv) and mimics texture coordinate space (range: [0,1]; u: horizontal axis; v: vertical axis). Calculating the vertex position is a bilinear interpolation. Specifically:

$$Position_0 = lerp(VertexPos_0, VertexPos_1, u)$$

$$Position_1 = lerp(VertexPos_2, VertexPos_3, u)$$

$$Position_{quad} = lerp(Position_0, Position_1, v)$$

[Listings 21.5](#), [21.6](#), and [21.7](#) put most of the pieces together for a basic tessellation demo using triangles. [Listing 21.8](#) presents the complete shader for tessellating quads (source code for both shaders is available on the book's companion website).

## Listing 21.8 A Quad Tessellation Shader

[Click here to view code image](#)

---

```
***** Resources *****
static const float4 ColorWheat = { 0.961f, 0.871f, 0.702f, 1.0f };

cbuffer CBufferPerFrame
{
    float TessellationEdgeFactors[4];
    float TessellationInsideFactors[2];
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection;
}

***** Data Structures *****
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
};

struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[4] : SV_TessFactor;
    float InsideFactors[2] : SV_InsideTessFactor;
};
```



```

HS_OUTPUT hull_shader(InputPatch<VS_OUTPUT, 4> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;

    return OUT;
}

/****** Domain Shader *****/

[domain("quad")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float2 uv : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 4> patch)
{
    DS_OUTPUT OUT;

    float4 v0 = lerp(patch[0].ObjectPosition,
patch[1].ObjectPosition,
uv.x);

    float4 v1 = lerp(patch[2].ObjectPosition,
patch[3].ObjectPosition,
uv.x);

    float4 objectPosition = lerp(v0, v1, uv.y);

    OUT.Position = mul(float4(objectPosition.xyz, 1.0f),
WorldViewProjection);

    return OUT;
}

/****** Pixel Shader *****/

float4 pixel_shader(DS_OUTPUT IN) : SV_Target
{
    return ColorWheat;
}

/****** Techniques *****/

technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetHullShader(CompileShader(hs_5_0, hull_shader()));
    }
}

```

```

        SetDomainShader(CompileShader(ds_5_0, domain_shader()));
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}

```

---

In this listing, the vertex and hull shaders merely pass through the control points without modification. Then the constant hull shader applies the tessellation factors supplied by the CPU-side application. Next, the domain shader computes the tessellated position and transforms it into homogeneous clip space. Finally, the pixel shader just outputs a solid color for each pixel (because the demo application renders the model in wireframe mode).

## A Basic Tessellation Demo

No particularly complicated C++ code is required to integrate the triangle and quad tessellation shaders from the last few sections. In a nutshell, you create the associated materials to integrate the two shaders, build vertex buffers for a single triangle and quad, and store some class members for the tessellation factors. In the demo, you can toggle between quad and triangle tessellation and modify the tessellation factors either uniformly or independently. Drawing a tessellated object follows the same patterns previously established but requires that you specify one of the control point patch list topologies. You use D3D11\_PRIMITIVE\_TOPOLOGY\_3\_CONTROL\_POINT\_PATCHLIST for triangles and D3D11\_PRIMITIVE\_TOPOLOGY\_4\_CONTROL\_POINT\_PATCHLIST for quads. [Listing 21.9](#) presents the core rendering code for the demo.

### [Listing 21.9](#) Draw Code for the Basic Tessellation Demo

[Click here to view code image](#)

---

```

ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
direct3DDeviceContext->RSSetState(RasterizerStates::Wireframe);

if (mShowQuadTopology)
{
    direct3DDeviceContext->IASetInputLayout(mQuadInputLayout);
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_4_CONTROL_POINT_PATCHLIST);

    UINT stride = mQuadMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mQuadVertexBuffer,
&stride, &offset);

    mQuadMaterial->WorldViewProjection() << mCamera-
>ViewMatrix() *
mCamera->ProjectionMatrix();
}

```

```

    mQuadMaterial->TessellationEdgeFactors()
<< mTessellationEdgeFactors;
    mQuadMaterial->TessellationInsideFactors()
<< mTessellationInsideFactors;
    mQuadPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->Draw(4, 0);
}
else
{
    direct3DDeviceContext->IASetInputLayout(mTriInputLayout);
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_3_CONTROL_POINT_PATCHLIST);

    UINT stride = mTriMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mTriVertexBuffer,
&stride, &offset);

    std::vector<float> tessellationEdgeFactors(mTessellationEdge
Factors.begin(), mTessellationEdgeFactors.end() - 1);

    mTriMaterial->WorldViewProjection() << mCamera->ViewMatrix()
*
mCamera->ProjectionMatrix();
    mTriMaterial->TessellationEdgeFactors() <<
tessellationEdgeFactors;
    mTriMaterial->TessellationInsideFactor()
<< mTessellationInsideFactors[0];
    mTriPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->Draw(3, 0);
}

```

---

## Displacing Tessellated Vertices

The idea of tessellation is to yield high-fidelity rendering from low-fidelity models. But having a triangle or a quad subdivided on the same plane doesn't provide that detail; a subdivided plane is still a plane. You have to displace the newly created vertices before your work can bear fruit. This involves the same concepts from [Chapter 9, “Normal Mapping and Displacement Mapping”](#): a texture for the displacement values and a scale factor. [Listing 21.10](#) presents a shader for tessellating a quad aligned on the xz-plane and displacing the vertices' y-position according to the displacement map.

**Listing 21.10** The QuadHeightmapTessellation.fx Effect

[Click here to view code image](#)

---

```
***** Resources *****/
static const float4 ColorWheat = { 0.961f, 0.871f, 0.702f, 1.0f
};

cbuffer CBufferPerFrame
{
    float TessellationEdgeFactors[4];
    float TessellationInsideFactors[2];
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection;
    float4x4 TextureMatrix;

    float DisplacementScale;
}

Texture2D Heightmap;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate: TEXCOORD;
};

struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate: TEXCOORD;
};

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[4] : SV_TessFactor;
    float InsideFactors[2] : SV_InsideTessFactor;
```



```

[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("constant_hull_shader")]
HS_OUTPUT hull_shader(InputPatch<VS_OUTPUT, 4> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;
    OUT.TextureCoordinate =
patch[controlPointID].TextureCoordinate;

    return OUT;
}

/***************** Domain Shader *****/
[domain("quad")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float2 uv : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 4> patch)
{
    DS_OUTPUT OUT;

    float4 v0 = lerp(patch[0].ObjectPosition,
patch[1].ObjectPosition,
uv.x);
    float4 v1 = lerp(patch[2].ObjectPosition,
patch[3].ObjectPosition,
uv.x);
    float4 objectPosition = lerp(v0, v1, uv.y);

    float2 texCoord0 = lerp(patch[0].TextureCoordinate,
patch[1].
TextureCoordinate, uv.x);
    float2 texCoord1 = lerp(patch[2].TextureCoordinate,
patch[3].
TextureCoordinate, uv.x);
    float2 textureCoordinate = lerp(texCoord0, texCoord1, uv.y);

    objectPosition.y = (2 *
Heightmap.SampleLevel(TrilinearSampler,
textureCoordinate, 0).x - 1) * DisplacementScale;

    OUT.Position = mul(float4(objectPosition.xyz, 1.0f),
WorldViewProjection);
}

```

```

        return OUT;
    }


$$\text{float4 pixel_shader(DS_OUTPUT IN) : SV_Target}$$

{
    return ColorWheat;
}


$$\text{***** Pixel Shader *****}$$



$$\text{technique11 main11}$$

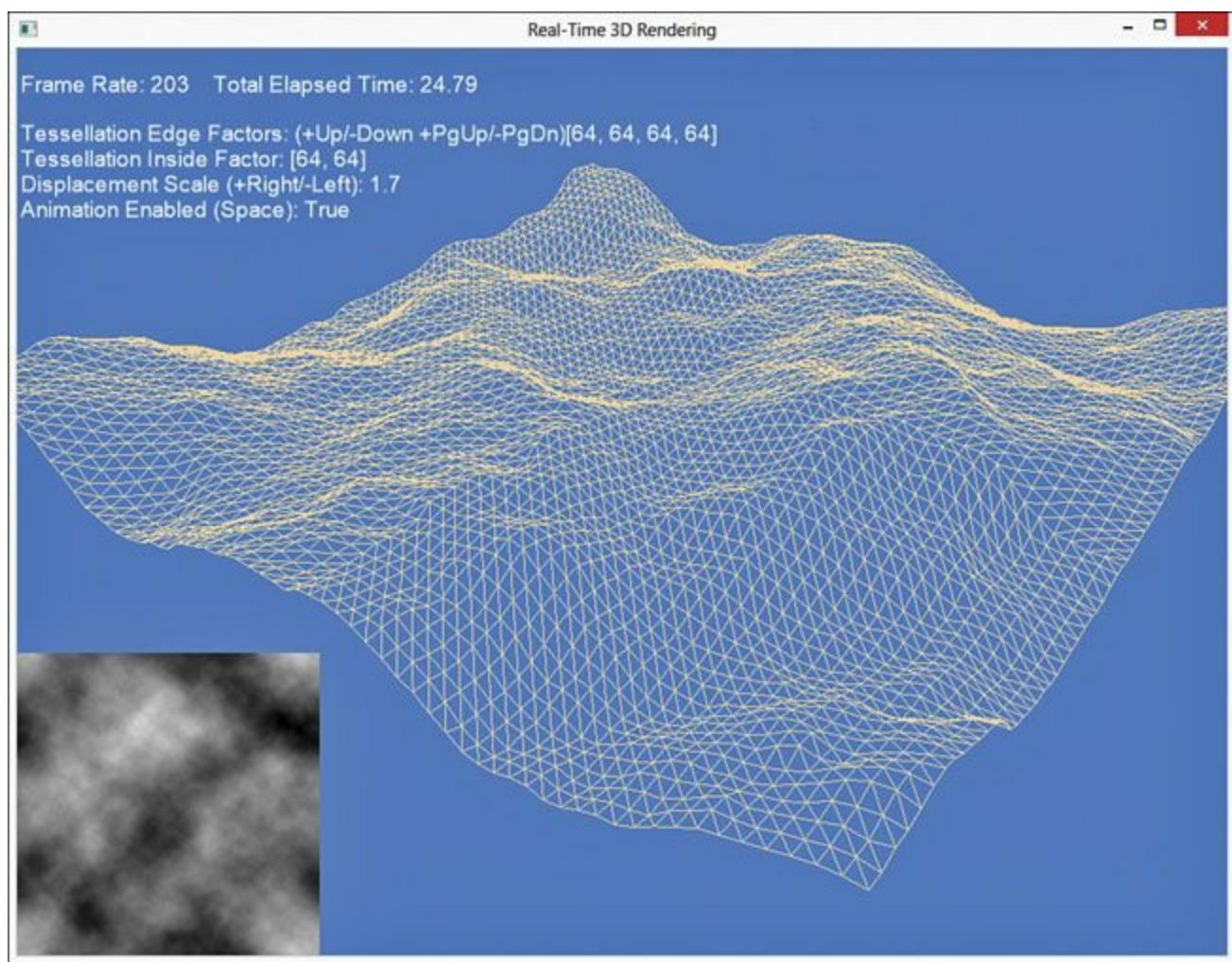
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetHullShader(CompileShader(hs_5_0, hull_shader()));
        SetDomainShader(CompileShader(ds_5_0, domain_shader()));
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}

```

---

This shader has a few interesting sections. First, observe that the texture coordinate, computed in the domain shader, is derived in the same way as the vertex position. Next, note how the displacement value is sampled and scaled to the range  $[-1, 1]$ , where the grayscale value 0.5 represents no displacement. This allows dark values (less than 0.5) in the displacement map to recess and light values (greater than 0.5) to protrude from the surface. The displacement value is further modulated by the `DisplacementScale` shader variable.

Finally, notice how the texture coordinates are transformed in the vertex shader. This allows the texture to animate. For example, if you assign the `TextureMatrix` variable to a translation matrix (with horizontal and/or vertical translation) that updates as a function of time, your tessellated quad can be made to look like undulating waves. [Figure 21.6](#) shows the output of the quad heightmap tessellation demo. The displacement map is rendered in the lower-left corner of the screen. Full source code for the demo application is available on the companion website.



**Figure 21.6** Output of the quad heightmap tessellation demo.

## Dynamic Levels of Detail

You need not manually specify tessellation factors. Instead, you can base your tessellation amounts on the distance of the surface to the camera. This provides for a dynamic LOD system. [Listing 21.11](#) presents the hull and domain shaders for such a system.

### **Listing 21.11** The Hull and Domain Shaders for a Dynamic Tessellation Effect

[Click here to view code image](#)

```
***** Resources *****  
  
cbuffer CBufferPerFrame  
{  
    float3 CameraPosition : CAMERAPOSITION;  
    int MaxTessellationFactor = 64;  
    float MinTessellationDistance = 2.0f;  
    float MaxTessellationDistance = 20.0f;  
}  
  
cbuffer CBufferPerObject  
{
```

```

    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
}

/****** Hull Shaders *****/

HS_CONSTANT_OUTPUT
distance_constant_hull_shader(InputPatch<VS_OUTPUT,
3> patch, uint patchID : SV_PrimitiveID)
{
    HS_CONSTANT_OUTPUT OUT = (HS_CONSTANT_OUTPUT) 0;

    // Calculate the center of the patch
    float3 objectCenter = (patch[0].ObjectPosition.xyz +
patch[1].
ObjectPosition.xyz + patch[2].ObjectPosition.xyz) / 3.0f;
    float3 worldCenter = mul(float4(objectCenter, 1.0f),
World).xyz;

    // Calculate uniform tessellation factor based on distance
from the
camera
    float tessellationFactor = max(min(MaxTessellationFactor,
(MaxTessellationDistance - distance(worldCenter,
CameraPosition)) /
(MaxTessellationDistance - MinTessellationDistance) *
MaxTessellationFactor), 1);
    [unroll]
    for (int i = 0; i < 3; i++)
    {
        OUT.EdgeFactors[i] = tessellationFactor;
    }

    OUT.InsideFactor = tessellationFactor;

    return OUT;
}

[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("distance_constant_hull_shader")]
HS_OUTPUT distance_hull_shader(InputPatch<VS_OUTPUT, 3> patch,
uint controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT) 0;
}

```

```

        OUT.ObjectPosition = patch[controlPointID].ObjectPosition;
        OUT.TextureCoordinate =
patch[controlPointID].TextureCoordinate;

    return OUT;
}

/***************** Domain Shader *****/
[domain("tri")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float3 uvw : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 3> patch)
{
    DS_OUTPUT OUT = (DS_OUTPUT)0;

    float3 objectPosition = uvw.x * patch[0].ObjectPosition.xyz
+ uvw.y
* patch[1].ObjectPosition.xyz + uvw.z *
patch[2].ObjectPosition.xyz;
    float2 textureCoordinate = uvw.x *
patch[0].TextureCoordinate
+ uvw.y * patch[1].TextureCoordinate + uvw.z * patch[2].TextureCoordinate;

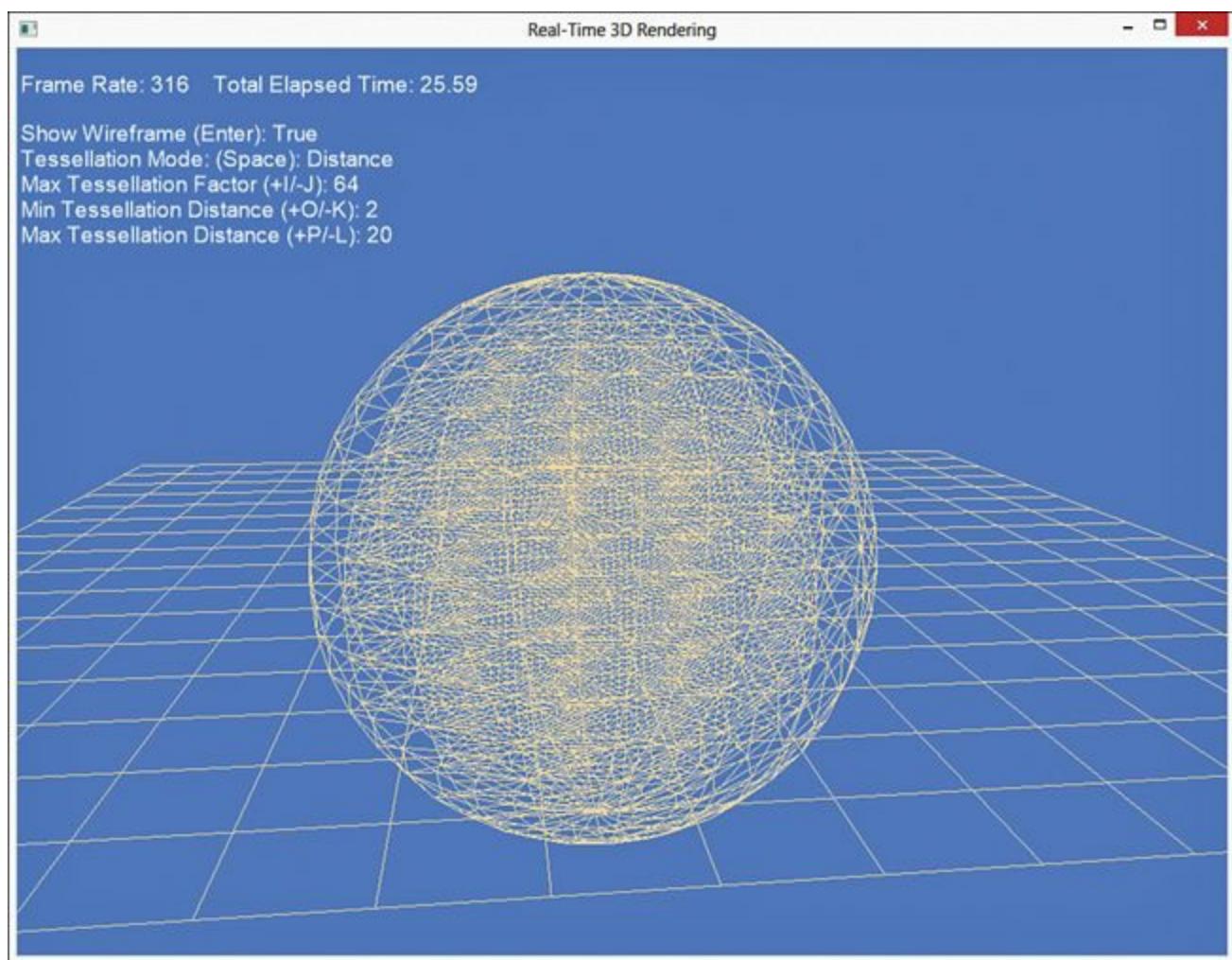
    OUT.Position = mul(float4(objectPosition, 1.0f),
WorldViewProjection);
    OUT.TextureCoordinate = textureCoordinate;

    return OUT;
}

```

---

The constant hull shader first calculates the center of the triangle patch and then transforms the position into world space. Next, a uniform tessellation factor is calculated using the maximum and minimum distances (the range) for tessellation. When the object is beyond the maximum distance, the tessellation factor is clamped at 1 (no tessellation). When the object is within the minimum distance, the tessellation factor is clamped at the passed-in `MaxTessellationFactor` variable. Between the minimum and maximum distances, the tessellation factor falls off as the distance increases. [Figure 21.7](#) shows the output of this system against a sphere. As you can see, the surfaces closer to the camera are more tessellated than those farther away.



**Figure 21.7** Output of the dynamic tessellation demo.

### Note

For performance reasons, it's best to disable the tessellation stages when the object is far enough away that no tessellation will occur. You can do this by testing the distance between the camera and the object's bounding volume on the CPU and switching between shader techniques.

## Summary

In this chapter, you discovered geometry and tessellation shaders. You implemented a point sprite system to demonstrate geometry shaders and a number of applications for hardware tessellation.

## Exercises

1. Experiment with geometry shaders. Modify the point sprite demo to output different vertex topologies.
2. Explore the three tessellation demos on the book's companion website. For the basic tessellation shader demo (whose output is shown in [Figures 21.4](#) and [21.5](#)), vary the edge and inside tessellation factors uniformly and nonuniformly, and observe the results. For the dynamic level-of-detail tessellation demo, experiment with the max tessellation factor and minimum and maximum tessellation distances, and observe the results. For the heightmap tessellation demo,

alter the heightmap and the transformation for animating the UVs, and observe the results.

3. Modify the dynamic level-of-detail tessellation shader to use an icosaheron model (included on the companion website). Run the demo and observe how the shape is not made more spherical as tessellation increases. Then modify the associated domain shader to displace the vertices along a unit sphere (to normalize the computed object position), and observe the results.

# Chapter 22. Additional Topics in Modern Rendering

This is the final chapter in the book, and it's intended to provide a look at what's next for a budding graphics developer. As such, the material is presented a little differently. Instead of taking a deep dive on a particular technique, this chapter touches on several topics in modern rendering. In particular, you examine options to improve the performance of your rendering applications and discover ways to improve the quality of your scenes through deferred shading and global illumination. You also learn about compute shaders and data-driven engine architecture.

## Rendering Optimization

You can optimize rendering speed using many approaches, but they can be roughly categorized into two groups:

1. Optimizations on the CPU (before data is sent to the GPU)
2. Optimizations on the GPU (shader optimization)

The overarching goal is to reduce the time it takes from when your application begins rendering to when your scene appears on the screen. Typically, the more objects are processed by the rendering pipeline, the longer the time to render. Thus, a common task in rendering optimization is to reduce the number of objects processed by the rendering pipeline. This means culling objects that are not visible to the camera and sorting objects so that they can be optimally rendered by the GPU. Furthermore, it's important to reduce the overhead of the rendering API and minimize the shader instructions that are executed within each stage of the pipeline.

Myriad approaches exist for each of these topics. The next few sections present just a few optimization options that are commonly used in modern rendering.

## View Frustum Culling

Ideally, objects outside the camera's view frustum should not be passed to the GPU for rendering. These objects have no chance of being written to the render target; they clog up the graphics bus and make their way at least through the early pipeline stages before being discarded. The process of view frustum culling tests the objects in the scene for collision with the view frustum. If an object collides with the frustum, it's sent to the GPU; otherwise, it's culled.

A simple approach to testing for frustum collisions is to walk the complete list of objects in the scene. But this quickly becomes unwieldy for scenes with many objects. So the real task becomes **pruning** the list of objects that will be tested for frustum collision. **Bounding volume hierarchies** (BVHs) describe a family of data structures commonly used for pruning. A BVH is constructed as a preprocessing step and organizes the list of objects into a hierarchy of bounding volumes. For example, a binary tree could be constructed as a hierarchy of bounding spheres. Each node within the hierarchy contains the smallest bounding sphere that is large enough to contain all its objects. A leaf node is represented by a single object (and its bounding sphere). A nonleaf node contains two child nodes, each consisting of half the objects of its parent node. Pruning the list of objects to render is a matter of traversing the tree and testing the bounding sphere of each node for intersection with the view frustum. If a node's bounding sphere does not intersect with the view frustum, neither will any

of its children, and that entire branch can be rejected from further processing. Leaf node objects whose bounding spheres intersect with the frustum are then sent to the GPU.

This approach is not restricted to bounding spheres. You might choose **axis-aligned bounding boxes** (AABBs) or **oriented bounding boxes** (OBBs) to encompass the objects. Trees with these bounding volumes are commonly referred to as AABB-trees, OBB-trees, and Sphere-trees. Tradeoffs come into play between the intersection cost and the bounding volume's **tightness of fit**. For example, spheres have the quickest intersection computation but often have the worst fit. You get progressively better fit (sphere, AABB, OBB, low-poly hull) with progressively higher intersection cost.

Whereas bounding volume hierarchies subdivide the objects within a scene, **space partitioning systems** subdivide the space in which the objects reside. Space partitioning systems divide a space into two or more nonoverlapping subsets. Each object within the scene is assigned into exactly one of these regions. As in the previous example, these regions can be organized into a tree, called a **space-partitioning tree**. A **binary-space partitioning tree** (or BSP tree), for example, assigns objects to either side of a plane, applying the partitioning scheme recursively to some maximum tree depth. BSP trees are one of the more common forms of space partitioning. Two other common space partitioning systems are **quadtrees** and **octrees**.

A quadtree recursively subdivides a space into four quadrants (typically square or rectangular). Objects are assigned to a quadrant until a maximum capacity is reached, at which point the quadrant is subdivided (typically to some maximum depth). Nodes are pruned when the space they encompass does not intersect with the view frustum.

An octree is a three-dimensional version of a quadtree, subdividing its space into eight octants (commonly cubes). However, this is not meant to suggest that quadtrees cannot be used for three-dimensional scenes. Indeed, a quadtree is preferable in a game in which most of the objects reside in the same plane (the  $xz$ -plane, for instance). Octrees consume more memory and are more costly to build and traverse than quadtrees. But if your scene is set, for example, in outer space, and you have objects distributed more or less uniformly throughout the space, the improved culling will mitigate the additional overhead of an octree. This hints that the distribution of the objects within the scene should dictate the data structure used for organization. Quadtrees and octrees are uniform space-partitioning systems; they divide the space uniformly. But if your scene is not uniformly distributed, you end up with a lot of empty regions (wasted space). In such scenarios, you might opt for a nonuniform space-partitioning system, such as **kD-tree**.

All these systems have tradeoffs, including their implementation complexity, memory requirements, and costs to update. These must be offset by culling performance for a positive impact. The choice of culling system is largely based on *a priori* knowledge of the application. Are the objects in the scene evenly distributed or clustered together? Are they arranged more or less within the same plane or do they have a vertical component? Furthermore, will you include dynamic objects within your tree, or just static meshes? Dynamic objects require updating the data structure; this could be prohibitively expensive, depending on the complexity of the scene. Ultimately, you have to experiment within your application to determine the best data structure to use.

## Occlusion Culling

You've already been introduced to backface culling, or not rendering triangles that face away from the camera. And you've employed the depth buffer (or z-buffer) to reject pixels that are behind other

**pixels (z-culling).** But these systems are employed only after an object has been sent to the GPU. CPU occlusion culling rejects objects that are fully occluded by other objects, before they are ever sent to the GPU. Consider a scenario in which you are inside a building that's part of a much larger scene. The walls of the building occlude all the objects outside, which might be the majority of the geometry. Discarding those occluded objects would dramatically improve rendering performance.

Two major occlusion culling approaches exist: **potentially visible set (PVS) rendering** and **portal rendering**. PVS rendering involves subdividing the scene into regions and precomputing what objects are visible for various camera positions within each region. The runtime cost is just a lookup for the PVS set associated with the camera position. The list of visible objects in the set is sent to the GPU. Disadvantages to PVS rendering include the memory overhead for storing the PVS data, the lack of support for entirely dynamic scenes, and the preprocessing time involved in generating the PVS data. Moreover, manual configuration typically is required to establish the PVS data; this can be time consuming. Furthermore, the PVS data becomes stale and must be regenerated as the scene changes.

Portal rendering also divides the scene into regions (or zones), where each zone occludes anything outside the zone. Adjacent zones are connected via **portals**, geometry that's shared between regions and that indicates potentially visible objects beyond the zone. PVS algorithms can precompute visibility within and between zones.

## Object Sorting

Occlusion culling is intended to reduce **overdraw**, or rendering the same pixel more than once. Rendering an opaque pixel, only to have it overwritten by a pixel that's closer to the camera, is a waste of processing time. The amount of overdraw can be described as a ratio between the number of unnecessarily drawn pixels (overdrawn pixels) and the number of pixels that *should* be drawn. Ideally, this value should be zero (no overdraw). As the overdraw ratio increases, performance decreases.

Front-to-back sorting (of opaque objects) can reduce overdraw by supporting the z-buffer's attempts to reject pixels that fail the depth test. Such sorting is expensive, but the cost can be mitigated as a tradeoff with less accurate sorting.

An alternative to object sorting is to render the scene with an **early z pass**. This technique renders the scene twice: first to the depth buffer (with no pixel shader enabled) and then to the frame buffer. When the scene is rendered the second time, only the pixels nearest the camera will pass the depth test; all others will be rejected before execution of the pixel shader.

## Shader Optimization

When you've culled as many objects as you can and you've organized the draw calls for efficient z-culling, you can turn your attention to optimizing the shaders. The shaders in this book have been developed with a bent toward readability rather than performance. This section aims to provide some guidance on improving the speed of your shaders.

### ■ Know thy enemy (or, profile your code).

Several tools on the market can help you profile your shaders, but I prefer NVIDIA Nsight Visual Studio Edition. You can obtain this free tool from NVIDIA's website. Quite a bit of documentation is available, and you have access to instructional videos to get you started.

Indeed, profiling isn't simply germane to shader optimization; profiling your applications from

the CPU side is just as important. Profiling is applied to the application as a whole and is truly the first step in any optimization attempt.

## ■ Don't early optimize.

If you aren't having a performance issue, there's little call for investing valuable development time to improve rendering speed.

## ■ Pixels outnumber vertices.

Generally, fewer vertices than pixels are moving through the rendering pipeline. Thus, try to move instructions out of the pixel shader and into the vertex shader, particularly if the visual impact is negligible.

A common technique for determining whether the pixel shader is the bottleneck is to compare the performance of the application rendered to different window sizes. The larger the window, the more pixels are rendered. If your performance drops precipitously at higher resolution, it's a good bet that your pixel shader is the culprit.

## ■ Skip unnecessary instructions.

This recommendation is largely about dynamic branching. For example, if your shader supports fog, test the fog amount at the beginning of the pixel shader. If the pixel is fully fogged, you can skip the rest of the shader. This holds true for a number of "early outs," including transparency. If you're alpha blending and the pixel you're on is fully transparent, you can skip the entire pixel shader.

That said, you're more likely to be able to skip blocks of code than the whole shader. For example, if you're using Lambert's cosine law to compute a diffuse component and the light is "behind" the surface, you can skip additional processing for the diffuse and specular terms. However, you still might have an ambient term. Dynamic branching is your friend in such scenarios. Use it sparingly, but you should almost certainly prefer dynamic branching over executing unnecessary instructions.

Another example is renormalizing data unnecessarily. If the CPU-side application guarantees that a direction vector is already normalized, there's no need to do it again. Similarly, you might renormalize direction vectors in the pixel shader because the data coming out of the interpolator could be slightly off. However, the visual artifacts might be small enough that you can skip the extra normalization.

## ■ Keep your shaders simple.

The more complex your shaders, the slower they are to execute.

## ■ Examine the assembly code.

You can ask the HLSL compiler to output the assembly instructions built for each of your shaders. To do so, open the Property Pages for your project and edit the **Additional Options** field under **Configuration Properties**, **HLSL Compiler**, **Command Line** to add the following option:

[Click here to view code image](#)

```
/Fc"$(OutDir)Content\Effects\% (Filename).asm"
```

[Listing 22.1](#) shows the assembly from the release build of the `BlinnPhongIntrinsic.fx` pixel shader.

## Listing 22.1 Assembly Code for the Pixel Shader from BlinnPhongIntrinsics.fx

[Click here to view code image](#)

---

```
PixelShader = asm {
    ps_5_0
    dcl_globalFlags refactoringAllowed
    dcl_constantbuffer cb0[2], immediateIndexed
    dcl_constantbuffer cb1[10], immediateIndexed
    dcl_sampler s0, mode_default
    dcl_resource_texture2d (float,float,float,float) t0
    dcl_input_ps linear v1.xyz
    dcl_input_ps linear v2.xy
    dcl_input_ps linear v3.xyz
    dcl_input_ps linear v4.xyz
    dcl_output o0.xyzw
    dcl_temps 3
    dp3 r0.x, v4.xyzz, v4.xyzz
    rsq r0.x, r0.x
    dp3 r0.y, v3.xyzz, v3.xyzz
    rsq r0.y, r0.y
    mul r0.yzw, r0.yyyy, v3.xxyz
    mad r1.xyz, v4.xyzz, r0.xxxx, r0.yzwy
    dp3 r0.x, r1.xyzz, r1.xyzz
    rsq r0.x, r0.x
    mul r1.xyz, r0.xxxx, r1.xyzz
    dp3 r0.x, v1.xyzz, v1.xyzz
    rsq r0.x, r0.x
    mul r2.xyz, r0.xxxx, v1.xyzz
    dp3 r0.x, r2.xyzz, r1.xyzz
    dp3 r0.y, r2.xyzz, r0.yzwy
    ge r0.zw, r0.xyyy, l(0.000000, 0.000000, 0.000000, 0.000000)
    log r0.x, r0.x
    mul r0.x, r0.x, cb1[9].x
    exp r0.x, r0.x
    max r0.y, r0.y, l(0.000000)
    and r0.z, r0.w, r0.z
    and r0.x, r0.x, r0.z
    sample_indexable(texture2d) (float,float,float,float)
    r1.xyzz,
    v2.xyzz, t0.xyzz, s0
    min r0.x, r0.x, r1.w
    mul r0.yzw, r0.yyyy, r1.xxyz
    mul r2.xyz, cb0[1].wwww, cb0[1].xyzz
    mul r0.yzw, r0.yyzw, r2.xxyz
    mul r2.xyz, cb0[0].wwww, cb0[0].xyzz
    mad r0.yzw, r2.xxyz, r1.xxyz, r0.yyzw
```

```

mul r1.xyz, cb1[8].wwww, cb1[8].xyzx
mad o0.xyz, r1.xyzx, r0.xxxx, r0.yzwy
mov o0.w, l(1.000000)
ret
// Approximately 32 instruction slots used
};

```

---

By examining the assembly, you can see exactly what instructions are generated from your code and refactor sections that are suspect. That said, the driver processes these instructions further for the actual graphics card. You need a vendor-specific application to view the instructions a particular driver produces.

### ■ Use intrinsic functions.

We discussed this concept in [Chapter 6](#), “[Lighting Models](#),” when we introduced the HLSL intrinsic `lit()` function to compute the diffuse and specular coefficients. Using intrinsic functions yields fewer instructions and has better performance than rolling the same code yourself.

### ■ Don’t perform calculations on otherwise constant data.

Recall the light intensity multiplications performed in shaders throughout the text (for example, `sampledColor.rgb * ambientColor.rgb * ambientIntensity.a`). The `ambientColor.rgb * ambientColor.a` calculation results in a constant value for each pixel rendered by the draw call. Such calculations are better done on the CPU and passed into the shader as a constant. [Chapter 6](#) briefly mentioned this, but it is worth emphasizing here. The book’s shaders were authored this way, to make interaction simple within NVIDIA FX Composer.

This section just scratches the surface of a much larger topic, but hopefully these few guidelines will point you in the right direction.

## Hardware Instancing

The final optimization topic we discuss is hardware instancing, or rendering multiple instances of the same mesh data through a single draw call. It’s common to duplicate geometry throughout a scene. However, while each instance shares the same geometry, an instance has (at least) a unique world matrix and possibly other attributes that differentiate the object from other copies. Without hardware instancing, you can render copies of the same mesh by issuing multiple draw calls against the same vertex buffer. And for each call, you update the shader constants specific to each object. Although that works, each draw call and material update has additional API overhead. Hardware instancing mitigates this overhead.

Configuring hardware instancing involves augmenting the data sent to the input-assembler stage with *per-instance* variables. [Listing 22.2](#) presents a hardware instancing version of the point light shader. Notice that the `VS_INPUT` structure now contains members for the object’s world matrix, specular power, and specular color.

## Listing 22.2 An Instancing Point Light Shader

[Click here to view code image](#)

```
#include "include\Common.fhx"

/****** Resources *****/

cbuffer CBufferPerFrame
{
    float4 AmbientColor = {1.0f, 1.0f, 1.0f, 0.0f};
    float4 LightColor = {1.0f, 1.0f, 1.0f, 1.0f};
    float3 LightPosition = {0.0f, 0.0f, 0.0f};
    float LightRadius = 10.0f;
    float3 CameraPosition : CAMERAPOSITION;
}

cbuffer CBufferPerObject
{
    float4x4 ViewProjection : VIEWPROJECTION;
}

Texture2D ColorTexture;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/****** Data Structures *****/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
    row_major float4x4 World : WORLD;
    float4 SpecularColor : SPECULARCOLOR;
    float SpecularPower : SPECULARPOWER;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD;
    float3 WorldPosition : WORLD;
    float Attenuation : ATTENUATION;
};
```

```

    float4 SpecularColor : SPECULAR;
    float SpecularPower : SPECULARPOWER;
};

/**************************************** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.WorldPosition = mul(IN.ObjectPosition, IN.World).xyz;
    OUT.Position = mul(float4(OUT.WorldPosition, 1.0f),
ViewProjection);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0),
IN.World).xyz);
    OUT.TextureCoordinate = IN.TextureCoordinate;

    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));
    OUT.SpecularColor = IN.SpecularColor;
    OUT.SpecularPower = IN.SpecularPower;

    return OUT;
}

/**************************************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition -
IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection +
viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h,

```

```

IN.SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular =
get_scalar_color_contribution(IN.SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

/**************************************** Techniques *****/
technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}

```

---

The new members to the VS\_INPUT structure replace global shader constants and are now used in the vertex and pixel shaders to transform the object's position and compute the specular term. Note also that the world matrix component has been removed from the global ViewProjection constant. The ViewProjection matrix will be constant across instances, but the world matrix now comes from the per-instance data.

On the CPU side, you must update your input layout to match the new input signature. In contrast with the D3D11\_INPUT\_PER\_VERTEX\_DATA enumeration you've used thus far, your D3D11\_INPUT\_ELEMENT\_DESC structures use the D3D11\_INPUT\_PER\_INSTANCE\_DATA enumeration for per-instance data elements. The following code shows the array of input element descriptions used for the shader in [Listing 22.2](#):

[Click here to view code image](#)

```

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
D3D11_INPUT_
PER_VERTEX_DATA, 0 },

```

```

    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 16,
D3D11_INPUT_PER_
VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 24,
D3D11_INPUT_PER_
VERTEX_DATA, 0 },
    { "WORLD", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0,
D3D11_INPUT_PER_
INSTANCE_DATA, 1 },
    { "WORLD", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16,
D3D11_INPUT_PER_
INSTANCE_DATA, 1 },
    { "WORLD", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32,
D3D11_INPUT_PER_
INSTANCE_DATA, 1 },
    { "WORLD", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48,
D3D11_INPUT_PER_
INSTANCE_DATA, 1 },
    { "SPECULARCOLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 64,
D3D11_INPUT_PER_
INSTANCE_DATA, 1 },
    { "SPECULARPOWER", 0, DXGI_FORMAT_R32_FLOAT, 1, 80,
D3D11_INPUT_PER_
INSTANCE_DATA, 1 }
} ;

```

Some additional constructs are noteworthy in these input element descriptions. Notice the four WORLD elements, each with  $4 \times 32$ -bit floating point values; they are identified with the semantic indices 0, 1, 2, and 3 (that's the second field of the D3D\_INPUT\_ELEMENT\_DESC structure). This is how the 16 floats in the  $4 \times 4$  world matrix shader input are referenced. Also note that all the per-instance elements are passed to the input-assembler stage using slot 1 (the field immediately after the format specification. The IA stage has 16 input slots that accommodate up to 16 vertex buffers in a single draw call. You can store your instance data in a vertex buffer that's separate from the buffer storing geometry data. In this way, this *instance buffer* can be updated independently. Finally, note that the last field (the InstanceDataStepRate member) is specified with the value 1 for each per-instance element. This is the number of instances to draw using the same per-instance data before advancing to the next element in the instance buffer.

[Listing 22.3](#) presents the initialization and draw calls for the hardware instancing demo. You can find the full source code on the companion website.

### Listing 22.3 Initialization and Draw Calls for the Hardware Instancing Demo

[Click here to view code image](#)

---

```

void InstancingDemo::Initialize()
{
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());
}

```

```

    std::unique_ptr<Model> model(new Model(*mGame,
"Content\Models\Sphere.obj", true));

    // Initialize the material
    mEffect = new Effect(*mGame);
    mEffect-
>LoadCompiledEffect(L"Content\Effects\Instancing.cso");
    mMaterial = new InstancingMaterial();
    mMaterial->Initialize(*mEffect);

    // Create vertex buffer
    Mesh* mesh = model->Meshes().at(0);
    ID3D11Buffer* vertexBuffer = nullptr;
    mMaterial->CreateVertexBuffer(mGame->Direct3DDevice(),
*mesh,
&vertexBuffer);
    mVertexBuffers.push_back(new VertexBufferData(vertexBuffer,
mMaterial->VertexSize(), 0));

    // Create instance buffer
    std::vector<InstancingMaterial::InstanceData> instanceData;
    UINT axisInstanceCount = 5;
    float offset = 20.0f;
    for (UINT x = 0; x < axisInstanceCount; x++)
    {
        float xPosition = x * offset;

        for (UINT z = 0; z < axisInstanceCount; z++)
        {
            float zPosition = z * offset;

            instanceData.push_back(InstancingMaterial::InstanceDa
XMMatrixTranslation(-xPosition, 0, -zPosition),
ColorHelper::ToFloat4(mSpecularColor), mSpecularPower));
            instanceData.push_back(InstancingMaterial::InstanceDa
XMMatrixTranslation(xPosition, 0, -zPosition),
ColorHelper::ToFloat4(mSpecularColor), mSpecularPower));
        }
    }

    ID3D11Buffer* instanceBuffer = nullptr;
    mMaterial->CreateInstanceBuffer(mGame->Direct3DDevice(),
instanceData, &instanceBuffer);
    mInstanceCount = instanceData.size();
    mVertexBuffers.push_back(new

```

```

VertexBufferData(instanceBuffer,
mMaterial->InstanceSize(), 0));

// Create index buffer
mesh->CreateIndexBuffer(&mIndexBuffer);
mIndexCount = mesh->Indices().size();

std::wstring textureName =
L"Content\\Textures\\EarthComposite.
jpg";
HRESULT hr = DirectX::CreateWICTextureFromFile(mGame-
>Direct3DDevice(), mGame->Direct3DDeviceContext(),
textureName.c_str(),
nullptr, &mColorTexture);
if (FAILED(hr))
{
    throw GameException("CreateWICTextureFromFile()
failed.", hr);
}

mPointLight = new PointLight(*mGame);
mPointLight->SetRadius(500.0f);
mPointLight->SetPosition(5.0f, 0.0f, 10.0f);

mKeyboard = (Keyboard*)mGame->Services().GetService
(Keyboard::TypeIdClass());
assert(mKeyboard != nullptr);

mProxyModel = new ProxyModel(*mGame, *mCamera,
"Content\\Models\\
PointLightProxy.obj", 0.5f);
mProxyModel->Initialize();
}

void InstancingDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mMaterial->CurrentTechnique()->Passes().at(0);
    ID3D11InputLayout* inputLayout = mMaterial->InputLayouts() .
at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);
}

```

```

    ID3D11Buffer* vertexBuffers[2] = { mVertexBuffers[0]-
>VertexBuffer,
mVertexBuffers[1]->VertexBuffer } ;
    UINT strides[2] = { mVertexBuffers[0]->Stride,
mVertexBuffers[1]
->Stride } ;
    UINT offsets[2] = { mVertexBuffers[0]->Offset,
mVertexBuffers[1]
->Offset } ;

    direct3DDeviceContext->IASetVertexBuffers(0, 2,
vertexBuffers,
strides, offsets);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer,
DXGI_FORMAT
R32_UINT, 0);

    mMaterial->ViewProjection() << mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mMaterial->AmbientColor() << XMLoadColor(&mAmbientColor);
    mMaterial->LightColor() << mPointLight->ColorVector();
    mMaterial->LightPosition() << mPointLight->PositionVector();
    mMaterial->LightRadius() << mPointLight->Radius();
    mMaterial->ColorTexture() << mColorTexture;
    mMaterial->CameraPosition() << mCamera->PositionVector();

    pass->Apply(0, direct3DDeviceContext);

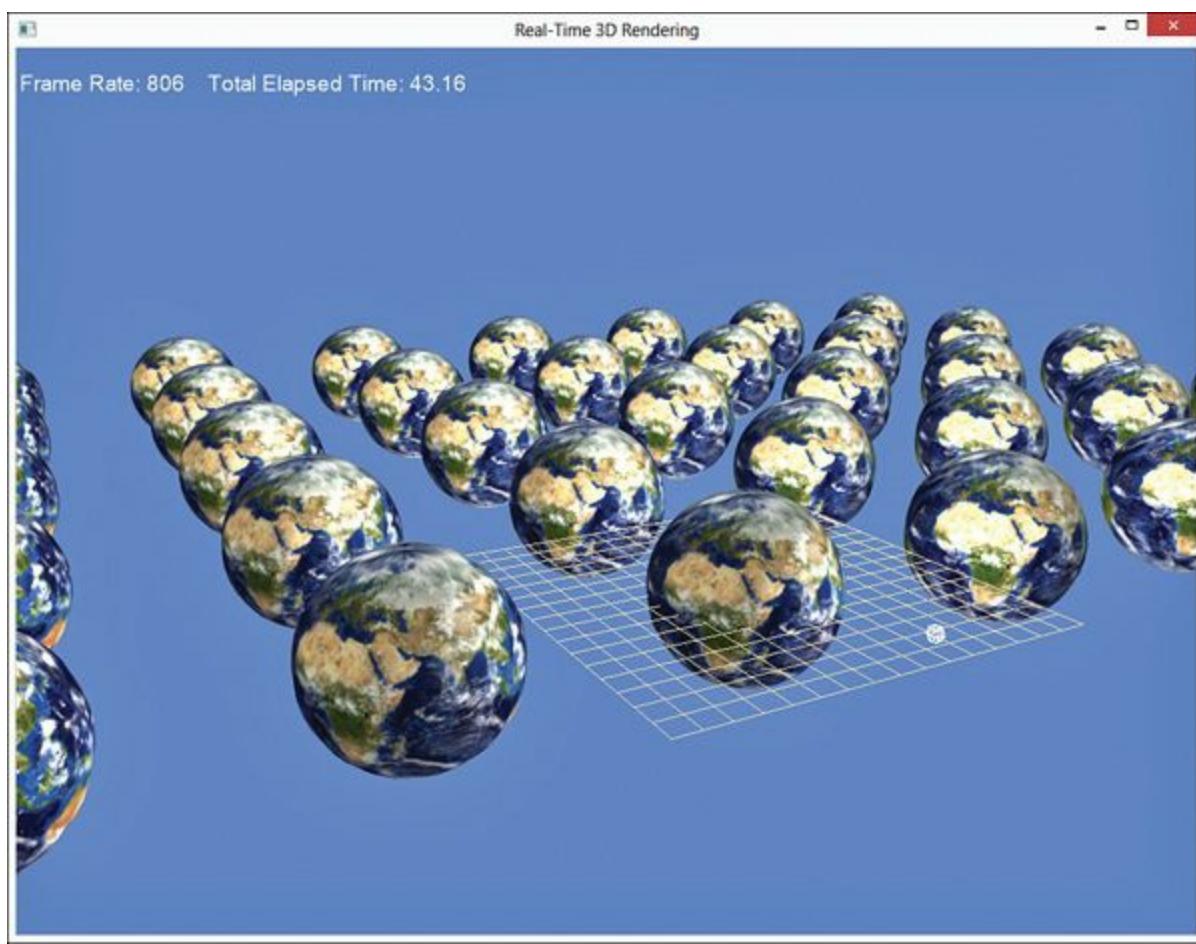
    direct3DDeviceContext->DrawIndexedInstanced(mIndexCount,
mInstanceCount, 0, 0, 0);

    mProxyModel->Draw(gameTime);
}

```

---

This code creates a grid of spheres along the  $xz$ -plane and builds a vertex buffer from the instance data to be used in the draw call. Note that two buffers are specified in the call to `ID3D11DeviceContext::IASetVertexBuffers()`. The draw method now invokes `ID3D11DeviceContext::DrawIndexInstanced()` with the index count of the shared object and the number of instances to draw. The demo's `Update()` method could modify the values within the instance buffer or add/remove instances. [Figure 22.1](#) shows the output of the hardware instancing demo.



**Figure 22.1** Output of the hardware instancing demo. (*Original texture from Reto Stöckli, NASA Earth Observatory. Additional texturing by Nick Zuccarello, Florida Interactive Entertainment Academy.*)

## Deferred Shading

The rendering you've done in this book has been **forward rendering**—essentially, rendering each object in the scene separately, using lights, textures, and shadows to determine the final color of the pixels associated with an object. You've been exposed to the concepts of single-pass rendering (all the lighting for an object is computed in a single shader) and multipass rendering (the final pixel values of an object are composited through multiple draw calls). In any forward rendering system, you are limited in the number of lights that can impact an object. The more lights, the more work the system has to do, and the slower your application will run. But another approach to rendering, called **deferred shading**, can handle a large number of lights in the scene without significant performance degradation.

### Note

Deferred shading is also known as deferred rendering. Unfortunately, this term is overloaded in graphics terminology. Microsoft, in particular, uses the term *deferred rendering* to refer to buffering commands so that they can be “played back” at some other time. This is entirely different from the deferred shading topic this section discusses.

With deferred shading, all the objects in the scene are first rendered without lighting information.

They're rendered not to the back buffer, but to several off-screen buffers using **multiple render targets** (MRT). These render targets store information about the scene's geometry, including the object's position, normal, color (typically sampled from a diffuse color texture), specular power, and specular intensity. Remember that render targets are just 2D textures—but instead of storing color, these **geometry buffers** (or G-buffers) store data that influences a second, **lighting pass** to compute the final color of all the pixels in the scene.

The lighting pass resembles a post-processing effect, such as those in [Chapter 18, “Post Processing.”](#) For each light in the scene, you sample the data from the geometry buffers and compute the direct and indirect lighting for each pixel influenced by the light. The final image is a composite of the contribution of all the lights in the scene, mapped to a full-screen quad.

Because the lighting is performed in screen space, the cost to render a frame is primarily a function of the number of lights in the scene and the number of pixels they influence. Geometry data is rendered just once per frame, and costly lighting calculations are performed only against the geometry that's actually affected. Beyond the performance advantages, deferred shading simplifies a rendering engine's architecture because all objects in the scene share exactly one shader. Considerable performance overhead is required to switch shaders, so forward rendering engines commonly batch objects with like materials. This entails a more complicated engine architecture that is absent from deferred shading systems.

That said, deferred shading has many disadvantages that still make forward rendering necessary, if not preferable. First, the geometry buffers require large amounts of memory. Second, deferred shading does not handle transparency. This is typically handled by a separate sorting step (from back to front) and a forward rendering pass of transparent objects. Another disadvantage is that, by separating the geometry and lighting stages, typical hardware anti-aliasing no longer functions properly and an additional post-processing technique must be applied to perform edge smoothing.

In the end, a modern rendering engine likely incorporates elements of both forward and deferred shading.

## Global Illumination

Global illumination refers to a set of techniques for adding more realistic light to a scene. In particular, global illumination simulates **indirect lighting**, light that reaches a surface after being reflected by other objects. Consider the ambient lighting used in this book's shaders. This term was modeled as a constant value and applied uniformly to an entire object. Clearly, this is not how ambient light actually works. Ambient light is the result of the complex inter-reflection of light between the objects in the scene; a surface that's more occluded (with respect to other reflective surfaces) receives less indirect lighting and appears darker.

A number of popular techniques can simulate indirect lighting, with tradeoffs between physical accuracy and speed. **Ambient occlusion** describes a family of rendering techniques that attempt to estimate the amount of indirect light that reaches a surface. Ambient occlusion can be pre-calculated and *baked* into textures. This is a common practice because it adds realism to the lighting with no runtime performance implications. However, such textures don't respond to dynamic lighting, and the trick can be revealed if the lighting changes or the surrounding environment is modified. A widely adopted real-time technique is known as **screen-space ambient occlusion** (SSAO). SSAO operates in screen space (as a post-processing step) and is therefore independent of scene complexity. In

particular, the scene is rendered to a depth buffer, and the depths around a pixel are sampled to compute an occlusion factor. Furthermore, SSAO can be implemented entirely on the GPU and works well with dynamic objects within the scene. A detailed discussion of SSAO is outside the scope of this text, and we leave it as a topic for the reader to explore.

Another topic in modern rendering (a full discussion of which is likewise outside the scope of this text), and one that is often used for global illumination, is **spherical harmonic lighting** (SH lighting). Quite a bit of math is involved, but SH lighting can be summarized as a set of techniques that precompute functions (such as the global lighting environment) into spherical harmonic coefficients. All our lighting techniques can be considered simplifications of the **rendering equation**, a system for generating images based entirely on physics. The rendering equation is an integral over a hemisphere of directions and is not real-time friendly. SH lighting replaces parts of the rendering equation with spherical functions. Instead of computing the rendering equation at runtime, using SH lighting, the integral can be reduced to a dot product of SH coefficients. SH lighting can produce highly realistic images in real time, but the position and form of objects within the scene must remain static or have separate sets of coefficients.

## Compute Shaders

With the release of DirectX, Microsoft introduced the **DirectCompute API**, a library supporting general-purpose programming on the GPU. **Compute shaders** enable you to move practically any computation onto the GPU, with the intent of offloading work from the CPU and leveraging the massively parallel architecture of the graphics card.

The compute shader (CS) stage is outside the normal rendering pipeline but it can read and write to GPU resources. Thus, although compute shaders are capable of supporting any number of general-purpose processes, some graphics applications are particularly interesting. For example, you can use the compute shader for the deferred shading lighting pass, deferred shading edge smoothing, depth-of-field blurring, or post-processing in general. Outputs from the compute shader can be bound as inputs to the rendering pipeline without ever transferring those outputs to the CPU.

## Threads

Compute shaders can run on multiple threads, in parallel, within a **thread group**. A thread group contains a set of threads with access to the same shared memory. Thread groups are created as a three-dimensional grid whose size is specified when you execute a compute shader through a call to `ID3D11DeviceContext::Dispatch (UINT threadGroupCountX, UINT threadGroupCountY, UINT threadGroupCountZ)`. The number of threads within a group is specified through the `numthreads (UINT x, UINT y, UINT z)` attribute attached to the compute shader. For example:

```
[numthreads(32, 32, 1)]  
void compute_shader()  
{  
    // Shader body  
}
```

The total number of threads is calculated by multiplying the number of thread groups by the number of threads per group. For example, a call to `ID3D11DeviceContext::Dispatch (32, 24,`

1) against the previous compute shader would yield  $(32 \times 32, 24 \times 32, 1 \times 1) = 1024 \times 768 \times 1$  total threads. Work is distributed among these threads according to the number of processors on the GPU.

## A Simple Compute Shader

Writing sophisticated compute shaders is beyond the scope of this section, but [Listing 22.4](#) presents a simple compute shader that writes colors to a 2D texture.

### **Listing 22.4** A Compute Shader That Writes to a 2D Texture

[Click here to view code image](#)

---

```
RWTexture2D<float4> OutputTexture;

cbuffer CBufferPerFrame
{
    float2 TextureSize;
    float BlueColor;
};

[numthreads(32, 32, 1)]
void compute_shader(uint3 threadID : SV_DispatchThreadID)
{
    float4 color = float4((threadID.xy / TextureSize),
BlueColor, 1);
    OutputTexture[threadID.xy] = color;
}

technique11 compute
{
    pass p0
    {
        SetVertexShader(NULL);
        SetGeometryShader(NULL);
        SetPixelShader(NULL);
        SetComputeShader(CompileShader(cs_5_0,
compute_shader()));
    }
}
```

---

This listing begins by declaring a read-write (RW) 2D texture (`OutputTexture`) that the compute shader writes to. This data type allows multiple threads to write to the texture simultaneously. A specific texel can be accessed through a 2D coordinate using array-style conventions.

This compute shader is declared with a single input parameter, `threadID`, marked with the `SV_DispatchThreadID` semantic. This parameter contains the unique identifier of the thread that the compute shader is executing within. The shader in [Listing 22.4](#) uses this ID as an index into the output texture. In this way, each execution of the compute shader writes a specific texel. If you do not

specify enough total threads (in this example) to cover the entire texture, some of the texels will not be written.

Note that you can declare and access constant buffer data just as you have for vertex and pixel shaders. In this example, the `TextureSize` variable is used to normalize the thread IDs (whose values are unsigned integers), and the `BlueColor` variable is used for the texel's blue channel.

In the application-side demo, an **unordered access view** (UAV) is bound to the `OutputTexture` variable. A UAV provides unordered read/write access to a resource from multiple threads. More specifically, multiple threads can simultaneously read from and write to the resource without concern for memory conflicts. [Listing 22.5](#) presents the initialization code for creating the UAV. Note that a shader resource view (SRV) is also created using the same texture and is subsequently bound to a pixel shader for rendering the output of the compute shader.

## Listing 22.5 Initializing a UAV and an SRV for the Compute Shader Demo

[Click here to view code image](#)

```
D3D11_TEXTURE2D_DESC textureDesc;
ZeroMemory(&textureDesc, sizeof(textureDesc));
textureDesc.Width = mGame->ScreenWidth();
textureDesc.Height = mGame->ScreenHeight();
textureDesc.MipLevels = 1;
textureDesc.ArraySize = 1;
textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.SampleDesc.Count = 1;
textureDesc.SampleDesc.Quality = 0;
textureDesc.BindFlags = D3D11_BIND_UNORDERED_ACCESS |
D3D11_BIND_SHADER_RESOURCE;

HRESULT hr;
ID3D11Texture2D* texture = nullptr;
if (FAILED(hr = mGame->Direct3DDevice()-
>CreateTexture2D(&textureDesc,
nullptr, &texture)))
{
    throw GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
}

D3D11_UNORDERED_ACCESS_VIEW_DESC uavDesc;
ZeroMemory(&uavDesc, sizeof(uavDesc));
uavDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
uavDesc.ViewDimension = D3D11_UAV_DIMENSION_TEXTURE2D;
uavDesc.Texture2D.MipSlice = 0;

if (FAILED(hr = mGame->Direct3DDevice()-
>CreateUnorderedAccessView
```

```

(texture, &uavDesc, &mOutputTexture) )
{
    ReleaseObject(texture);
    throw
GameException("IDXGIDevice::CreateUnorderedAccessView()
failed.", hr);
}

D3D11_SHADER_RESOURCE_VIEW_DESC resourceViewDesc;
ZeroMemory(&resourceViewDesc, sizeof(resourceViewDesc));
resourceViewDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
resourceViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
resourceViewDesc.Texture2D.MipLevels = 1;

if (FAILED(hr = mGame->Direct3DDevice()->
CreateShaderResourceView(texture, &resourceViewDesc,
&mColorTexture)))
{
    ReleaseObject(texture);
    throw GameException("IDXGIDevice::CreateShaderResourceView()
failed.", hr);
}

ReleaseObject(texture);

```

---

Note how the `mOutputTexture` (UAV) and `mColorTexture` (SRV) members refer to the same texture.

Drawing the demo is a matter of updating the compute shader's material, dispatching the compute shader, and rendering the output texture to a full-screen quad. [Listing 22.6](#) presents the demo's `Draw()` method.

## **Listing 22.6** The `Draw()` Method of the Compute Shader Demo

[Click here to view code image](#)

---

```

void ComputeShaderDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();

    // Update the compute shader's material
    mMaterial->TextureSize() << XMLoadFloat2(&mTextureSize);
    mMaterial->BlueColor() << mBlueColor;
    mMaterial->OutputTexture() << mOutputTexture;
    mComputePass->Apply(0, direct3DDeviceContext);

```

```

// Dispatch the compute shader
direct3DDeviceContext->Dispatch(mThreadGroupCount.x,
mThreadGroupCount.y, 1);

    // Unbind the UAV from the compute shader, so we can bind
    // the same underlying resource as an SRV
    static ID3D11UnorderedAccessView* emptyUAV = nullptr;
    direct3DDeviceContext->CSSetUnorderedAccessViews(0, 1,
&emptyUAV,
nullptr);

    // Draw the texture written by the compute shader
    mFullScreenQuad->Draw(gameTime);
    mGame->UnbindPixelShaderResources(0, 1);
}

void ComputeShaderDemo::UpdateRenderingMaterial()
{
    mMaterial->ColorTexture() << mColorTexture;
}

```

---

Note how the UAV is unbound from the compute shader stage through the `ID3D11DeviceContext::CSSetUnorderedAccessViews()` call. This is necessary because the same resource is being used in the full-screen quad's draw call. More specifically, the full-screen quad is attached to the `ComputeShaderDemo::UpdateRenderingMaterial()` call back, which sets the material's `ColorTexture()` member.

Within the demo, the `BlueColor` shader variable is updated each frame through the following method:

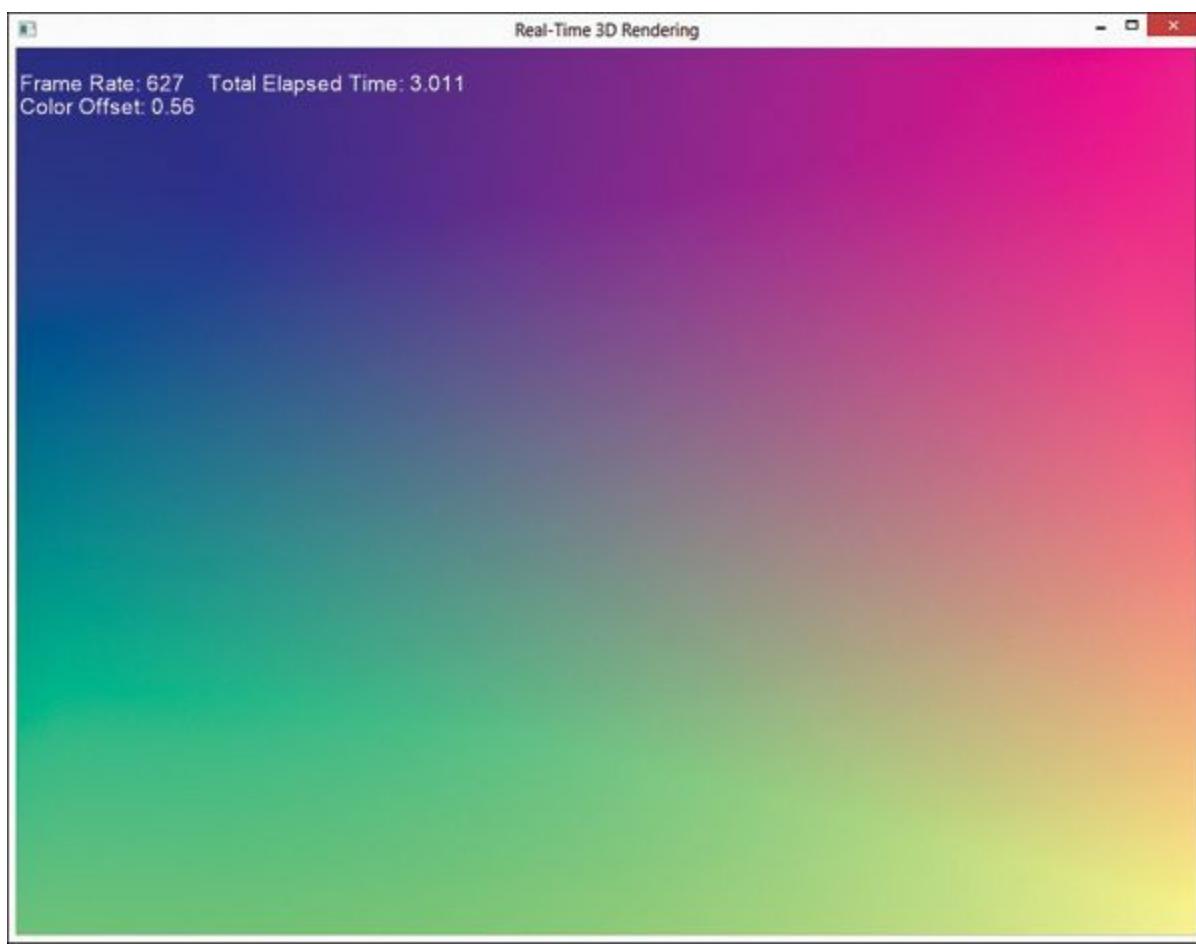
[Click here to view code image](#)

```

void ComputeShaderDemo::Update(const GameTime& gameTime)
{
    mBlueColor = (0.5f) * static_cast<float>(sin(gameTime.
TotalGameTime())) + 0.5f;
}

```

[Figure 22.2](#) shows a snapshot of the output of the compute shader demo.



**Figure 22.2** Output of the compute shader demo.

This section merely introduces compute shaders. You have an amazing amount of capability available with DirectCompute.

## Data-Driven Engine Architecture

All the demos in this book have specified their assets—the models, textures, and materials—in code. This simplified the demos but is not common practice for modern rendering engines. Instead, engines are **data driven**; their scenes are populated from configuration files. A data-driven engine implies that the game will happily execute even without any (or much) data in the world. Furthermore, modern engines incorporate not only assets from configuration files, but behavior as well. This is typically accomplished through scripting languages. The Unreal Development Kit (UDK), for example, uses UnrealScript to author practically all game code. Unity (another popular game engine) offers a choice between C#, JavaScript, and Boo for game scripting.

Making a game engine data-driven starts by defining a configuration file format and an entry point for loading assets. XML is a common format choice because the data is (at least somewhat) human readable and can be hand-crafted in any text editor. The specific definition of configuration files matches the attributes associated with runtime classes. Furthermore, configuration files are often hierarchical, with a “master” configuration file broken into myriad subfiles that define specific levels or asset types. Multiple developers can work on different sections of the game without concern over conflicting file changes. The following example presents a hypothetical XML configuration format:

[Click here to view code image](#)

```
<!-- Game.xml, the root asset store of the configuration
```

```

hierarchy -->
<AssetStore>
  <Items>
    <Item Class="AssetStore" File="Content\Levels\Level1.xml" />
    <Item Class="AssetStore" File="Content\Levels\Level2.xml" />
    <Item Class="AssetStore" File="Content\Levels\Level3.xml" />
  </Items>
</AssetStore>

<!-- Level1.xml -->
<AssetStore>
  <Items>
    <Item Class="Camera" FOV="0.785398" AspectRatio="1.33333"
NearPlaneDistance="1.0" FarPlaneDistance="1000.0">
      <Position X="0" Y="0" Z="0" />
      <Direction X="0" Y="0" Z="-1" />
      <Up X="0" Y="1" Z="0" />
    </Item>

    <Item Class="Skybox"
CubemapFile="Content\Textures\Maskonaiive2_1024.
dds" Scale="500" />
  </Items>
</AssetStore>

```

At the heart of this system is the capability to match runtime types with elements in the configuration file. This can be accomplished with the **factory pattern**, a software design pattern that allows objects to be instantiated through an interface. More specifically, your configuration files store class names used for lookup into a set of factories. Each factory knows how to instantiate a single class, and the factory manager locates the factory associated with a name.

In the previous example, a C++ AssetStore class contains a collection of Item instances. The AssetStore class represents the root configuration data structure. The Item class represents a single asset within the store, and its Class attribute identifies the C++ class to instantiate. If the item's File attribute is present, the runtime loading system should retrieve the asset's configuration from a secondary file; otherwise, the configuration data required for the class is included as attributes within the XML Item property. In this example, Camera and Skybox classes will be instantiated and configured with the associated data.

The AssetStore class interacts with a templated Factory class, whose declaration could resemble that of [Listing 22.7](#).

## **Listing 22.7** The Factory<T> Class Declaration

[Click here to view code image](#)

---

```

template <class T>
class Factory

```

```

{
public:
    virtual ~Factory() ;

    virtual const std::string ClassName() const = 0;
    virtual T* Create() const = 0;

    static Factory<T>* Find(const std::string& className);
    static T* Create(const std::string& className);

    static typename std::map<std::string, Factory<T>*>::iterator
Begin();
    static typename std::map<std::string, Factory<T>*>::iterator
End();

protected:
    static void Add(Factory<T>* const factory);
    static void Remove(Factory<T>* const factory);

private:
    static std::map<std::string, Factory<T>*> sFactories;
};

```

---

This class acts as both factory and factory manager. Each derived class implements the pure-virtual `ClassName()` and `Create()` methods, and adds/removes itself to the static list of factories through `Add()` and `Remove()`. The static `Create()` method attempts to find the factory for the given name and invokes the nonstatic `Create()` method if it finds a factory. Creating factory subclasses is a common requirement (necessary for any class you want to expose to your data-driven configuration system), so you could implement the macro in [Listing 22.8](#) for easy factory definition.

## **Listing 22.8** ConcreteFactory Class Macro

[Click here to view code image](#)

---

```

#define ConcreteFactory(ConcreteProductT,
AbstractProductT) \
class ConcreteProductT ## Factory : public \
Factory<AbstractProductT> \
{

public:
    ConcreteProductT ## \
Factory() \
{
    Add(this);
}

```

```

~ConcreteProductT ##

Factory()
{
    Remove(this);
}

    virtual const std::string ClassName()
const
{
    return std::string( #ConcreteProductT
);
}

    virtual AbstractProductT* Create()
const
{
    AbstractProductT* product = new
ConcreteProductT(); 
    return
product;
}
;

```

---

With this system in place, you would create a Skybox factory with a call such as:

[Click here to view code image](#)

```
ConcreteFactory(Skybox, RTTI)
```

And you would instantiate a Skybox with a call such as:

[Click here to view code image](#)

```
RTTI* skybox = Factory<RTTI>::Create("Skybox");
```

Populating your game world with objects becomes a matter of deserializing the configuration files, instantiating game objects through the factory system, and adding them to whatever data structure represents your scene.

## The End of the Beginning

In this chapter, you examined some modern rendering topics. You looked at optimizing the speed of your rendering applications through view frustum culling, occlusion culling, object sorting, shader optimization, and hardware instancing. You also explored deferred shading, global illumination, compute shaders, and data-driven engine architecture.

This is the final chapter of the book. Congratulations on your accomplishments! But instead of seeing this as the end, I hope you think of this as a commencement. You are now at the beginning of an exploration of even more topics in rendering. I sincerely hope you have enjoyed this book.

## Exercises

1. Install a graphics profiling tool such as NVIDIA Nsight Visual Studio Edition, and become familiar with the process of profiling your applications.
2. Implement a deferred shading architecture. The discussion in this chapter should be enough to get you pointed in the right direction. You'll need to investigate multiple render targets (MRT).
3. Implement a screen-space ambient occlusion shader.

# Index

## Numerics

3D cube, rendering, [306-313](#)

3D models

asset loading

*loading meshes*, [325-328](#)

*loading model materials*, [328-331](#)

*loading models*, [323-325](#)

content pipeline, [317](#)

*Open Asset Import Library*, [317-318](#)

meshes, [318-321](#)

model file formats, [316-317](#)

model materials, [321-323](#)

## A

AABBs (axis-aligned bounding boxes), [531](#)

Add Effect dialog box (FX Composer), [60](#)

adding vectors, [25](#)

additive blending, [159-161](#)

addressing modes, [86-88](#)

border address mode, [88](#)

clamp address mode, [87-88](#)

mirror address mode, [87](#)

wrap address mode, [86](#)

AdjustWindowRect() function, [202](#)

alpha blending, [159-161](#)

ambient lighting, [92-96](#)

AmbientColor shader constant, [94-95](#)

incrementing/decrementing intensity of, [381-382](#)

pixel shader, [95](#)

AmbientColor shader constant, [94-95](#)

AmbientLighting.fx, [92-94](#)

animated models

bind pose, retrieving, [491](#)

importing, [476-489](#)

skeletal animation, rendering, [489-495](#)

AnimationPlayer class, [489-491](#)

anisotropic filtering, [83](#)

annotations, HLSL, [77-78](#)

# APIs

Direct3D, [9](#)

DirectX, [7](#)

Open Asset Import Library, [317-318](#)

OpenGL, [9](#)

## applications

FX Composer

*Assets panel*, [49](#)

*effects*, [49](#)

*materials*, [49](#)

*Render panel*, [51-52](#)

*Textures panel*, [52](#)

“Hello, Rendering”, [284-306](#)

*Effects 11 library*, [288-293](#)

*input layout, creating*, [293-297](#)

*rendering a triangle*, [297-304](#)

*rotating the triangle*, [305-306](#)

*TriangleDemo class*, [286-287](#)

NVIDIA FX Composer, [47-52](#)

Visual Studio, [44-46](#)

Visual Studio Graphics Debugger, [53-55](#)

ApplyRotation() method, [276](#)

## asset loading

loading meshes, [325-328](#)

loading model materials, [328-331](#)

loading models, [323-325](#)

Assets panel (FX Composer), [49](#)

audience for this book, [2](#)

authoring file formats, [316](#)

AutoDesk Maya, [52](#)

automatic binding, [105](#)

# B

backface culling, [15](#)

basic effect materials, [357-364](#)

basic material demo, [361-364](#)

BasicEffect shader, [342](#)

    material, creating, [357-361](#)

bias, calculating, [465-466](#)

billboarding, [499-502](#)

bind pose, [491](#)

bitmaps, [18](#)

blend factor options, [160-159](#)

blend operation options, [159](#)

BlendState objects, [159](#)

Blinn-Phong, [111-114](#)

    BlinnPhongIntrinsics.fx, [113-114](#)

BlinnPhongIntrinsics.fx, [113-114](#)

bloom, [419-424](#)

    glow maps, creating, [420-424](#)

Bloom.fx file, [421-422](#)

blurring, Gaussian blurring, [410-418](#)

bones, skinning, [472-476](#)

border address mode, [88](#)

brightness, Lambert's cosine law, [97](#)

BVHs (bounding volume hierarchies), [530](#)

## C

calculating

    bias, [465-466](#)

    linear interpolation, [489](#)

    vector length, [26](#)

Camera class, [270-276](#)

CenterWindow() method, [203](#)

changing coordinate systems, [34-35](#)

checking for multisampling support, [208-210](#)

clamp address mode, [87-88](#)

classes

    AnimationPlayer class, [489-491](#)

    DepthMap class, [451-452](#)

    DiffuseLightingDemo class, [377-378](#)

    DiffuseLightingMaterial class, [375-377](#)

    Effect class, [342-346](#)

    FullScreenQuad class, [396-401](#)

    FullScreenRenderTarget class, [392-395](#)

    Game class, header file, [200-201](#)

    GameClock, [197](#)

    GameComponent class, [235-237](#)

    GaussianBlur class, [413-418](#)

    Material class, [352-357](#)

    Mesh class, [320-321](#)

    Model class, [319-320](#)

    ModelMaterial, [322-323](#)

Pass class, [348-349](#)  
ProxyModel class, [378](#)  
ServiceContainer class, [266-267](#)  
Skybox class, [366-368](#)  
SpriteBatch class, [243-247](#)  
SpriteFont class, [243-247](#)  
Technique class, [347-348](#)  
TextureModelDemo, [335-339](#)  
TriangleDemo class, [286-287](#)  
Variable class, [350-352](#)

color blending, [159-167](#)  
    additive blending, [159-161](#)  
    alpha blending, [159-161](#)  
    blend factor options, [160-159](#)  
    blend operation options, [159](#)  
    BlendState objects, [159](#)  
    common settings, [159](#)  
    multiplicative blending, [159-161](#)  
    TransparencyMapping.fx, [161-165](#)

color filtering  
    color inverse filter, [405-406](#)  
    demo, [403-405](#)  
    generic color filter, [408-410](#)  
    grayscale shader, [401-403](#)  
    sepia filter, [407](#)

color inverse filters, [405-406](#)

ColorFilteringGame::Draw() method, [404](#)

ColorFilteringGame::UpdateColorFilterMaterial() method, [404](#)

column-major order, [30](#)

comments, HLSL, [77](#)

common color blending settings, [159](#)

Common.fhx, [130-138](#)

companion website for this book, [4](#)

comparing Blinn-Phong and Phong models, [114](#)

compute shaders, [545-549](#)  
    threads, [545](#)

concatenating matrices, [33-34](#)

constant buffers, [62](#)

constant hull shaders, [511-512](#)

constants, AmbientColor shader constant, [94-95](#)

content pipeline, [317](#)

Open Asset Import Library, [317-318](#)

control point patch lists, [16](#)

control points, [510](#)

coordinate systems, changing, [34-35](#)

CPU (central processing unit), [9](#)

creating

depth maps, [446-452](#)

game components, [234](#)

glow maps, [420-424](#)

HelloShaders effect, [60-67](#)

*constant buffers*, [62](#)

*effect files*, [62](#)

*output*, [67](#)

*render states*, [63-64](#)

*techniques*, [66-67](#)

*vertex shader*, [64-65](#)

material for BasicEffect shader, [357-361](#)

rendering engine project, [187](#)

skybox material, [364-366](#)

specialized Game class, [228-230](#)

texture cubes, [142-144](#)

texture mapping effect, [75-81](#)

*output*, [81](#)

cross product, [27](#)

cube maps. *See* [texture cubes](#)

culling, disabling, [63](#)

customizing RTTI, [237-239](#)

cylindrical billboarding, [499-502](#)

## D

D3DX (Direct3D Extension) library, [46](#)

data-driven engine architecture, [550-552](#)

data structures, rewriting HelloShaders effect for C-style structs, [68-70](#)

data types, light data types, [372-373](#)

DDS (DirectDraw Surface), [46](#)

debugging

graphics, [55](#)

shaders, [54-55](#)

decrementing ambient light intensity, [381-382](#)

deferred shading, [543-544](#)

demos

3D cube, rendering, [306-313](#)

color filtering, [403-405](#)

diffuse lighting, [377-383](#)

Gaussian blurring, [416-418](#)

geometry shader demo, [504-506](#)

material demo, [361-364](#)

model rendering demo, [331-333](#)

point lights, [383-386](#)

spotlights, [386](#)

tessellation, [518-520](#)

texture mapping, [334-339](#)

depth maps, [435, 436-437](#)

creating, [446-452](#)

occlusion testing, [453-456](#)

depth testing, [20](#)

DepthMap class, [451-452](#)

deserialization, [317](#)

device input

  keyboard input, [249-258](#)

  mouse input, [258-264](#)

diffuse lighting, [97-105](#)

  demo, [377-383](#)

  directional lights, [97-101](#)

  Lambert's cosine law, [97](#)

  material, [373-377](#)

  pixel shader, [102-103](#)

  vertex shader, [102](#)

DiffuseLightingDemo class, [377-378](#)

DiffuseLighting.fx, [98-101](#)

DiffuseLightingMaterial class, [375-377](#)

Direct3D, [7, 9](#)

  2D texture, [216-217](#)

  initializing

*associating views to output-merger stage, [218](#)*

*checking for multisampling support, [208-210](#)*

*creating a depth-stencil view, [215-218](#)*

*creating a render target view, [214-215](#)*

*creating the device context, [206-208](#)*

*creating the swap chain, [210-214](#)*

*setting the viewport, [219](#)*

    magnification, [82-83](#)

Direct3D Graphics Pipeline

domain-shader stage, [514-518](#)

geometry shader stage, [18](#)

hull-shader stage, [510-512](#)

input-assembler stage, [10-16](#)

*index buffers*, [11](#)

*primitives*, [13-14](#)

*vertex buffers*, [10-11](#)

output-merger stage, [19-20](#)

pixel shader stage, [19](#)

rasterizer stage, [18-19](#)

tessellation stage, [512](#)

tessellation stages, [16-18](#)

vertex shader stage, [16](#)

DirectInput library

    keyboard input, [249-258](#)

    mouse input, [258-264](#)

directional lights, [97-101](#)

    intensity, [105](#)

    rotating, [382-383](#)

directory structure, rendering engine project, [186](#)

DirectX, [7](#)

    history, [8-9](#)

*DirectX 11*, [8](#)

*DirectX 8*, [8](#)

*DirectX 9*, [8](#)

*OpenGL*, [9](#)

    texture coordinates, [74](#)

DirectX Texture Tool, [143](#)

DirectXMath, [35-40](#)

    matrices, [39-40](#)

    vectors

*calling conventions*, [37](#)

*initialization functions*, [37](#)

*loading and storing*, [36](#)

*operators*, [38](#)

DirectXTK (DirectX Tool Kit), [46](#)

disabling culling, [63](#)

displacement mapping, [178-181](#)

displacing tessellated vertices, [520-523](#)

distortion mapping, [425-431](#)

    full-screen distortion shader, [425-426](#)

masking distortion shader, [427-431](#)

DLLs, Open Asset Import Library, [317-318](#)

domain-shader stage (Direct3D Graphics Pipeline), [514-518](#)

dot product, [26-27](#)

Draw() method, [379-380](#)

draw order, effect on alpha-blended objects, [166-167](#)

drawable game components, [239-240](#)

dynamic environment mapping, [153-154](#)

dynamic tessellation effect, [524-527](#)

## E

Effect class, [342-346](#)

effects, [49](#)

ambient lighting effect, [92-94](#)

*output*, [95-96](#)

Blinn-Phong, [111-114](#)

*pixel shaders*, [112](#)

bloom effect, [420-424](#)

diffuse lighting

*material*, [373-377](#)

diffuse lighting effect

*output*, [103-105](#)

*pixel shader*, [102-103](#)

*preamble*, [101-102](#)

*vertex shader*, [102](#)

displacement mapping effect, [179-181](#)

dynamic tessellation effect, [524-527](#)

environment mapping

*preamble*, [151-152](#)

*vertex shader*, [152](#)

environment mapping effect

*output*, [153](#)

fog effect, [154-157](#)

*output*, [157](#)

*pixel shader*, [157](#)

*preamble*, [157](#)

*vertex shader*, [157](#)

multiple point lights effect, [132-138](#)

*output*, [138](#)

*pixel shader*, [136-137](#)

*vertex shader*, [136-137](#)

normal mapping effect, [173-177](#)

*preamble*, [176](#)

Phong effect, [106-111](#)

*preamble*, [109](#)

PointLight.fx

*output*, [121-123](#)

*preamble*, [120](#)

skybox effect

*output*, [147-148](#)

*pixel shader*, [147](#)

*preamble*, [147](#)

*vertex shader*, [147](#)

Spotlight.fx, [125-129](#)

texture mapping effect, creating, [75-81](#)

transparency mapping effect, [165-166](#)

Effects 11 library, [46, 288-293](#)

environment mapping, [149-154](#)

dynamic environment mapping, [153-154](#)

EnvironmentMapping.fx, [149-151](#)

## F

feature levels (Direct3D), [207-208](#)

file formats

3D models, [316-317](#)

texture file formats, [46](#)

final project settings, rendering engine project, [192-193](#)

first-person camera, implementing, [277-281](#)

fog effect, [154-157](#)

*output*, [157](#)

*pixel shader*, [157](#)

*preamble*, [157](#)

forward rendering, [543](#)

frame rate component, [242-243](#)

full-screen distortion shaders, [425-426](#)

FullScreenQuad class, [396-401](#)

FullScreenRenderTarget class, [392-395](#)

functions

AdjustWindowRect(), [202](#)

lit(), [113](#)

reflect(), [152](#)

WinMain(), [194](#)

WinMain(), updating for RenderingGame class, [230-232](#)

FX Composer, [47-52](#)

  Add Effect dialog box, [60](#)

  Assets panel, [49](#)

  effects, [49](#)

  materials, [49](#)

  Render panel, [51-52](#)

  Select HLSL FX Template dialog box, [60](#)

  Textures panel, [52](#)

## G

Game class

  header file, [200-201](#)

  ServiceContainer member, adding, [267-268](#)

  specialized Game class, creating, [228-230](#)

  updating for Direct3D initialization, [220-228](#)

game components

  creating, [234](#)

  drawable game components, [239-240](#)

  frame rate component, [242-243](#)

  Game class support for, [240-248](#)

  RTTI, customizing, [237-239](#)

  SpriteBatch class, [243-247](#)

  SpriteFont class, [243-247](#)

game engine file formats, [316](#)

game loop, [195-199](#)

  initialization, [199](#)

  time-related information, [196-199](#)

*GameClock.cpp* file, [198-199](#)

Game project

  final project settings, [192-193](#)

  linking libraries, [190-191](#)

  Program.cpp file, adding, [193-194](#)

Game.cpp file, [222-227](#)

GameClock class, [197](#)

GameClock.cpp file, [198-199](#)

GameComponent class, [235-237](#)

Game::Initialization() method, [196](#)

Game::InitializeWindow() method, [201-202](#)

Game::Run() method, [203-204](#)

gaming consoles

  XBox, [8](#)

  XBox 360, [8](#)

Gaussian blurring, [410-418](#)

demo, [416-418](#)

sample offsets and weights, initializing, [415-416](#)

GaussianBlur class, [413-418](#)

general-purpose Game class, updating for Direct3D initialization, [220-228](#)

generic color filters, [408-410](#)

geometry shader stage (Direct3D Graphics Pipeline), [18](#)

geometry shaders, [498](#)

demo, [504-506](#)

graphics pipeline, hull-shader stage, [510-512](#)

point sprite shaders, [499-506](#)

primitive IDs, [507](#)

processing primitives, [498-499](#)

global illumination, [544-545](#)

glow maps, creating, [420-424](#)

GPU (graphics processing unit), [1](#), [9](#)

GPU PerfStudio 2, [55](#)

GPU skinning, [472](#)

graphics, debugging, [55](#)

graphics cards, [8](#)

grayscale filter, [401-403](#)

## H

hardware instancing, [535-542](#)

header file, Game class, [200-201](#)

“Hello, Rendering” application, [284-306](#)

Effects 11 library, [288-293](#)

input layout, creating, [293-297](#)

rendering a triangle, [297-304](#)

rotating the triangle, [305-306](#)

TriangleDemo class, [286-287](#)

HelloShaders effect

constant buffers, [62](#)

creating, [60-67](#)

effect files, [62](#)

output, [67](#)

pixel shader, [65](#)

render states, [63-64](#)

rewriting for C-style structs, [68-70](#)

techniques, [66-67](#)

vertex shader, [64-65](#)

Hello, structs, [68-70](#)  
hierarchical transformations, [470](#)  
history of DirectX, [8-9](#)  
    DirectX 11, [8](#)  
    DirectX 8, [8](#)  
    DirectX 9, [8](#)  
    HLSL, [8](#)  
    OpenGL, [9](#)  
        programmable shaders, [8](#)  
HLSL (High-Level Shading Language), [8](#)  
    annotations, [77-78](#)  
    comments, [77](#)  
    HLSL, texture objects, [78-79](#)  
    preprocessor commands, [77](#)  
    texture mapping  
        *samplers*, [78-79](#)  
        *texture coordinates*, [79-81](#)  
homogeneous coordinates, [31](#)  
hull-shader stage (graphics pipeline), [510-512](#)

## I

IDE (integrated development environment), [44](#)  
identity matrix, [31](#)  
implementing a first-person camera, [277-281](#)  
importing animated models, [476-489](#)  
include directories, rendering engine project, [189-190](#)  
incrementing ambient light intensity, [381-382](#)  
index buffers, [11](#)  
    3D cube, rendering, [306-313](#)  
indirect lighting, [544](#)  
initializing  
    Direct3D  
        *associating views to output-merger stage*, [218](#)  
        *checking for multisampling support*, [208-210](#)  
        *creating a render target view*, [214-215](#)  
        *creating the device context*, [206-208](#)  
        *creating the swap chain*, [210-214](#)  
        *setting the viewport*, [219](#)  
    game loop, [199](#)  
    Gaussian blurring sample offsets and weights, [415-416](#)  
    windows, [199-201](#)  
input layout, creating for “Hello Rendering” application, [293-297](#)

input-assembler stage (Direct3D Graphics Pipeline), [10-16](#)

  index buffers, [11](#)

  primitives, [13-14](#)

  vertex buffers, [10-11](#)

  prerequisites, [4](#)

intensity

  of ambient lights, incrementing/decrementing, [381-382](#)

  of directional lights, changing, [105](#)

interchange file formats, [316](#)

interpolation, linear interpolation, [82-83](#)

## J-K

keyboard input, [249-258](#)

keyframes, [483-486](#)

## L

Lambert's cosine law, [97](#)

left-handed coordinate systems, [24-25](#)

Library project, final project settings, [192-193](#)

light data types, [372-373](#)

lighting, SH lighting, [544](#)

lighting models

  ambient lighting, [92-96](#)

  diffuse lighting, [97-105](#)

*demo*, [377-383](#)

*material*, [373-377](#)

  directional lights, [97-101](#)

  global illumination, [544-545](#)

  multiple lights, [130-138](#)

  point lights, [116-123](#)

*demo*, [383-386](#)

*manipulating*, [385-386](#)

  specular highlights, [105-114](#)

*Blinn-Phong*, [111-114](#)

*Phong reflection model*, [105-111](#)

  spotlights, [124-129](#)

linear interpolation, calculating, [489](#)

linking libraries, rendering engine project, [190-191](#)

lit() function, [113](#)

loading

  model materials, [328-331](#)

  models, [323-325](#)

vectors, [36](#)

LODs (levels of detail), [17-18](#), [524](#)

## M

magnification, [82-83](#)

- anisotropic filtering, [83](#)

- linear interpolation, [82-83](#)

- point filtering, [82](#)

manipulating point lights, [385-386](#)

manual binding, [105](#)

masking distortion shaders, [427-431](#)

Material class, [352-357](#)

materials, [49](#), [342](#)

- basic effect materials, [357-364](#)

- diffuse lighting material, [373-377](#)

- demo*, [377-383](#)

- Effect class, [342-346](#)

- Material class, [352-357](#)

- Pass class, [348-349](#)

- skybox material, creating, [364-366](#)

- Technique class, [347-348](#)

- Variable class, [350-352](#)

matrices, [27-31](#)

- column-major order, [30](#)

- concatenation, [33-34](#)

- DirectXMath, [39-40](#)

- identity matrix, [31](#)

- multiplication, [28](#)

- row-major order, [30](#)

- subtracting, [28](#)

- transposing, [29](#)

meshes, [318-321](#)

- loading, [325-328](#)

- skinning, [472-476](#)

methods

- ApplyRotation(), [276](#)

- CenterWindow(), [203](#)

- ColorFilteringGame::Draw(), [404](#)

- ColorFilteringGame::UpdateColorFilterMaterial(), [404](#)

- Draw(), [379-380](#)

- Game::Initialization(), [196](#)

Game::InitializeWindow(), [201-202](#)

Game::Run(), [203-204](#)

Reset() method, [275](#)

Sample(), [79](#)

SetBlurAmount(), [415](#)

SetMaterial(), [397](#)

UpdateGameTime(), [196](#)

UpdateProjectionMatrix(), [276](#)

UpdateViewMatrix(), [276](#)

Microsoft Visual Studio. *See* [Visual Studio](#)

minification, [83](#)

mipmaps, [84-85](#)

mirror address mode, [87](#)

Model class, [319-320](#), [323-325](#)

model file formats, [316](#)

model materials, [321-323](#)

    loading, [328-331](#)

model rendering demo, [331-333](#)

ModelMaterial class, [322-323](#)

mouse input, [258-264](#)

MRTs (multiple render targets), [543](#)

MSAA (Multisample Anti-Aliasing), [208](#)

multiple lights, [130-138](#)

    MultiplePointLights.fx, [132-138](#)

multiple point lights effect

    output, [138](#)

    techniques, [137](#)

MultiplePointLights.fx, [132-138](#)

    preamble, [136](#)

multiplicative blending, [159-161](#)

multiplying matrices, [28](#)

## N

normal mapping, [170-177](#)

    displacement mapping, [178-181](#)

    tangent space, [171-173](#)

NormalMapping.fx, [173-177](#)

normals, [102](#)

Nsight Visual Studio Edition, [55](#)

nvDXT command-line tool, [142](#)

NVIDIA

    FX Composer, [47-52](#)

[Add Effect dialog box, 60](#)

[Select HLSL FX Template dialog box, 60](#)

Nsight Visual Studio Edition, [55](#)

## O

OBBs (oriented bounding boxes), [531](#)

object sorting, [532](#)

object space, [34](#)

occlusion, [444-445](#)

SSAO, [544](#)

occlusion culling, [532](#)

occlusion testing, [453-456](#)

octrees, [531](#)

Open Asset Import Library, [317-318](#)

    animated models, importing, [476-489](#)

    asset loading, [323-325](#)

*loading meshes, 325-328*

*loading model materials, 328-331*

OpenGL, [9](#)

optimizing rendering speed

    hardware instancing, [535-542](#)

    object sorting, [532](#)

    occlusion culling, [532](#)

    shader optimization, [533-535](#)

    view frustum culling, [530-531](#)

Orbit camera (FX Composer), [52](#)

output

    ambient lighting effect, [95-96](#)

    diffuse lighting effect, [103-105](#)

    displacement mapping effect, [180-181](#)

    environment mapping effect, [153](#)

    fog effect, [157](#)

    HelloShaders effect, [67](#)

    multiple point lights effect, [138](#)

    normal mapping effect, [177](#)

    Phong effect, [111](#)

    PointLight.fx, [121-123](#)

    skybox effect, [147-148](#)

    Spotlight.fx, [129](#)

    transparency mapping effect, [165-166](#)

output-merger stage (Direct3D Graphics Pipeline), [19-20](#)

# P

Pass class, [348-349](#)

patches, [510](#)

percentage closer filtering, [460-465](#)

peter panning, [466](#)

Phong reflection model, [105-111](#)

Phong.fx, [106-111](#)

    output, [111](#)

    preamble, [109](#)

PIX, [52](#)

pixel shader stage (Direct3D Graphics Pipeline), [19](#)

pixel shaders

    ambient lighting pixel shader, [95](#)

    Blinn-Phong effect, [112](#)

    diffuse lighting pixel shader, [102-103](#)

    environment mapping effect, [152](#)

    fog effect, [157](#)

    multiple point lights effect, [136-137](#)

    normal mapping effect, [177](#)

    Phong effect, [109-111](#)

    point light pixel shader, [120](#)

    skybox effect, [147](#)

    spotlight pixel shader, [129](#)

pixels

    blurring images, [411](#)

    minification, [83](#)

point filtering, [82](#)

point lights, [116-123](#)

    demo, [383-386](#)

    manipulating, [385-386](#)

point lists, [13](#)

point sprite shaders, [499-506](#)

PointLight.fx, [116-123](#)

    output, [121-123](#)

    preamble, [120](#)

portal rendering, [532](#)

post-processing, [391](#)

    bloom, [419-424](#)

    color filtering

*color inverse filter*, [405-406](#)

        demo, [403-405](#)

*generic color filter*, [408-410](#)

*grayscale shader*, [401-403](#)

*sepia filter*, [407](#)

distortion mapping, [425-431](#)

*full-screen distortion shader*, [425-426](#)

*masking distortion shader*, [427-431](#)

full-screen quad component, [396-401](#)

Gaussian blurring, [410-418](#)

render targets, [392-396](#)

    shadow mapping, [435](#)

preamble

    diffuse lighting effect, [101-102](#)

    displacement mapping effect, [180](#)

    environment mapping effect, [151-152](#)

    fog effect, [157](#)

    MultiplePointLights.fx, [136](#)

    normal mapping effect, [176](#)

    Phong effect, [109](#)

    PointLight.fx, [120](#)

    skybox effect, [147](#)

    Spotlight.fx, [129](#)

preprocessor commands, HLSL, [77](#)

primitive IDs, [507](#)

primitives, [13-14](#)

    adjacency data, [15](#)

    geometry shaders, [498-499](#)

processors, CPU, [9](#)

Program.cpp file, adding to Game project, [193-194](#)

programmable shaders, [8](#)

project build order, rendering engine project, [188-189](#)

project setup (rendering engine)

    directory structure, [186](#)

projection space, [35](#)

projective texture mapping, [436-456](#)

    occlusion, [444-445](#)

    projective texture coordinates, [438-439](#)

    projective texture-mapping shader, [439-442](#)

    reverse projection, [443-444](#)

ProxyModel class, [378](#)

PVS (potentially visible set) rendering, [532](#)

quadtrees, [531](#)  
quaternions, [486](#)  
rasterizer stage (Direct3D Graphics Pipeline), [18-19](#)  
reflect() function, [152](#)  
reflection  
    Blinn-Phong, [111-114](#)  
    Phong reflection model, [105-111](#)  
reflection mapping, [149-154](#)  
Render panel (FX Composer), [51-52](#)  
render states, [63-64](#)  
render target views, [396](#)  
render targets, [392-396](#)  
rendering, [1](#)  
    forward rendering, [543](#)  
optimizing  
    *hardware instancing*, [535-542](#)  
    *object sorting*, [532](#)  
    *occlusion culling*, [532](#)  
    *shader optimization*, [533-535](#)  
    *view frustum culling*, [530-531](#)  
shaders, [2](#)  
rendering engine  
    application startup, [193-194](#)  
    data driven engine architecture, [550-552](#)  
    final project settings, [192-193](#)  
    Game class, header file, [200-201](#)  
    game loop, [195-199](#)  
        *initialization*, [199](#)  
        *time-related information*, [196-199](#)  
    Game project, linking libraries, [190-191](#)  
    include directories, [189-190](#)  
    project build order, [188-189](#)  
    project creation, [187](#)  
        project setup, directory structure, [186](#)  
RenderMonkey, [48](#)  
Reset() method, [275](#)  
retrieving skinned model's bind pose, [491-492](#)  
reverse projection, [443-444](#)  
rewriting HelloShaders effect for C-style structs, [68-70](#)  
rigging, [470](#)  
right-handed coordinate systems, [24-25](#)

rotating directional lights, [382-383](#)

rotation matrix, [33](#)

row-major order, [30](#)

RTTI (Runtime Type Information), customizing, [237-239](#)

## S

Sample() method, [79](#)

samplers, [78-79](#)

SamplerState filtering options (TextureMapping.fx), [85](#)

sampling

Gaussian blurring sample offsets and weights, initializing, [415-416](#)

percentage closer filtering, [460-465](#)

texture cubes, [144](#)

scalars, [24](#)

scaling transformations, [31-32](#)

scenes, post-processing, [391](#)

full-screen quad component, [396-401](#)

render targets, [392-396](#)

Select HLSL FX Template dialog box (FX Composer), [60](#)

semantics, [62](#)

sepia filters, [407](#)

serialization, [317](#)

ServiceContainer class, [266-267](#)

adding to Game class, [267-268](#)

SetBlurAmount() method, [415](#)

SetMaterial() method, [397](#)

SH (spherical harmonic) lighting, [544](#)

shaders, [2, 59](#)

authoring tools, [48](#)

BasicEffect shader, [342](#)

*material, creating, 357-361*

compute shaders, [545-549](#)

*threads, 545*

constants, [62](#)

*AmbientColor shader constant, 94-95*

debugging, [54-55](#)

deferred shading, [543-544](#)

full-screen distortion shader, [425-426](#)

Gaussian blurring shader, [411-413](#)

geometry shaders, [18](#)

*demo, 504-506*

*point sprite shaders, 499-506*

*primitive IDs*, [507](#)

*processing primitives*, [498](#)

grayscale shader, [401-403](#)

HelloShaders effect

*constant buffers*, [62](#)

*effect files*, [62](#)

*output*, [67](#)

*render states*, [63-64](#)

*rewriting for C-style structs*, [68-70](#)

*techniques*, [66-67](#)

*vertex shader*, [64-65](#)

masking distortion shader, [427-431](#)

optimizing, [533-535](#)

pixel shaders, [19](#)

*ambient lighting pixel shader*, [95](#)

*Blinn-Phong effect*, [112](#)

*diffuse lighting pixel shader*, [102-103](#)

*environment mapping effect*, [152](#)

*fog effect*, [157](#)

*multiple point lights effect*, [136-137](#)

*normal mapping effect*, [177](#)

*Phong effect*, [109-111](#)

*point light pixel shader*, [120](#)

*skybox effect*, [147](#)

*spotlight pixel shader*, [129](#)

programmable shaders, [8](#)

projective texture-mapping shader, [439-442](#)

shadow-mapping shader, [456-459](#)

tessellation shaders, [507-509](#)

vertex shader

*diffuse lighting vertex shader*, [102](#)

*displacement mapping effect*, [180](#)

*environment mapping effect*, [152](#)

*fog effect*, [157](#)

*multiple point lights effect*, [136-137](#)

*normal mapping effect*, [177](#)

*Phong effect*, [109](#)

*point light vertex shader*, [120](#)

*spotlight vertex shader*, [129](#)

vertex shader stage (Direct3D Graphics Pipeline), [16](#)

vertex shaders, skybox effect, [147](#)

shadow acne, [456](#)  
shadow mapping, [435-436](#), [456-466](#)  
    depth maps  
        *creating*, [446-452](#)  
        *occlusion testing*, [453-456](#)  
percentage closer filtering, [460-465](#)  
projective texture mapping, [436-456](#)  
    *occlusion*, [444-445](#)  
    *projective texture coordinates*, [438-439](#)  
    *projective texture-mapping shader*, [439-442](#)  
    *reverse projection*, [443-444](#)  
slope-scaled depth biasing, [465-466](#)  
shadow-mapping shaders, [456-459](#)  
Silicon Graphics Inc., [9](#)  
skeletal animation, [469](#)  
    animated models, importing, [476-489](#)  
    animation rendering, [489-495](#)  
    bind pose, retrieving, [491](#)  
    hierarchical transformations, [470](#)  
    skinning, [472-476](#)  
skinning, [472-476](#)  
Skybox class, [366-368](#)  
skybox effect, output, [147-148](#)  
skyboxes, [145-148](#)  
    material, creating, [364-366](#)  
Skybox.fx, [145-147](#)  
slope-scaled depth biasing, [456](#), [465-466](#)  
    bias, calculating, [465-466](#)  
software services, [265-268](#)  
    ServiceContainer class, [266-267](#)  
specialized Game class, creating, [228-230](#)  
specular highlights, [105-114](#)  
    Blinn-Phong, [111-114](#)  
    Phong reflection model, [105-111](#)  
spherical billboarding, [499-502](#)  
Spotlight.fx  
    output, [129](#)  
    preamble, [129](#)  
spotlights, [124-129](#)  
    demo, [386](#)  
SpriteBatch class, [243-247](#)

SpriteFont class, [243-247](#)

SSAO (screen-space ambient occlusion), [544](#)

stencil testing, [20](#)

storing vectors, [36](#)

subtracting

- matrices, [28](#)

- vectors, [25](#)

surface normals, [102](#)

- normal mapping, [170-177](#)

swap chain, creating, [210-214](#)

## T

tangent space, [171-173](#)

Technique class, [347-348](#)

techniques

- HelloShaders effect, [66-67](#)

- multiple point lights effect, [137](#)

tessellation

- demo, [518-520](#)

- displacing tessellated vertices, [520-523](#)

- dynamic levels of detail, [524-527](#)

tessellation shaders, [507-509](#)

tessellation stages (Direct3D Graphics Pipeline), [16-18](#), [512](#)

texels, minification, [83](#)

text file formats, [316](#)

texture coordinates, [79-81](#)

- addressing modes, [86-88](#)

- border address mode*, [88](#)

- clamp address mode*, [87-88](#)

- mirror address mode*, [87](#)

- wrap address mode*, [86](#)

texture cubes, [142-144](#)

- creating, [142-144](#)

- environment mapping, [149-154](#)

- sampling, [144](#)

- skyboxes, [145-148](#)

texture file formats, [46](#)

texture filtering, [81-85](#)

- magnification

- anisotropic filtering*, [83](#)

- linear interpolation*, [82-83](#)

- point filtering*, [82](#)

minification, [83](#)

mipmaps, [84-85](#)

SamplerState filtering options (`TextureMapping.fx`), [85](#)

texture mapping, [74-75](#), [334-339](#)

DirectX texture coordinates, [74](#)

effect, creating, [75-81](#)

output, [81](#)

projective texture mapping, [436-456](#)

samplers, [78-79](#)

texture coordinates, [79-81](#)

*addressing modes*, [86-88](#)

`TextureMapping.fx`, SamplerState filtering options, [85](#)

`TextureModelDemo` class, [335-339](#)

Textures panel (FX Composer), [52](#)

TGA (Targa), [46](#)

threads, [545](#)

time-related information

    game loop, [196-199](#)

*GameClock.cpp file*, [198-199](#)

transformations

    hierarchical transformations, [470](#)

    homogeneous coordinates, [31](#)

    quaternions, [486](#)

    rotating, [33](#)

    scaling, [31-32](#)

    translating, [32-33](#)

translating transformations, [32-33](#)

transparency mapping effect, [165-166](#)

`TransparencyMapping.fx`, [161-165](#)

transposing matrices, [29](#)

triangle, rendering for “Hello, Rendering” application, [297-304](#)

triangle list, [14](#)

`TriangleDemo` class, [286-287](#)

## U

UAVs (unordered access views), [547](#)

UDK (Unreal Development Kit), [550](#)

Unity game engine, [550](#)

`UpdateGameTime()` method, [196](#)

`UpdateProjectionMatrix()` method, [276](#)

`UpdateViewMatrix()` method, [276](#)

## V

Variable class, [350-352](#)

vectors, [24-27](#)

- adding and subtracting, [25](#)

- coordinate systems, [24-25](#)

- cross product, [27](#)

- DirectXMath, [36](#)

- dot product, [26-27](#)

- length of, [26](#)

- tangent space, [171-173](#)

vertex buffers, [10-11](#)

vertex shader stage (Direct3D Graphics Pipeline), [16](#)

vertex shaders, [64-65](#)

- diffuse lighting vertex shader, [102](#)

- displacement mapping effect, [180](#)

- environment mapping effect, [152](#)

- fog effect, [157](#)

- multiple point lights effect, [136-137](#)

- normal mapping effect, [177](#)

- Phong effect, [109](#)

- point light vertex shader, [120](#)

- skybox effect, [147](#)

- spotlight vertex shader, [129](#)

vertices

- displacing, [178](#)

- displacing tessellated vertices, [520-523](#)

- meshes, [320-321](#)

- point sprite shaders, [499-506](#)

view frustum, [270, 530](#)

view space, [34](#)

viewports, [219](#)

views

- associating to output-merger stage, [218](#)

- render target views, [396](#)

- UAVs, [547](#)

Visual Studio, [44-46](#)

Visual Studio Graphics Debugger, [53-55](#)

## W

websites, companion website for this book, [4](#)

WIC (Windows Imaging Components), [46](#)

windows, initializing, [199-201](#)

Windows 7, [8](#)

Windows SDK, [44-45](#)

Windows Vista, [8](#)

WinMain() function, [194](#)

    updating for RenderingGame class, [230-232](#)

world space, [34](#)

wrap address mode, [86](#)

## X-Y-Z

XBox, [8](#)

XBox 360, [8](#)

XML, [550-551](#)

z-culling, [532](#)

# REGISTER



## THIS PRODUCT

[informit.com/register](http://informit.com/register)

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

**informIT.com**

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

PEARSON

**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

▲Addison-Wesley

Cisco Press

EXAM/CRAM

IBM

Press

QUE'



PRENTICE

HALL

SAMS

Safari<sup>®</sup>

## LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters.  
Visit [informit.com/newsletters](http://informit.com/newsletters).
- Access FREE podcasts from experts at [informit.com/podcasts](http://informit.com/podcasts).
- Read the latest author articles and sample chapters at [informit.com/articles](http://informit.com/articles).
- Access thousands of books and videos in the Safari Books Online digital library at [safari.informit.com](http://safari.informit.com).
- Get tips from expert blogs at [informit.com/blogs](http://informit.com/blogs).

Visit [informit.com/learn](http://informit.com/learn) to discover all the ways you can access the hottest technology content.

### Are You Part of the **IT** Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit [informit.com/socialconnect](http://informit.com/socialconnect).



# informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

▲Addison-Wesley

Cisco Press

EXAM/CRAM

IBM

Press

QUE'



PRENTICE

HALL

SAMS

Safari<sup>®</sup>

```
float4 vertex_shader(float3 objectPosition : POSITION) : SV_Position
{
    return mul(float4(objectPosition, 1), WorldViewProjection);
}
```

---

```
float4 pixel_shader() : SV_Target
{
    return float4(1, 0, 0, 1);
}
```

---

```
XMVECTOR XMLoadFloat2(const XMFLOAT2* pSource);  
XMVECTOR XMLoadFloat3(const XMFLOAT3* pSource);  
XMVECTOR XMLoadFloat4(const XMFLOAT4* pSource);
```

```
void XMStoreFloat2(XMFLOAT2* pDestination, FXMVECTOR V);  
void XMStoreFloat3(XMFLOAT3* pDestination, FXMVECTOR V);  
void XMStoreFloat4(XMFLOAT4* pDestination, FXMVECTOR V);
```

```
float XMVectorGetX(XMVECTOR V);
float XMVectorGetY(XMVECTOR V);
float XMVectorGetZ(XMVECTOR V);
float XMVectorGetW(XMVECTOR V);

void XMVectorSetX(XMVECTOR V, float X);
void XMVectorSetY(XMVECTOR V, float Y);
void XMVectorSetZ(XMVECTOR V, float Z);
void XMVectorSetW(XMVECTOR V, float W);
```

```
XMVECTOR operator+ (FXMVECTOR V);
XMVECTOR operator- (FXMVECTOR V);

XMVECTOR& operator+= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator-= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator*= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator/= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR& operator*= (XMVECTOR& V, float S);
XMVECTOR& operator/= (XMVECTOR& V, float S);

XMVECTOR operator+ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator- (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator* (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator/ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR operator* (FXMVECTOR V, float S);
XMVECTOR operator* (float S, FXMVECTOR V);
XMVECTOR operator/ (FXMVECTOR V, float S);
```

```
struct XMFLOAT4X4
{
    union
    {
        struct
        {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
}
```

```
XMVECTOR XMLoadFloat4x4 (const XMFLOAT4X4* pSource);  
  
void XMStoreFloat4x4 (XMFLOAT4X4* pDestination, FXMMATRIX M);
```

```
XMMATRIX XMMatrixIdentity();
```

```
XMMATRIX& operator+= (FXMMATRIX M);
XMMATRIX& operator-= (FXMMATRIX M);
XMMATRIX& operator*= (FXMMATRIX M);
XMMATRIX& operator*= (float S);
XMMATRIX& operator/= (float S);

XMMATRIX operator+ (FXMMATRIX M) const;
XMMATRIX operator- (FXMMATRIX M) const;
XMMATRIX operator* (FXMMATRIX M) const;
XMMATRIX operator* (float S) const;
XMMATRIX operator/ (float S) const;
```

```
cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

RasterizerState DisableCulling
{
    CullMode = NONE;
};

float4 vertex_shader(float3 objectPosition : POSITION) : SV_Position
{
    return mul(float4(objectPosition, 1), WorldViewProjection);
}

float4 pixel_shader() : SV_Target
{
    return float4(1, 0, 0, 1);
}

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
RasterizerState DisableCulling
{
    CullMode = NONE;
};
```

---

```
float4 vertex_shader(float3 objectPosition : POSITION) : SV_Position
{
    return mul(float4(objectPosition, 1), WorldViewProjection);
}
```

---

```
float4 pixel_shader() : SV_Target
{
    return float4(1, 0, 0, 1);
}
```

---

```
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

RasterizerState DisableCulling
{
    CullMode = NONE;
};

struct VS_INPUT
{
    float4 ObjectPosition: POSITION;
};

struct VS_OUTPUT
{
    float4 Position: SV_Position;
};

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);

    return OUT;
}

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return float4(1, 0, 0, 1);
}

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

```
***** Resources *****
```

```
#define FLIP_TEXTURE_Y 1

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
    UIWidget="None"; >;
}

RasterizerState DisableCulling
{
    CullMode = NONE;
};

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
```

```
Filter = MIN_MAG_MIP_LINEAR;
AddressU = WRAP;
AddressV = WRAP;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/***** Utility Functions *****/
float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else

```

```
        return textureCoordinate;
#endif
}

/***************************************** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    return OUT;
}

/***************************************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(ColorSampler, IN.TextureCoordinate);
}

/***************************************** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));
        SetRasterizerState(DisableCulling);
    }
}
```

---

```
Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>;
```

---

```
SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

---

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(ColorSampler, IN.TextureCoordinate);
}
```

---

```
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};
```

---

```
float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    return OUT;
}
```

---

```
SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

---

```
***** Resources *****

#define FLIP_TEXTURE_Y 1

cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None";
    >;
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

***** Data Structures *****

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};
```

```
struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/**************** Utility Functions *****/
float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

/**************** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
```

```
    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    return OUT;
}

/***** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    OUT = ColorTexture.Sample(ColorSampler, IN.TextureCoordinate);
    OUT.rgb *= AmbientColor.rgb * AmbientColor.a; // Color (.rgb) *
Intensity (.a)

    return OUT;
}

/***** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
#include "include\\Common.fxh"

/***** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightDirection : DIRECTION <
        string Object = "DirectionalLight0";
        string UIName = "Light Direction";
        string Space = "World";
    > = {0.0f, 0.0f, -1.0f};
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>;

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
```

```
CullMode = NONE;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 LightDirection : TEXCOORD1;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
```

```
    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
    OUT.LightDirection = normalize(-LightDirection);

    return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float n_dot_l = dot(lightDirection, normal);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a * color.rgb;
```

```
float3 diffuse = (float3)0;

if (n_dot_l > 0)
{
    diffuse = LightColor.rgb * LightColor.a * n_dot_l * color.rgb;
}

OUT.rgb = ambient + diffuse;
OUT.a = color.a;

return OUT;
}

/***** Techniques *****/
technique10 main10
{
    pass p0
    {

        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
#ifndef _COMMON_FXH
#define _COMMON_FXH

/***** Constants *****/
#define FLIP_TEXTURE_Y 1

/***** Utility Functions *****/
float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

#endif /* _COMMON_FXH */
```

---

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float n_dot_l = dot(lightDirection, normal);
    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a * color.rgb;

    float3 diffuse = (float3)0;

    if (n_dot_l > 0)
    {
        diffuse = LightColor.rgb * LightColor.a * n_dot_l * color.rgb;
    }

    OUT.rgb = ambient + diffuse;
    OUT.a = color.a;

    return OUT;
}
```

---

```
#include "include\\Common.fhx"

/********************* Resources ********************/

cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightDirection : DIRECTION <
        string Object = "DirectionalLight0";
        string UIName = "Light Direction";
        string Space = "World";
    > = {0.0f, 0.0f, -1.0f};

    float3 CameraPosition : CAMERAPosition < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
    > = {25.0f};
```

```
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
```

```
    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
    OUT.LightDirection = normalize(-LightDirection);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.ViewDirection = normalize(CameraPosition - worldPosition);

    return OUT;
}

/****************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(lightDirection, normal);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a * color.rgb;

    float3 diffuse = (float3)0;
    float3 specular = (float3)0;

    if (n_dot_l > 0)
    {
        diffuse = LightColor.rgb * LightColor.a * saturate(n_dot_l) *
color.rgb;

        // R = 2 * (N.L) * N - L
        float3 reflectionVector = normalize(2 * n_dot_l * normal
- lightDirection);

        // specular = R.V^n with gloss map in color texture's alpha
channel
```

```
        specular = SpecularColor.rgb * SpecularColor.a * min(pow(saturate
(dot(reflectionVector, viewDirection)), SpecularPower), color.w);
    }

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}

technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
// specular = R.V^n with gloss map stored in color texture's alpha
// channel
specular = SpecularColor.rgb * SpecularColor.a * min(pow(saturate(dot
(reflectionVector, viewDirection)), SpecularPower), color.w);
```

---

```
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(lightDirection, normal);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a * color.rgb;

    float3 diffuse = (float3)0;
    float3 specular = (float3)0;

    if (n_dot_l > 0)
    {
        diffuse = LightColor.rgb * LightColor.a * saturate(n_dot_l) *
color.rgb;

        float3 halfVector = normalize(lightDirection + viewDirection);

        //specular = N.H^s w/ gloss map stored in color texture's alpha
channel
        specular = SpecularColor.rgb * SpecularColor.a *
min(pow(saturate(dot(normal, halfVector)), SpecularPower), color.w);
    }

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}
```

---

```
float3 get_vector_color_contribution(float4 light, float3 color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float3 get_scalar_color_contribution(float4 light, float color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb);
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w));

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}
```

```
#include "include\\Common.fhx"

/***** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightPosition : POSITION <
        string Object = "PointLight0";
        string UIName = "Light Position";
        string Space = "World";
    > = {0.0f, 0.0f, 0.0f};

    float LightRadius <
        string UIName = "Light Radius";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 100.0;
        float UIStep = 1.0;
    > = {10.0f};

    float3 CameraPosition : CAMERAPosition < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
    > = {1.0f};
}
```

```
    float UIMin = 1.0;
    float UIMax = 255.0;
    float UIStep = 1.0;
    > = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***************** Data Structures ********************/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
```

```
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float4 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
};

/************************************************************************************************ Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
    float3 lightDirection = LightPosition - worldPosition;
    OUT.LightDirection.xyz = normalize(lightDirection);
    OUT.LightDirection.w = saturate(1.0f - (length(lightDirection) /
LightRadius)); // Attenuation
    OUT.ViewDirection = normalize(CameraPosition - worldPosition);
```

```
    return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection.xyz);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.LightDirection.w;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.LightDirection.w;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

/***************** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

```
struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));
    return OUT;
}

/***** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = normalize(LightPosition
- IN.WorldPosition);
    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);
```

```
float3 normal = normalize(IN.Normal);
float n_dot_l = dot(normal, lightDirection);
float3 halfVector = normalize(lightDirection + viewDirection);
float n_dot_h = dot(normal, halfVector);

float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

float3 ambient = get_vector_color_contribution(AmbientColor, color.rgb);
float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

return OUT;
}
```

---

```
#include "include\\Common.fxh"

/**************** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Spotlight Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightPosition : POSITION <
        string Object = "SpotLightPosition0";
        string UIName = "Spotlight Position";
        string Space = "World";
    > = {0.0f, 0.0f, 0.0f};

    float3 LightLookAt : DIRECTION <
        string Object = "SpotLightDirection0";
        string UIName = "Spotlight Direction";
        string Space = "World";
    > = {0.0f, 0.0f, -1.0f};

    float LightRadius <
        string UIName = "Spotlight Radius";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 100.0;
        float UIStep = 1.0;
    > = {10.0f};

    float SpotLightInnerAngle <
        string UIName = "Spotlight Inner Angle";
        string UIWidget = "slider";
        float UIMin = 0.5;
        float UIMax = 1.0;
        float UIStep = 0.01;
    > = {0.75f};
```

```
float SpotLightOuterAngle <
    string UIName = "Spotlight Outer Angle";
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 0.5;
    float UIStep = 0.01;
> = {0.25f};

    float3 CameraPosition : CAMERAPOSITION < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
> = {25.0f};
}
```

```
Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***************** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
```

```
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
    float3 LightLookAt : TEXCOORD3;
};

/***************** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal,0), World).xyz);
```

```
    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - length(lightDirection) /
LightRadius);

    OUT.LightLookAt = -LightLookAt;

    return OUT;
}

/***** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float)0;

    float3 lightDirection = normalize(LightPosition
- IN.WorldPosition);
    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);
    float3 lightLookAt = normalize(IN.LightLookAt);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    float spotFactor = 0.0f;
    float lightAngle = dot(lightLookAt, lightDirection);
    if (lightAngle > 0.0f)
    {
        spotFactor = smoothstep(SpotLightOuterAngle,
SpotLightInnerAngle, lightAngle);
    }
}
```

```
    OUT.rgb = ambient + (spotFactor * (diffuse + specular));
    OUT.a = 1.0f;

    return OUT;
}

/***** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
#ifndef _COMMON_FXH
#define _COMMON_FXH

/***** Constants *****/
#define FLIP_TEXTURE_Y 1

/***** Data Structures *****/
struct POINT_LIGHT
{
    float3 Position;
    float LightRadius;
    float4 Color;
};

struct LIGHT_CONTRIBUTION_DATA
{
    float4 Color;
    float3 Normal;
    float3 ViewDirection;
    float4 LightColor;
    float4 LightDirection;
    float4 SpecularColor;
    float SpecularPower;
};

/***** Utility Functions *****/
float2 get_corrected_texture_coordinate(float2 textureCoordinate)
{
    #if FLIP_TEXTURE_Y
        return float2(textureCoordinate.x, 1.0 - textureCoordinate.y);
    #else
        return textureCoordinate;
    #endif
}

float3 get_vector_color_contribution(float4 light, float3 color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}
```

```
float3 get_scalar_color_contribution(float4 light, float color)
{
    // Color (.rgb) * Intensity (.a)
    return light.rgb * light.a * color;
}

float4 get_light_data(float3 lightPosition, float3 worldPosition, float
lightRadius)
{
    float4 lightData;
    float3 lightDirection = lightPosition - worldPosition;

    lightData.xyz = normalize(lightDirection);
    lightData.w = saturate(1.0f - length(lightDirection) /
lightRadius); // Attenuation

    return lightData;
}

float3 get_light_contribution(LIGHT_CONTRIBUTION_DATA IN)
{
    float3 lightDirection = IN.LightDirection.xyz;
    float n_dot_l = dot(IN.Normal, lightDirection);
    float3 halfVector = normalize(lightDirection + IN.ViewDirection);
    float n_dot_h = dot(IN.Normal, halfVector);

    float4 lightCoefficients = lit(n_dot_l, n_dot_h, IN.SpecularPower);
    float3 diffuse = get_vector_color_contribution(IN.LightColor,
lightCoefficients.y * IN.Color.rgb) * IN.LightDirection.w;
    float3 specular = get_scalar_color_contribution(IN.SpecularColor,
min(lightCoefficients.z, IN.Color.w)) * IN.LightDirection.w *
IN.LightColor.w;

    return (diffuse + specular);
}

#endif /* _COMMON_FXH */
```

---

```
#include "include\\Common.fxh"

/********************* Resources ********************/

#define NUM_LIGHTS 4

cbuffer CBufferPerFrame
{
    POINT_LIGHT PointLights[NUM_LIGHTS];

    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float3 CameraPosition : CAMERAPosition < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >;
```

```
float4 SpecularColor : SPECULAR <
    string UIName = "Specular Color";
    string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};

float SpecularPower : SPECULARPOWER <
    string UIName = "Specular Power";
    string UIWidget = "slider";
    float UIMin = 1.0;
    float UIMax = 255.0;
    float UIStep = 1.0;
> = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

```
RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 WorldPosition : POSITION;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
```

```
    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0) World).xyz);

    return OUT;
}

/****************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN, uniform int lightCount) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);
    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);

    LIGHT_CONTRIBUTION_DATA lightContributionData;
    lightContributionData.Color = color;
```

```
lightContributionData.Normal = normal;
lightContributionData.ViewDirection = viewDirection;
lightContributionData.SpecularColor = SpecularColor;
lightContributionData.SpecularPower = SpecularPower;

float3 totalLightContribution = (float3)0;

[unroll]
for (int i = 0; i < lightCount; i++)
{
    lightContributionData.LightDirection = get_light_
data(PointLights[i].Position, IN.WorldPosition, PointLights[i].
LightRadius);
    lightContributionData.LightColor = PointLights[i].Color;
    totalLightContribution += get_light_contribution
(lightContributionData);
}

OUT.rgb = ambient + totalLightContribution;
OUT.a = 1.0f;

return OUT;
}

***** Techniques *****
```

```
technique10 Lights1
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(1)));

        SetRasterizerState(DisableCulling);
    }
}

technique10 Lights2
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(2)));

        SetRasterizerState(DisableCulling);
    }
}

technique10 Lights3
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(3)));
        SetRasterizerState(DisableCulling);
    }
}

technique10 Lights4
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(4)));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
***** Resources *****
```

```
cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
}
```

```
TextureCube SkyboxTexture <
    string UIName = "Skybox Texture";
    string ResourceType = "3D";
>;
```

```
SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
};
```

```
RasterizerState DisableCulling
{
    CullMode = NONE;
};
```

```
***** Data Structures *****
```

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

```
struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 TextureCoordinate : TEXCOORD;
};
```

```
***** Vertex Shader *****
```

```
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
```

```
OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
OUT.TextureCoordinate = IN.ObjectPosition;

return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return SkyboxTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
}

/***************** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
#include "include\\Common.fxh"

/**************** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 EnvColor : COLOR <
        string UIName = "Environment Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 CameraPosition : CAMERAPOSITION < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >

    float ReflectionAmount <
        string UIName = "Reflection Amount";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 1.0;
        float UIStep = 0.05;
    > = {0.5f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>;
```

```
TextureCube EnvironmentMap <
    string UIName = "Environment Map";
    string ResourceType = "3D";
>

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD0;
    float3 ReflectionVector : TEXCOORD1;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
```

```
float3 incident = normalize(worldPosition - CameraPosition);
float3 normal = normalize(mul(float4(IN.Normal, 0), World).xyz);

// Reflection Vector for cube map: R = I - 2 * N * (I.N)
OUT.ReflectionVector = reflect(incident, normal);

return OUT;
}

/***** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 environment = EnvironmentMap.Sample(TrilinearSampler,
IN.ReflectionVector).rgb;
    float3 reflection = get_vector_color_contribution(EnvColor,
environment);

    OUT.rgb = lerp(ambient, reflection, ReflectionAmount);

    return OUT;
}

/***** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
cbuffer CBufferPerFrame
{
    /* ... */

    float3 FogColor <
        string UIName = "Fog Color";
        string UIWidget = "Color";
    > = {0.5f, 0.5f, 0.5f};

    float FogStart = { 20.0f };
    float FogRange = { 40.0f };

    float3 CameraPosition : CAMERAPosition < string UIWidget="None"; >;
}

/***************** Data Structures ****************/

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
    float FogAmount: TEXCOORD3;
};

/***************** Utility Functions ****************/

float get_fog_amount(float3 viewDirection, float fogStart, float fogRange)
{
    return saturate((length(viewDirection) - fogStart) / (fogRange));
}

/***************** Vertex Shader ****************/

VS_OUTPUT vertex_shader(VS_INPUT IN, uniform bool fogEnabled)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
```

```
OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
OUT.LightDirection = normalize(-LightDirection);

float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
float3 viewDirection = CameraPosition - worldPosition;
OUT.ViewDirection = normalize(viewDirection);

if (fogEnabled)
{
    OUT.FogAmount = get_fog_amount(viewDirection, FogStart,
FogRange);
}

return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN, uniform bool fogEnabled) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 viewDirection = normalize(IN.ViewDirection);
    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);

    LIGHT_CONTRIBUTION_DATA lightContributionData;
    lightContributionData.Color = color;
    lightContributionData.Normal = normal;
    lightContributionData.ViewDirection = viewDirection;
    lightContributionData.LightDirection = float4(IN.LightDirection, 1);
    lightContributionData.SpecularColor = SpecularColor;
    lightContributionData.SpecularPower = SpecularPower;
    lightContributionData.LightColor = LightColor;
    float3 light_contribution = get_light_contribution(lightContributi
onData);

    OUT.rgb = ambient + light_contribution;
    OUT.a = 1.0f;
```

```
if (fogEnabled)
{
    OUT.rgb = lerp(OUT.rgb, FogColor, IN.FogAmount);
}

return OUT;
}

/********************* Techniques *****/
technique10 fogEnabled
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader(true)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(true)));

        SetRasterizerState(DisableCulling);
    }
}

technique10 fogDisabled
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader(false)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(false)));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
#include "include\\Common.fhx"

/**************** Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 0.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightPosition : POSITION <
        string Object = "PointLight0";
        string UIName = "Light Position";
        string Space = "World";
    > = {0.0f, 0.0f, 0.0f};

    float LightRadius <
        string UIName = "Light Radius";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 100.0;
        float UIStep = 1.0;
    > = {10.0f};

    float3 FogColor <
        string UIName = "Fog Color";
        string UIWidget = "Color";
    > = {0.5f, 0.5f, 0.5f};

    float FogStart = { 20.0f };
    float FogRange = { 40.0f };

    float3 CameraPosition : CAMERAPosition < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
```

```
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
    > = {25.0f};
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>;

Texture2D TransparencyMap <
    string UIName = "Transparency Map";
    string ResourceType = "2D";
>;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

BlendState EnableAlphaBlending
{
```

```
BlendEnable[0] = True;
SrcBlend = SRC_ALPHA;
DestBlend = INV_SRC_ALPHA;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float4 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float4 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
    float FogAmount : TEXCOORD3;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN, uniform bool fogEnabled)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;

    OUT.LightDirection = get_light_data(LightPosition, worldPosition,
LightRadius);
    float3 viewDirection = CameraPosition - worldPosition;

    if (fogEnabled)
    {
```

```
        OUT.FogAmount = get_fog_amount(viewDirection, FogStart,
FogRange);
    }

    OUT.ViewDirection = normalize(viewDirection);

    return OUT;
}

/***************** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN, uniform bool fogEnabled) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 viewDirection = normalize(IN.ViewDirection);
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);

    LIGHT_CONTRIBUTION_DATA lightContributionData;
    lightContributionData.Color = color;
    lightContributionData.Normal = normal;
    lightContributionData.ViewDirection = viewDirection;
    lightContributionData.SpecularColor = SpecularColor;
    lightContributionData.SpecularPower = SpecularPower;
    lightContributionData.LightDirection = IN.LightDirection;
    lightContributionData.LightColor = LightColor;
    float3 light_contribution = get_light_contribution(lightContributi
onData);

    OUT.rgb = ambient + light_contribution;
    OUT.a = TransparencyMap.Sample(TrilinearSampler,
IN.TextureCoordinate).a;

    if (fogEnabled)
    {
        OUT.rgb = lerp(OUT.rgb, FogColor, IN.FogAmount);
    }
}
```

```
    return OUT;
}

technique10 alphaBlendingWithFog
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader(true)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(true)));

        SetRasterizerState(DisableCulling);
        SetBlendState(EnableAlphaBlending, (float4)0, 0xFFFFFFFF);
    }
}

technique10 alphaBlendingWithoutFog
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader(false)));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader(false)));

        SetRasterizerState(DisableCulling);
        SetBlendState(EnableAlphaBlending, (float4)0, 0xFFFFFFFF);
    }
}
```

---

```
#include "include\\Common.fhx"

cbuffer CBufferPerFrame
{
    float4 AmbientColor : AMBIENT <
        string UIName = "Ambient Light";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float4 LightColor : COLOR <
        string Object = "LightColor0";
        string UIName = "Light Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float3 LightDirection : DIRECTION <
        string Object = "DirectionalLight0";
        string UIName = "Light Direction";
        string Space = "World";
    > = {0.0f, 0.0f, -1.0f};

    float3 CameraPosition : CAMERAPosition < string UIWidget="None"; >;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION < string
UIWidget="None"; >;
    float4x4 World : WORLD < string UIWidget="None"; >

    float4 SpecularColor : SPECULAR <
        string UIName = "Specular Color";
        string UIWidget = "Color";
    > = {1.0f, 1.0f, 1.0f, 1.0f};

    float SpecularPower : SPECULARPOWER <
        string UIName = "Specular Power";
        string UIWidget = "slider";
        float UIMin = 1.0;
        float UIMax = 255.0;
        float UIStep = 1.0;
    > = {25.0f};
```

```
}

Texture2D ColorTexture <
    string ResourceName = "default_color.dds";
    string UIName = "Color Texture";
    string ResourceType = "2D";
>

Texture2D NormalMap <
    string ResourceName = "default_bump_normal.dds";
    string UIName = "Normap Map";
    string ResourceType = "2D";
>

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

/***************** Data Structures *******/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
    float3 Tangent : TANGENT;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float3 Tangent : TANGENT;
    float3 Binormal : BINORMAL;
    float2 TextureCoordinate : TEXCOORD0;
```

```
    float3 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
};

//********************************************************************* Vertex Shader ******/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
    OUT.Tangent = normalize(mul(float4(IN.Tangent, 0), World).xyz);
    OUT.Binormal = cross(OUT.Normal, OUT.Tangent);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);
    OUT.LightDirection = normalize(-LightDirection);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
    float3 viewDirection = CameraPosition - worldPosition;
    OUT.ViewDirection = normalize(viewDirection);

    return OUT;
}

//********************************************************************* Pixel Shader ******/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 sampledNormal = (2 * NormalMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xyz) - 1.0; // Map normal from [0..1] to [-1..1]
    float3x3 tbn = float3x3(IN.Tangent, IN.Binormal, IN.Normal);

    sampledNormal = mul(sampledNormal, tbn); // Transform normal to
world space

    float3 viewDirection = normalize(IN.ViewDirection);
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
```

```
LIGHT_CONTRIBUTION_DATA lightContributionData;
lightContributionData.Color = color;
lightContributionData.Normal = sampledNormal;
lightContributionData.ViewDirection = viewDirection;
lightContributionData.LightDirection = float4(IN.LightDirection, 1);
lightContributionData.SpecularColor = SpecularColor;
lightContributionData.SpecularPower = SpecularPower;
lightContributionData.LightColor = LightColor;
float3 light_contribution = get_light_contribution
(lightContributionData);

OUT.rgb = ambient + light_contribution;
OUT.a = 1.0f;

return OUT;
}

/***** Techniques *****/
technique10 main10
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_4_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

---

```
cbuffer CBufferPerObject
{
    /* ... */

    float DisplacementScale <
        string UIName = "Displacement Scale";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 2.0;
        float UIStep = 0.01;
    > = {0.0f};
}

Texture2D DisplacementMap <
    string UIName = "Displacement Map";
    string ResourceType = "2D";
>

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float2 textureCoordinate = get_corrected_texture_coordinate(IN.
TextureCoordinate);

    if (DisplacementScale > 0.0f)
    {
        float displacement = DisplacementMap.
SampleLevel(TrilinearSampler, textureCoordinate, 0);
        IN.ObjectPosition.xyz += IN.Normal * DisplacementScale *
(displacement - 1);
    }

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = textureCoordinate;
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);

    float3 worldPosition = normalize(mul(IN.ObjectPosition, World)).
xyz;
    OUT.ViewDirection = normalize(CameraPosition - worldPosition);

    OUT.LightDirection = get_light_data(LightPosition, worldPosition,
LightRadius);

    return OUT;
}
```

`$(SolutionDir)..\source\Library`

```
#include <memory>
#include "Game.h"
#include "GameException.h"

#if defined(DEBUG) || defined(_DEBUG)
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#endif

using namespace Library;

int WINAPI WinMain(HINSTANCE instance, HINSTANCE previousInstance,
LPSTR commandLine, int showCommand)
{
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    std::unique_ptr<Game> game(new Game(instance, L"RenderingClass",
L"Real-Time 3D Rendering", showCommand));

    try
    {
        game->Run();
    }
    catch (GameException ex)
    {
        MessageBox(game->WindowHandle(), ex.whatw().c_str(), game-
>WindowTitle().c_str(), MB_ABORTRETRYIGNORE);
    }

    return 0;
}
```

---

```
void Game::Run()
{
    Initialize();

    while(isRunning)
    {
        mGameClock.UpdateGameTime(mGameTime);

        Update(mGameTime);
        Draw(mGameTime);
    }

    Shutdown();
}

void Game::Exit()
{
    isRunning = false;
}
```

---

```
#pragma once

#include <windows.h>
#include <exception>

namespace Library
{
    class GameTime;
    class GameClock
    {
    public:
        GameClock();

        const LARGE_INTEGER& StartTime() const;
        const LARGE_INTEGER& CurrentTime() const;
        const LARGE_INTEGER& LastTime() const;

        void Reset();
        double GetFrequency() const;
        void GetTime(LARGE_INTEGER& time) const;
        void UpdateGameTime(GameTime& gameTime);

    private:
        GameClock(const GameClock& rhs);
        GameClock& operator=(const GameClock& rhs);

        LARGE_INTEGER mStartTime;
        LARGE_INTEGER mCurrentTime;
        LARGE_INTEGER mLastTime;
        double mFrequency;
    };
}
```

---

```
#include "GameClock.h"
#include "GameTime.h"

namespace Library
{
    GameClock::GameClock()
        : mStartTime(), mCurrentTime(), mLastTime(), mFrequency()
    {
        mFrequency = GetFrequency();
        Reset();
    }

    const LARGE_INTEGER& GameClock::StartTime() const
    {
        return mStartTime;
    }

    const LARGE_INTEGER& GameClock::CurrentTime() const
    {
        return mCurrentTime;
    }

    const LARGE_INTEGER& GameClock::LastTime() const
    {
        return mLastTime;
    }

    void GameClock::Reset()
    {
        GetTime(mStartTime);
        mCurrentTime = mStartTime;
        mLastTime = mCurrentTime;
    }

    double GameClock::GetFrequency() const
    {
        LARGE_INTEGER frequency;

        if (QueryPerformanceFrequency(&frequency) == false)
        {
```

```
        throw std::exception("QueryPerformanceFrequency()\n"
failed.");  
    }  
  
    return (double)frequency.QuadPart;  
}  
  
void GameClock::GetTime(LARGE_INTEGER& time) const  
{  
    QueryPerformanceCounter(&time);  
}  
  
void GameClock::UpdateGameTime(GameTime& gameTime)  
{  
    GetTime(mCurrentTime);  
    gameTime.SetTotalGameTime((mCurrentTime.QuadPart - mStartTime.  
QuadPart) / mFrequency);  
    gameTime.SetElapsedGameTime((mCurrentTime.QuadPart - mLasteTime.  
QuadPart) / mFrequency);  
    mLasteTime = mCurrentTime;  
}  
}
```

---

```
#pragma once

#include <windows.h>
#include <string>
#include "GameClock.h"
#include "GameTime.h"

namespace Library
{
    class Game
    {
        public:
            Game(HINSTANCE instance, const std::wstring& windowClass, const std::wstring& windowTitle, int showCommand);
            virtual ~Game();

            HINSTANCE Instance() const;
            HWND WindowHandle() const;
            const WNDCLASSEX& Window() const;
            const std::wstring& WindowClass() const;
            const std::wstring& WindowTitle() const;
            int ScreenWidth() const;
            int ScreenHeight() const;

            virtual void Run();
            virtual void Exit();
            virtual void Initialize();
            virtual void Update(const GameTime& gameTime);
            virtual void Draw(const GameTime& gameTime);

        protected:
            virtual void InitializeWindow();
            virtual void Shutdown();

            static const UINT DefaultScreenWidth;
            static const UINT DefaultScreenHeight;

            HINSTANCE mInstance;
            std::wstring mWindowClass;
            std::wstring mWindowTitle;
            int mShowCommand;
    };
}
```

```
    HWND mWindowHandle;
    WNDCLASSEX mWindow;

    UINT mScreenWidth;
    UINT mScreenHeight;

    GameClock mGameClock;
    GameTime mGameTime;

private:
    Game(const Game& rhs);
    Game& operator=(const Game& rhs);

    POINT CenterWindow(int windowHeight, int windowWidth);
    static LRESULT WINAPI WndProc(HWND windowHandle, UINT message,
WPARAM, LPARAM lParam);
};

}
```

---

```
void Game::InitializeWindow()
{
    ZeroMemory(&mWindow, sizeof(mWindow));
    mWindow.cbSize = sizeof(WNDCLASSEX);
    mWindow.style = CS_CLASSDC;
    mWindow.lpfnWndProc = WndProc;
    mWindow.hInstance = mInstance;
    mWindow.hIcon = LoadIcon(nullptr, IDI_APPLICATION);
    mWindow.hIconSm = LoadIcon(nullptr, IDI_APPLICATION);
    mWindow.hCursor = LoadCursor(nullptr, IDC_ARROW);
    mWindow.hbrBackground = GetSysColorBrush(COLOR_BTNFACE);
    mWindow.lpszClassName = mWindowClass.c_str();

    RECT windowRectangle = { 0, 0, mScreenWidth, mScreenHeight };
    AdjustWindowRect(&windowRectangle, WS_OVERLAPPEDWINDOW, FALSE);

    RegisterClassEx(&mWindow);
    POINT center = CenterWindow(mScreenWidth, mScreenHeight);
    mWindowHandle = CreateWindow(mWindowClass.c_str(), mWindowTitle.c_str(),
        WS_OVERLAPPEDWINDOW, center.x, center.y, windowRectangle.right -
        windowRectangle.left, windowRectangle.bottom - windowRectangle.top,
        nullptr, nullptr, mInstance, nullptr);

    ShowWindow(mWindowHandle, mShowCommand);
    UpdateWindow(mWindowHandle);
}
```

---

```
LRESULT WINAPI Game::WndProc(HWND windowHandle, UINT message, WPARAM,
LPARAM lParam)
{
    switch(message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }

    return DefWindowProc(windowHandle, message, wParam, lParam);
}

POINT Game::CenterWindow(int windowHeight, int windowWidth)
{
    int screenWidth = GetSystemMetrics(SM_CXSCREEN);
    int screenHeight = GetSystemMetrics(SM_CYSCREEN);

    POINT center;
    center.x = (screenWidth - windowWidth) / 2;
    center.y = (screenHeight - windowHeight) / 2;

    return center;
}
```

---

```
void Game::Run()
{
    InitializeWindow();

    MSG message;
    ZeroMemory(&message, sizeof(message));

    mGameClock.Reset();

    while(message.message != WM_QUIT)
    {
        if (PeekMessage(&message, nullptr, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&message);
            DispatchMessage(&message);
        }
        else
        {
            mGameClock.UpdateGameTime(mGameTime);
            Update(mGameTime);
            Draw(mGameTime);
        }
    }

    Shut down();
}
```

---

```
HRESULT WINAPI D3D11CreateDevice(
    IDXGIAdapter* pAdapter,
    D3D_DRIVER_TYPE DriverType,
    HMODULE Software,
    UINT Flags,
    CONST D3D_FEATURE_LEVEL* pFeatureLevels,
    UINT FeatureLevels,
    UINT SDKVersion,
    ID3D11Device** ppDevice,
    D3D_FEATURE_LEVEL* pFeatureLevel,
    ID3D11DeviceContext** ppImmediateContext );
```

```
HRESULT hr;
UINT createDeviceFlags = 0;

#if defined(DEBUG) || defined(_DEBUG)
    createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

D3D_FEATURE_LEVEL featureLevels[] = {
    D3D_FEATURE_LEVEL_11_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_10_0
};

ID3D11Device* direct3DDevice;
D3D_FEATURE_LEVEL selectedFeatureLevel;
ID3D11DeviceContext* direct3DDeviceContext;
if (FAILED(hr = D3D11CreateDevice(NULL, D3D_DRIVER_TYPE_HARDWARE,
NULL, createDeviceFlags, featureLevels, ARRAYSIZE(featureLevels),
D3D11_SDK_VERSION, &direct3DDevice, &selectedFeatureLevel,
&direct3DDeviceContext)))
{
    throw GameException("D3D11CreateDevice() failed", hr);
}
```

---

```
HRESULT CheckMultisampleQualityLevels(  
    DXGI_FORMAT Format,  
    UINT SampleCount,  
    UINT *pNumQualityLevels);
```

```
typedef struct DXGI_SWAP_CHAIN_DESC1
{
    UINT Width;
    UINT Height;
    DXGI_FORMAT Format;
    BOOL Stereo;
    DXGI_SAMPLE_DESC SampleDesc;
    DXGI_USAGE BufferUsage;
    UINT BufferCount;
    DXGI_SCALING Scaling;
    DXGI_SWAP_EFFECT SwapEffect;
    DXGI_ALPHA_MODE AlphaMode;
    UINT Flags;
} DXGI_SWAP_CHAIN_DESC1;
```

```
mDirect3DDevice->CheckMultisampleQualityLevels(D  
XGI_FORMAT_R8G8B8A8_UNORM,  
mMultiSamplingCount, &mMultiSamplingQualityLevels);  
if (mMultiSamplingQualityLevels == 0)  
{  
    throw GameException("Unsupported multi-sampling quality");  
}  
  
DXGI_SWAP_CHAIN_DESC1 swapChainDesc;  
ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));  
swapChainDesc.Width = mScreenWidth;  
swapChainDesc.Height = mScreenHeight;  
swapChainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;  
  
if (mMultiSamplingEnabled)  
{  
    swapChainDesc.SampleDesc.Count = mMultiSamplingCount;  
    swapChainDesc.SampleDesc.Quality = mMultiSamplingQualityLevels - 1;  
}  
else  
{  
    swapChainDesc.SampleDesc.Count = 1;  
    swapChainDesc.SampleDesc.Quality = 0;  
}  
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;  
swapChainDesc.BufferCount = 1;  
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
```

---

```
typedef struct DXGI_SWAP_CHAIN_FULLSCREEN_DESC
{
    DXGI_RATIONAL RefreshRate;
    DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;
    DXGI_MODE_SCALING Scaling;
    BOOL Windowed;
} DXGI_SWAP_CHAIN_FULLSCREEN_DESC;
```

```
DXGI_SWAP_CHAIN_FULLSCREEN_DESC fullScreenDesc;  
ZeroMemory(&fullScreenDesc, sizeof(fullScreenDesc));  
fullScreenDesc.RefreshRate.Numerator = mFrameRate;  
fullScreenDesc.RefreshRate.Denominator = 1;  
fullScreenDesc.Windowed = !mIsFullScreen;
```

---

```
HRESULT CreateSwapChainForHwnd(
    IUnknown *pDevice,
    HWND hWnd,
    const DXGI_SWAP_CHAIN_DESC1 *pDesc,
    const DXGI_SWAP_CHAIN_FULLSCREEN_DESC *pFullscreenDesc,
    IDXGIOOutput *pRestrictToOutput,
    IDXGISwapChain1 **ppSwapChain);
```

```
#define ReleaseObject(object) if((object) != NULL) { object->Release();  
object = NULL; }  
  
IDXGIDevice* dxgiDevice = nullptr;  
if (FAILED(hr = mDirect3DDevice->QueryInterface(_uuidof(IDXGIDevice),  
reinterpret_cast<void**>(&dxgiDevice)))  
{  
    throw GameException("ID3D11Device::QueryInterface() failed", hr);  
}  
  
IDXGIAdapter *dxgiAdapter = nullptr;  
if (FAILED(hr = dxgiDevice->GetParent(_uuidof(IDXGIAdapter),  
reinterpret_cast<void**>(&dxgiAdapter)))  
{  
    ReleaseObject(dxgiDevice);  
    throw GameException("IDXGIDevice::GetParent() failed retrieving  
adapter.", hr);  
}  
  
IDXGIFactory2* dxgiFactory = nullptr;  
if (FAILED(hr = dxgiAdapter->GetParent(_uuidof(IDXGIFactory2),  
reinterpret_cast<void**>(&dxgiFactory)))  
{  
    ReleaseObject(dxgiDevice);  
    ReleaseObject(dxgiAdapter);  
    throw GameException("IDXGIAdapter::GetParent() failed retrieving  
factory.", hr);  
}  
  
IDXGISwapChain1* mSwapChain;  
if (FAILED(hr = dxgiFactory->CreateSwapChainForHwnd(dxgiDevice,  
mWindowHandle, &swapChainDesc, &fullScreenDesc, nullptr, &mSwapChain))  
{  
    ReleaseObject(dxgiDevice);  
    ReleaseObject(dxgiAdapter);  
    ReleaseObject(dxgiFactory);  
    throw GameException("IDXGIDevice::CreateSwapChainForHwnd()  
failed.", hr);  
}  
  
ReleaseObject(dxgiDevice);  
ReleaseObject(dxgiAdapter);  
ReleaseObject(dxgiFactory);
```

```
HRESULT CreateRenderTargetView(  
    ID3D11Resource *pResource,  
    const D3D11_RENDER_TARGET_VIEW_DESC *pDesc,  
    ID3D11RenderTargetView **ppRTView);
```

```
ID3D11Texture2D* backBuffer;
if (FAILED(hr = mSwapChain->GetBuffer(0, __uuidof(ID3D11Texture2D),
reinterpret_cast<void**>(&backBuffer))))
{
    throw GameException("IDXGISwapChain::GetBuffer() failed.", hr);
}

if (FAILED(hr = mDirect3DDevice->CreateRenderTargetView(backBuffer,
nullptr, &mRenderTargetView)))
{
    ReleaseObject(backBuffer);
    throw GameException("IDXGIDevice::CreateRenderTargetView()
failed.", hr);
}

ReleaseObject(backBuffer);
```

---

```
typedef struct D3D11_TEXTURE2D_DESC
{
    UINT Width;
    UINT Height;
    UINT MipLevels;
    UINT ArraySize;
    DXGI_FORMAT Format;
    DXGI_SAMPLE_DESC SampleDesc;
    D3D11_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
} D3D11_TEXTURE2D_DESC;
```

```
D3D11_TEXTURE2D_DESC depthStencilDesc;
ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));
depthStencilDesc.Width = mScreenWidth;
depthStencilDesc.Height = mScreenHeight;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.ArraySize = 1;
depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthStencilDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;

if (mMultiSamplingEnabled)
{
    depthStencilDesc.SampleDesc.Count = mMultiSamplingCount;
    depthStencilDesc.SampleDesc.Quality = mMultiSamplingQualityLevels - 1;
}
else
{
    depthStencilDesc.SampleDesc.Count = 1;
    depthStencilDesc.SampleDesc.Quality = 0;
}
```

---

```
HRESULT CreateTexture2D(
    const D3D11_TEXTURE2D_DESC *pDesc,
    const D3D11_SUBRESOURCE_DATA *pInitialData,
    ID3D11Texture2D **ppTexture2D);
```

```
ID3D11Texture2D* mDepthStencilBuffer;
ID3D11DepthStencilView* mDepthStencilView;

if (FAILED(hr = mDirect3DDevice->CreateTexture2D(&depthStencilDesc,
nullptr, &mDepthStencilBuffer)))
{
    throw GameException("IDXGIDevice::CreateTexture2D() failed.", hr);
}

if (FAILED(hr = mDirect3DDevice->CreateDepthStencilView(mDepthStencil
Buffer, nullptr, &mDepthStencilView)))
{
    throw GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
}
```

---

```
void OMSetRenderTargets(
    UINT NumViews,
    ID3D11RenderTargetView *const *ppRenderTargetViews,
    ID3D11DepthStencilView *pDepthStencilView);
```

```
mDirect3DDeviceContext->OMSetRenderTargets(1, &mRenderTargetView,  
mDepthStencilView);
```

---

```
typedef struct D3D11_VIEWPORT
{
    FLOAT TopLeftX;
    FLOAT TopLeftY;
    FLOAT Width;
    FLOAT Height;
    FLOAT MinDepth;
    FLOAT MaxDepth;
} D3D11_VIEWPORT;
```

```
D3D11_VIEWPORT mViewport;
mViewport.TopLeftX = 0.0f;
mViewport.TopLeftY = 0.0f;
mViewport.Width = static_cast<float>(mScreenWidth);
mViewport.Height = static_cast<float>(mScreenHeight);
mViewport.MinDepth = 0.0f;
mViewport.MaxDepth = 1.0f;

mDirect3DDeviceContext->RSSetViewports(1, &mViewport);
```

---

```
#pragma once

#include "Common.h"
#include "GameClock.h"
#include "GameTime.h"

namespace Library
{
    class Game
    {
        public:
            /* ... Previously presented members removed for brevity ... */

            ID3D11Device1* Direct3DDevice() const;
            ID3D11DeviceContext1* Direct3DDeviceContext() const;
            bool DepthBufferEnabled() const;
            bool IsFullScreen() const;
            const D3D11_TEXTURE2D_DESC& BackBufferDesc() const;
            const D3D11_VIEWPORT& Viewport() const;

        protected:
            /* ... Previously presented members removed for brevity ... */

            virtual void InitializeDirectX();

            static const UINT DefaultFrameRate;
            static const UINT DefaultMultiSamplingCount;

            D3D_FEATURE_LEVEL mFeatureLevel;
            ID3D11Device1* mDirect3DDevice;
            ID3D11DeviceContext1* mDirect3DDeviceContext;
            IDXGISwapChain1* mSwapChain;

            UINT mFrameRate;
            bool mIsFullScreen;
            bool mDepthStencilBufferEnabled;
            bool mMultiSamplingEnabled;
            UINT mMultiSamplingCount;
            UINT mMultiSamplingQualityLevels;

            ID3D11Texture2D* mDepthStencilBuffer;
            D3D11_TEXTURE2D_DESC mBackBufferDesc;
            ID3D11RenderTargetView* mRenderTargetView;
            ID3D11DepthStencilView* mDepthStencilView;
            D3D11_VIEWPORT mViewport;
    };
}
```

```
#pragma once

#include <windows.h>
#include <exception>
#include <cassert>
#include <string>
#include <vector>
#include <map>
#include <memory>

#include <d3d11_1.h>
#include <DirectXMath.h>
#include <DirectXPackedVector.h>

#define DeleteObject(object) if((object) != NULL) { delete object; \
object = NULL; }
#define DeleteObjects(objects) if((objects) != NULL) { delete[] \
objects; objects = NULL; }
#define ReleaseObject(object) if((object) != NULL) { object->Release(); \
object = NULL; }

namespace Library
{
    typedef unsigned char byte;
}

using namespace DirectX;
using namespace DirectX::PackedVector;
```

---

```
#include "Game.h"
#include "GameException.h"

namespace Library
{
    const UINT Game::DefaultScreenWidth = 1024;
    const UINT Game::DefaultScreenHeight = 768;
    const UINT Game::DefaultFrameRate = 60;
    const UINT Game::DefaultMultiSamplingCount = 4;

    Game::Game(HINSTANCE instance, const std::wstring& windowClass,
    const std::wstring& windowTitle, int showCommand)
        : mInstance(instance), mWindowClass(windowClass),
        mWindowTitle(windowTitle), mShowCommand(showCommand),
        mWindowHandle(), mWindow(),
        mScreenWidth(DefaultScreenWidth), mScreenHeight(DefaultScreen
Height),
        mGameClock(), mGameTime(),
        mFeatureLevel(D3D_FEATURE_LEVEL_9_1),
        mDirect3DDevice(nullptr),
```

```
mDirect3DDeviceContext(nullptr), mSwapChain(nullptr),
mFrameRate(DefaultFrameRate), mIsFullScreen(false),
mDepthStencilBufferEnabled(false),
mMultiSamplingEnabled(false),
mMultiSamplingCount(DefaultMultiSamplingCount),
mMultiSamplingQualityLevels(0),
mDepthStencilBuffer(nullptr), mRenderTargetView(nullptr),
mDepthStencilView(nullptr), mViewport()

{

void Game::Run()
{
    InitializeWindow();
    InitializeDirectX();
    Initialize();

    MSG message;

    ZeroMemory(&message, sizeof(message));

    mGameClock.Reset();

    while(message.message != WM_QUIT)
    {
        if (PeekMessage(&message, nullptr, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&message);
            DispatchMessage(&message);
        }
    }
}
```

```
        }

    else
    {
        mGameClock.UpdateGameTime(mGameTime);
        Update(mGameTime);
        Draw(mGameTime);
    }
}

Shutdown();
}

void Game::Shutdown()
{
    ReleaseObject(mRenderTargetView);
    ReleaseObject(mDepthStencilView);
    ReleaseObject(mSwapChain);
    ReleaseObject(mDepthStencilBuffer);

    if (mDirect3DDeviceContext != nullptr)
    {
        mDirect3DDeviceContext->ClearState();
    }

    ReleaseObject(mDirect3DDeviceContext);
}
```

```
    ReleaseObject(mDirect3DDevice);

    UnregisterClass(mWindowClass.c_str(), mWindow.hInstance);
}

void Game::InitializeDirectX()
{
    HRESULT hr;
    UINT createDeviceFlags = 0;

#ifndef DEBUG
#define DEBUG
#endif

    D3D_FEATURE_LEVEL featureLevels[] = {
        D3D_FEATURE_LEVEL_11_0,
        D3D_FEATURE_LEVEL_10_1,
        D3D_FEATURE_LEVEL_10_0
    };

    // Step 1: Create the Direct3D device and device context
    // interfaces
    ID3D11Device* direct3DDevice = nullptr;
    ID3D11DeviceContext* direct3DDeviceContext = nullptr;
    if (FAILED(hr = D3D11CreateDevice(NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, createDeviceFlags, featureLevels,
        ARRAYSIZE(featureLevels), D3D11_SDK_VERSION, &direct3DDevice,
        &mFeatureLevel, &direct3DDeviceContext)))

```

```
{  
    throw GameException("D3D11CreateDevice() failed", hr);  
}  
  
if (FAILED(hr = direct3DDevice->QueryInterface(  
    __uuidof(ID3D11Device1), reinterpret_cast<void**>(&mDirect3DDevice)))  
{  
    throw GameException("ID3D11Device::QueryInterface()  
failed", hr);  
}  
  
if (FAILED(hr = direct3DDeviceContext->QueryInterface(  
    __uuidof(ID3D11DeviceContext1), reinterpret_cast<void**>  
(&mDirect3DDeviceContext)))  
{  
    throw GameException("ID3D11Device::QueryInterface()  
failed", hr);  
}  
  
ReleaseObject(direct3DDevice);  
ReleaseObject(direct3DDeviceContext);  
  
// Step 2: Check for multisampling support  
mDirect3DDevice->CheckMultisampleQualityLevels(DXGI_FORMAT_  
R8G8B8A8_UNORM, mM�tiSamplingCount, &mMultiSamplingQualityLevels);  
if (mMultiSamplingQualityLevels == 0)  
{  
    throw GameException("Unsupported multi-sampling quality");  
}
```

```
DXGI_SWAP_CHAIN_DESC1 swapChainDesc;
ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));
swapChainDesc.Width = mScreenWidth;
swapChainDesc.Height = mScreenHeight;
swapChainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;

if (mMultiSamplingEnabled)
{
    swapChainDesc.SampleDesc.Count = mMultiSamplingCount;
    swapChainDesc.SampleDesc.Quality =
mMultiSamplingQualityLevels - 1;
}
else
{
    swapChainDesc.SampleDesc.Count = 1;
    swapChainDesc.SampleDesc.Quality = 0;
}

swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
swapChainDesc.BufferCount = 1;
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;

// Step 3: Create the swap chain
IDXGIDevice* dxgiDevice = nullptr;
if (FAILED(hr = mDirect3DDevice->QueryInterface(
uuidof(IDXGIDevice), reinterpret_cast<void**>(&dxgiDevice))))
{
    throw GameException("ID3D11Device::QueryInterface() failed", hr);
}
```

```
    IDXGIAAdapter *dxgiAdapter = nullptr;
    if (FAILED(hr = dxgiDevice->GetParent(__uuidof(IDXGIAAdapter),
reinterpret_cast<void**>(&dxgiAdapter))))
    {
        ReleaseObject(dxgiDevice);
        throw GameException("IDXGIDevice::GetParent() failed
retrieving adapter.", hr);
    }

    IDXGIFactory2* dxgiFactory = nullptr;
    if (FAILED(hr = dxgiAdapter->GetParent(__uuidof(IDXGIFactory2),
reinterpret_cast<void**>(&dxgiFactory))))
    {
        ReleaseObject(dxgiDevice);
        ReleaseObject(dxgiAdapter);
        throw GameException("IDXGIAAdapter::GetParent() failed
retrieving factory.", hr);
    }

    DXGI_SWAP_CHAIN_FULLSCREEN_DESC fullScreenDesc;
    ZeroMemory(&fullScreenDesc, sizeof(fullScreenDesc));
    fullScreenDesc.RefreshRate.Numerator = mFrameRate;
    fullScreenDesc.RefreshRate.Denominator = 1;
    fullScreenDesc.Windowed = !mIsFullScreen;
    if (FAILED(hr = dxgiFactory->CreateSwapChainForHwnd(dxgiDevice,
mWindowHandle, &swapChainDesc, &fullScreenDesc, nullptr, &mSwapChain)))
    {
        ReleaseObject(dxgiDevice);
        ReleaseObject(dxgiAdapter);
        ReleaseObject(dxgiFactory);
        throw GameException("IDXGIDevice::CreateSwapChainForHwnd()
failed.", hr);
    }

    ReleaseObject(dxgiDevice);
    ReleaseObject(dxgiAdapter);
    ReleaseObject(dxgiFactory);

    // Step 4: Create the render target view
    ID3D11Texture2D* backBuffer;
    if (FAILED(hr = mSwapChain->GetBuffer(0, __
uuidof(ID3D11Texture2D), reinterpret_cast<void**>(&backBuffer))))
    {
        throw GameException("IDXGISwapChain::GetBuffer() failed.", hr);
    }
```

```
    backBuffer->GetDesc(&mBackBufferDesc);

    if (FAILED(hr = mDirect3DDevice->CreateRenderTargetView
(backBuffer, nullptr, &mRenderTargetView)))
    {
        ReleaseObject(backBuffer);
        throw GameException("IDXGIDevice::CreateRenderTargetView() failed.", hr);
    }

    ReleaseObject(backBuffer);

    // Step 5: Create the depth-stencil view
    if (mDepthStencilBufferEnabled)
    {
        D3D11_TEXTURE2D_DESC depthStencilDesc;
        ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));
        depthStencilDesc.Width = mScreenWidth;
        depthStencilDesc.Height = mScreenHeight;
        depthStencilDesc.MipLevels = 1;
        depthStencilDesc.ArraySize = 1;
        depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
        depthStencilDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
        depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;
```

```
    if (mMultiSamplingEnabled)
    {
        depthStencilDesc.SampleDesc.Count =
mMultiSamplingCount;
        depthStencilDesc.SampleDesc.Quality =
mMultiSamplingQualityLevels - 1;
    }
    else
    {
        depthStencilDesc.SampleDesc.Count = 1;
        depthStencilDesc.SampleDesc.Quality = 0;
    }

    if (FAILED(hr = mDirect3DDevice->CreateTexture2D
(&depthStencilDesc, nullptr, &mDepthStencilBuffer)))
    {
        throw GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
    }

    if (FAILED(hr = mDirect3DDevice->CreateDepthStencilView
(mDepthStencilBuffer, nullptr, &mDepthStencilView)))
    {
        throw GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
    }
}

// Step 6: Bind the render target and depth-stencil views to OM
stage
mDirect3DDeviceContext->OMSetRenderTargets(1,
&mRenderTargetView, mDepthStencilView);

mViewport.TopLeftX = 0.0f;
mViewport.TopLeftY = 0.0f;
mViewport.Width = static_cast<float>(mScreenWidth);
mViewport.Height = static_cast<float>(mScreenHeight);
mViewport.MinDepth = 0.0f;
mViewport.MaxDepth = 1.0f;

// Step 7: Set the viewport
mDirect3DDeviceContext->RSSetViewports(1, &mViewport);
}
```

---

```
#pragma once

#include "Common.h"
#include "Game.h"

using namespace Library;

namespace Rendering
{
    class RenderingGame : public Game
    {
        public:
            RenderingGame(HINSTANCE instance, const std::wstring& windowClass, const std::wstring& windowTitle, int showCommand);
            ~RenderingGame();

            virtual void Initialize() override;
            virtual void Update(const GameTime& gameTime) override;
            virtual void Draw(const GameTime& gameTime) override;

        private:
            static const XMVECTORF32 BackgroundColor;
    };
}
```

---

```
#include "RenderingGame.h"
#include "GameException.h"

namespace Rendering
{
    const XMVECTORF32 RenderingGame::BackgroundColor = { 0.392f,
0.584f, 0.929f, 1.0f };

    RenderingGame::RenderingGame(HINSTANCE instance, const
std::wstring& windowClass, const std::wstring& windowTitle, int
showCommand)
        : Game(instance, windowClass, windowTitle, showCommand)
    {
        mDepthStencilBufferEnabled = true;
        mMutiSamplingEnabled = true;
    }

    RenderingGame::~RenderingGame()
    {
    }

    void RenderingGame::Initialize()
    {
        Game::Initialize();
    }

    void RenderingGame::Update(const GameTime &gameTime)
    {
        Game::Update(gameTime);
    }

    void RenderingGame::Draw(const GameTime &gameTime)
    {
        mDirect3DDeviceContext->ClearRenderTargetView(mRenderTarget
View, reinterpret_cast<const float*>(&BackgroundColor));
        mDirect3DDeviceContext->ClearDepthStencilView(mDepthStencil
View, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

        Game::Draw(gameTime);

        HRESULT hr = mSwapChain->Present(0, 0);
        if (FAILED(hr))
        {
            throw GameException("IDXGISwapChain::Present() failed.", hr);
        }
    }
}
```

```
#include <memory>
#include "GameException.h"
#include "RenderingGame.h"

#if defined(DEBUG) || defined(_DEBUG)
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#endif

using namespace Library;
using namespace Rendering;

int WINAPI WinMain(HINSTANCE instance, HINSTANCE previousInstance,
LPSTR commandLine, int showCommand)
{
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    std::unique_ptr<RenderingGame> game(new RenderingGame(instance,
L"RenderingClass", L"Real-Time 3D Rendering", showCommand));

    try
    {
        game->Run();
    }
    catch (GameException ex)
    {
        MessageBox(game->WindowHandle(), ex.whatw().c_str(),
game->WindowTitle().c_str(), MB_ABORTRETRYIGNORE);
    }

    return 0;
}
```

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;
    class GameTime;

    class GameComponent : public RTTI
    {
        RTTI_DECLARATIONS(GameComponent, RTTI)

        public:
            GameComponent();
            GameComponent(Game& game);
            virtual ~GameComponent();

            Game* GetGame();
            void SetGame(Game& game);
            bool Enabled() const;
            void SetEnabled(bool enabled);

            virtual void Initialize();
            virtual void Update(const GameTime& gameTime);

        protected:
            Game* mGame;
            bool mEnabled;

        private:
            GameComponent(const GameComponent& rhs);
            GameComponent& operator=(const GameComponent& rhs);
    };
}
```

---

```
#include "GameComponent.h"
#include "GameTime.h"

namespace Library
{
    RTTI_DEFINITIONS(GameComponent)

    GameComponent::GameComponent()
        : mGame(nullptr), mEnabled(true)
    {

    }

    GameComponent::GameComponent(Game& game)
        : mGame(&game), mEnabled(true)
    {

    }

    GameComponent::~GameComponent()
    {

    }

    Game* GameComponent::GetGame()
    {
        return mGame;
    }

    void GameComponent::SetGame(Game& game)
    {
        mGame = &game;
    }
}
```

```
bool GameComponent::Enabled() const
{
    return mEnabled;
}

void GameComponent::SetEnabled(bool enabled)
{
    mEnabled = enabled;
}

void GameComponent::Initialize()
{
}

void GameComponent::Update(const GameTime& gameTime)
{
}
}
```

---

```
#pragma once

#include <string>

namespace Library
{
    class RTTI
    {
public:
    virtual const unsigned int& TypeIdInstance() const = 0;

    virtual RTTI* QueryInterface(const unsigned id) const
    {
        return nullptr;
    }

    virtual bool Is(const unsigned int id) const
    {
        return false;
    }

    virtual bool Is(const std::string& name) const
    {
        return false;
    }

    template <typename T>
    T* As() const
    {
        if (Is(T::TypeIdClass()))
        {
```

```
        return (T*)this;
    }

    return nullptr;
}
};

#define RTTI_DECLARATIONS(Type, ParentType)
public:
    typedef ParentType Parent;
    static std::string TypeName() { return std::string
(#Type); }
    virtual const unsigned int& TypeIdInstance() const { return
Type::TypeIdClass(); }
    static const unsigned int& TypeIdClass() { return
sRunTimeTypeId; }
    virtual Library::RTTI* QueryInterface( const unsigned int
id ) const
    {
        if (id == sRunTimeTypeId)
            { return (RTTI*)this; }
        else
            { return Parent::QueryInterface(id); }
    }
    virtual bool Is(const unsigned int id) const
    {
        if (id == sRunTimeTypeId)
            { return true; }
        else
            { return Parent::Is(id); }
    }
    virtual bool Is(const std::string& name) const
    {
        if (name == TypeName())
            { return true; }
        else
            { return Parent::Is(name); }
    }
private:
    static unsigned int sRunTimeTypeId;

#define RTTI_DEFINITIONS(Type) unsigned int Type::sRunTimeTypeId =
(unsigned int)& Type::sRunTimeTypeId;
}
```

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class Camera;

    class DrawableGameComponent : public GameComponent
    {
        RTTI_DECLARATIONS(DrawableGameComponent, GameComponent)

    public:
        DrawableGameComponent();
        DrawableGameComponent(Game& game);
        DrawableGameComponent(Game& game, Camera& camera);
        virtual ~DrawableGameComponent();

        bool Visible() const;
        void SetVisible(bool visible);

        Camera* GetCamera();
        void SetCamera(Camera* camera);

        virtual void Draw(const GameTime& gameTime);

    protected:
        bool mVisible;
        Camera* mCamera;

    private:
        DrawableGameComponent(const DrawableGameComponent& rhs);
        DrawableGameComponent& operator=(const DrawableGameComponent&
rhs);
    };
}
```

---

```
class Game
{
public:
    /* ... Previously presented members removed for brevity ... */
    const std::vector<GameComponent*>& Components() const;

protected:
    /* ... Previously presented members removed for brevity ... */
    std::vector<GameComponent*> mComponents;
};
```

---

```
/* ... Previously presented members removed for brevity ... */

void Game::Initialize()
{
    for (GameComponent* component : mComponents)
    {
        component->Initialize();
    }
}

void Game::Update(const GameTime& gameTime)
{
    for (GameComponent* component : mComponents)
    {
        if (component->Enabled())
        {
            component->Update(gameTime);
        }
    }
}

void Game::Draw(const GameTime& gameTime)
{
    for (GameComponent* component : mComponents)
    {
        DrawableGameComponent* drawableGameComponent =
component->As<DrawableGameComponent>();
        if (drawableGameComponent != nullptr &&
drawableGameComponent->Visible())
        {
            drawableGameComponent->Draw(gameTime);
        }
    }
}
```

---

```
#pragma once

#include "DrawableGameComponent.h"

namespace DirectX
{
    class SpriteBatch;
    class SpriteFont;
}

namespace Library
{
    class FpsComponent : public DrawableGameComponent
    {
        RTTI_DECLARATIONS(FpsComponent, DrawableGameComponent)

        public:
            FpsComponent(Game& game);
            ~FpsComponent();

            XMFLOAT2& TextPosition();
            int FrameRate() const;

            virtual void Initialize() override;
            virtual void Update(const GameTime& gameTime) override;
            virtual void Draw(const GameTime& gameTime) override;

        private:
            FpsComponent();
            FpsComponent(const FpsComponent& rhs);
            FpsComponent& operator=(const FpsComponent& rhs);

            SpriteBatch* mSpriteBatch;
            SpriteFont* mSpriteFont;
            XMFLOAT2 mTextPosition;

            int mFrameCount;
            int mFrameRate;
            double mLastTotalElapsedTime;
    };
}
```

---

```
mSpriteBatch->Begin();  
mSpriteBatch->Draw(texture, position);  
mSpriteBatch->End();
```

```
MakeSpriteFont.exe "Arial" Arial_14-Regular.spritefont /FontSize:14
```

```
mkdir "$(SolutionDir)..\\lib\\"
copy "$(TargetPath)" "$(SolutionDir)..\\lib\\"

mkdir "$(SolutionDir)..\\content\\"
IF EXIST "$(ProjectDir)Content" xcopy /E /Y "$(ProjectDir)Content"
"$(SolutionDir)..\\content\\"
IF EXIST "$(TargetDir)Content" xcopy /E /Y "$(TargetDir)Content"
"$(SolutionDir)..\\content\\"
```

---

```
mkdir "$(OutDir)Content"
IF EXIST "$(SolutionDir)..\\content" xcopy /E /Y "$(SolutionDir)..\\
content" "$(OutDir)Content\\"
IF EXIST "$(ProjectDir)content" xcopy /E /Y "$(ProjectDir)Content"
"$(OutDir)Content\\"
```

---

```
mSpriteFont = new SpriteFont(mGame->Direct3DDevice() ,  
L"Content\\Fonts\\Arial_14-Regular.spritefont");
```

```
#include "FpsComponent.h"
#include <sstream>
#include <iomanip>
#include <SpriteBatch.h>
#include <SpriteFont.h>
#include "Game.h"
#include "Utility.h"

namespace Library
{
    RTTI_DEFINITIONS(FpsComponent)

    FpsComponent::FpsComponent(Game& game)
        : DrawableGameComponent(game), mSpriteBatch(nullptr),
          mSpriteFont(nullptr), mTextPosition(0.0f, 60.0f),
          mFrameCount(0), mFrameRate(0), mLastTotalElapsedTime(0.0)
    {
    }

    FpsComponent::~FpsComponent()
    {
        DeleteObject(mSpriteFont);
        DeleteObject(mSpriteBatch);
    }

    XMFLOAT2& FpsComponent::TextPosition()
    {
```

```
        return mTextPosition;
    }

    int FpsComponent::FrameRate() const
    {
        return mFrameCount;
    }

    void FpsComponent::Initialize()
    {
        SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

        mSpriteBatch = new SpriteBatch(mGame->Direct3DDeviceContext());
        mSpriteFont = new SpriteFont(mGame->Direct3DDevice(),
L"Content\\Fonts\\Arial_14-Regular.spritefont");
    }

    void FpsComponent::Update(const GameTime& gameTime)
    {
        if (gameTime.TotalGameTime() - mLastTotalElapsedTime >= 1)

        {
            mLastTotalElapsedTime = gameTime.TotalGameTime();
            mFrameRate = mFrameCount;
            mFrameCount = 0;
        }

        mFrameCount++;
    }

    void FpsComponent::Draw(const GameTime& gameTime)
    {
        mSpriteBatch->Begin();

        std::wostringstream fpsLabel;
        fpsLabel << std::setprecision(4) << L"Frame Rate: " <<
mFrameRate << "      Total Elapsed Time: " << gameTime.TotalGameTime();
        mSpriteFont->DrawString(mSpriteBatch, fpsLabel.str().c_str(),
mTextPosition);

        mSpriteBatch->End();
    }
}
```

---

```
void RenderingGame::Initialize()
{
    mFpsComponent = new FpsComponent(*this);
    mComponents.push_back(mFpsComponent);

    Game::Initialize();
}

void RenderingGame::Shutdown()
{
    DeleteObject(mFpsComponent);

    Game::Shutdown();
}
```

---

```
HRESULT DirectInput8Create(  
    HINSTANCE hinst, DWORD dwVersion, REFIID riidltf,  
    LPVOID *ppvOut, LPUNKNOWN punkOuter);
```

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(mInstance, DIRECTINPUT_VERSION,
        IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    /* ... Previously presented statements removed for brevity ... */
}

void RenderingGame::Shutdown()
{
    /* ... Previously presented statements removed for brevity ... */

    ReleaseObject(mDirectInput);

    Game::Shutdown();
}
```

---

```
HRESULT CreateDevice(  
    REFGUID rguid,  
    LPDIRECTINPUTDEVICE * lplpDirectInputDevice,  
    LPUNKNOWN pUnkOuter);
```

```
HRESULT SetCooperativeLevel(  
    HWND hwnd,  
    DWORD dwFlags);
```

```
void Keyboard::Initialize()
{
    if (FAILED(mDirectInput->CreateDevice(GUID_SysKeyboard, &mDevice,
nullptr)))
    {
        throw GameException("IDIRECTINPUT8::CreateDevice() failed");
    }

    if (FAILED(mDevice->SetDataFormat(&c_dfDIKeyboard)))
    {
        throw GameException("IDIRECTINPUTDEVICE8::SetDataFormat()
failed");
    }

    if (FAILED(mDevice->SetCooperativeLevel(mGame->WindowHandle(),
DISCL_FOREGROUND | DISCL_NONEXCLUSIVE)))
    {
        throw GameException("IDIRECTINPUTDEVICE8::SetCooperativeLevel()
failed");
    }

    if (FAILED(mDevice->Acquire()))
    {
        throw GameException("IDIRECTINPUTDEVICE8::Acquire() failed");
    }
}
```

---

```
HRESULT GetDeviceState(  
    DWORD cbData,  
    LPVOID lpvData);
```

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class Keyboard : public GameComponent
    {
        RTTI_DECLARATIONS(Keyboard, GameComponent)

    public:
        Keyboard(Game& game, LPDIRECTINPUT8 directInput);
        ~Keyboard();

        const byte* const CurrentState() const;
        const byte* const LastState() const;

        virtual void Initialize() override;
        virtual void Update(const GameTime& gameTime) override;

        bool IsKeyUp(byte key) const;
        bool IsKeyDown(byte key) const;
        bool WasKeyUp(byte key) const;
        bool WasKeyDown(byte key) const;
        bool WasKeyPressedThisFrame(byte key) const;
        bool WasKeyReleasedThisFrame(byte key) const;
        bool IsKeyHeldDown(byte key) const;

    private:
        Keyboard();

        static const int KeyCount = 256;

        Keyboard(const Keyboard& rhs);

        LPDIRECTINPUT8 mDirectInput;
        LPDIRECTINPUTDEVICE8 mDevice;
        byte mCurrentState[KeyCount];
        byte mLastState[KeyCount];
    };
}
```

```
#include "Keyboard.h"
#include "Game.h"
#include "GameTime.h"
#include "GameException.h"

namespace Library
{
    RTTI_DEFINITIONS(Keyboard)

    Keyboard::Keyboard(Game& game, LPDIRECTINPUT8 directInput)
        : GameComponent(game), mDirectInput(directInput),
        mDevice(nullptr)
    {
        assert(mDirectInput != nullptr);
        ZeroMemory(mCurrentState, sizeof(mCurrentState));
        ZeroMemory(mLastState, sizeof(mLastState));
    }

    Keyboard::~Keyboard()
    {
        if (mDevice != nullptr)
        {
            mDevice->Unacquire();
            mDevice->Release();
            mDevice = nullptr;
        }
    }
}
```

```
const byte* const Keyboard::CurrentState() const
{
    return mcurrentState;
}

const byte* const Keyboard::LastState() const
{
    return mLastState;
}

void Keyboard::Initialize()
{
    if (FAILED(mDirectInput->CreateDevice(GUID_SysKeyboard,
&mDevice, nullptr)))
    {
        throw GameException("IDIRECTINPUT8::CreateDevice() failed");
    }

    if (FAILED(mDevice->SetDataFormat(&c_dfDIKeyboard)))
    {
        throw GameException("IDIRECTINPUTDEVICE8::SetDataFormat() failed");
    }
}
```

```
        if (FAILED(mDevice->SetCooperativeLevel(mGame->WindowHandle(),
DISCL_FOREGROUND| DISCL_NONEXCLUSIVE)))
{
    throw GameException("IDIRECTINPUTDEVICE8::"
SetCooperativeLevel() failed");
}

if (FAILED(mDevice->Acquire()))
{
    throw GameException("IDIRECTINPUTDEVICE8::Acquire()"
failed");
}
}

void Keyboard::Update(const GameTime& gameTime)
{
    if (mDevice != nullptr)
    {
        memcpy(mLastState, mCurrentState, sizeof(mCurrentState));

        if (FAILED(mDevice->GetDeviceState(sizeof(mCurrentState),
(LPVOID)mCurrentState)))
        {
            // Try to reaqcuire the device
            if (SUCCEEDED(mDevice->Acquire()))
            {
```

```
    mDevice->GetDeviceState(sizeof(mCurrentState),
(LPVOID)mCurrentState);
}
}
}

bool Keyboard::IsKeyUp(byte key) const
{
    return ((mCurrentState[key] & 0x80) == 0);
}

bool Keyboard::IsKeyDown(byte key) const
{
    return ((mCurrentState[key] & 0x80) != 0);
}

bool Keyboard::WasKeyUp(byte key) const
{
    return ((mLastState[key] & 0x80) == 0);
}

bool Keyboard::WasKeyDown(byte key) const
{
    return ((mLastState[key] & 0x80) != 0);
}

bool Keyboard::WasKeyPressedThisFrame(byte key) const
{
    return (IsKeyDown(key) && WasKeyUp(key));
}

bool Keyboard::WasKeyReleasedThisFrame(byte key) const
{
    return (IsKeyUp(key) && WasKeyDown(key));
}

bool Keyboard::IsKeyHeldDown(byte key) const
{
    return (IsKeyDown(key) && WasKeyDown(key));
}
```

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(&mInstance, DIRECTINPUT_VERSION,
IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    mKeyboard = new Keyboard(*this, mDirectInput);
    mComponents.push_back(mKeyboard);

    mFpsComponent = new FpsComponent(*this);
    mComponents.push_back(mFpsComponent);

    Game::Initialize();
}

void RenderingGame::Shutdown()
{
    DeleteObject(mKeyboard);
    DeleteObject(mFpsComponent);

    ReleaseObject(mDirectInput);

    Game::Shutdown();
}

void RenderingGame::Update(const GameTime &gameTime)
{
    if (mKeyboard->WasKeyPressedThisFrame(DIK_ESCAPE))
    {
        Exit();
    }

    Game::Update(gameTime);
}
```

---

```
typedef struct _DIMOUSESTATE {
    LONG     lX;
    LONG     lY;
    LONG     lZ;
    BYTE     rgButtons[4];
} DIMOUSESTATE;
```

```
enum MouseButtons
{
    MouseButtonsLeft = 0,
    MouseButtonsRight = 1,
    MouseButtonsMiddle = 2,
    MouseButtonsX1 = 3
};
```

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class GameTime;

    enum MouseButtons
    {
        MouseButtonsLeft = 0,
        MouseButtonsRight = 1,
        MouseButtonsMiddle = 2,
        MouseButtonsX1 = 3
    };

    class Mouse : public GameComponent
    {
        RTTI_DECLARATIONS(Mouse, GameComponent)

    public:
        Mouse(Game& game, LPDIRECTINPUT8 directInput);
        ~Mouse();

        LPDIMOUSESTATE CurrentState();
        LPDIMOUSESTATE LastState();

        virtual void Initialize() override;
        virtual void Update(const GameTime& gameTime) override;
    };
}
```

```
long X() const;
long Y() const;
long Wheel() const;

bool IsButtonUp(MouseButtons button) const;
bool IsButtonDown(MouseButtons button) const;
bool WasButtonUp(MouseButtons button) const;
bool WasButtonDown(MouseButtons button) const;
bool WasButtonPressedThisFrame(MouseButtons button) const;
bool WasButtonReleasedThisFrame(MouseButtons button) const;
bool IsButtonHeldDown(MouseButtons button) const;

private:
    Mouse();

    LPDIRECTINPUT8 mDirectInput;
    LPDIRECTINPUTDEVICE8 mDevice;
    DIMOUSESTATE mCurrentState;
    DIMOUSESTATE mLastState;

    long mX;
    long mY;
    long mWheel;
};

}
```

---

```
#include "Mouse.h"
#include "Game.h"
#include "GameTime.h"
#include "GameException.h"

namespace Library
{
    RTTI_DEFINITIONS(Mouse)

    Mouse::Mouse(Game& game, LPDIRECTINPUT8 directInput)
        : GameComponent(game), mDirectInput(directInput),
        mDevice(nullptr), mX(0), mY(0), mWheel(0)
    {
        assert(mDirectInput != nullptr);
        ZeroMemory(&mCurrentState, sizeof(mCurrentState));
        ZeroMemory(&mLastState, sizeof(mLastState));
    }

    Mouse::~Mouse()
    {
        if (mDevice != nullptr)
        {
            mDevice->Unacquire();
            mDevice->Release();
            mDevice = nullptr;
        }
    }

    LPDIMOUSESTATE Mouse::CurrentState()
    {
        return &mCurrentState;
    }
}
```

```
LPDIMOUSESTATE Mouse::LastState()
{
    return &mLastState;
}

long Mouse::X() const
{
    return mX;
}

long Mouse::Y() const
{
    return mY;
}

long Mouse::Wheel() const
{
    return mWheel;
}

void Mouse::Initialize()
{
    if (FAILED(mDirectInput->CreateDevice(GUID_SysMouse, &mDevice,
nullptr)))
    {
        throw GameException("IDIRECTINPUT8::CreateDevice() failed");
    }
}
```

```
    if (FAILED(mDevice->SetDataFormat(&c_dfDIMouse)))
    {
        throw GameException("IDIRECTINPUTDEVICE8::SetDataFormat() failed");
    }

    if (FAILED(mDevice->SetCooperativeLevel(mGame->WindowHandle(),
DISCL_FOREGROUND | DISCL_NONEXCLUSIVE)))
    {
        throw GameException("IDIRECTINPUTDEVICE8::SetCooperativeLevel() failed");
    }

    if (FAILED(mDevice->Acquire()))
    {
        throw GameException("IDIRECTINPUTDEVICE8::Acquire() failed");
    }
}

void Mouse::Update(const GameTime& gameTime)
{
    if (mDevice != nullptr)
    {
        memcpy(&mLastState, &mCurrentState, sizeof(mCurrentState));
    }
}
```

```
    if (FAILED(mDevice->GetDeviceState(sizeof(mCurrentState),
&mCurrentState)))
    {
        // Try to reaqcuire the device
        if (SUCCEEDED(mDevice->Acquire()))
        {
            if (FAILED(mDevice->GetDeviceState(sizeof
(mCurrentState), &mCurrentState)))
            {
                return;
            }
        }
    }

    // Accumulate positions
    mX += mCurrentState.lX;
    mY += mCurrentState.lY;
    mWheel += mCurrentState.lZ;
}

bool Mouse::IsButtonUp(MouseButtons button) const
{
    return ((mCurrentState.rgbButtons[button] & 0x80) == 0);
}

bool Mouse::IsButtonDown(MouseButtons button) const
{
```

```
        return ((mCurrentState.rgbButtons[button] & 0x80) != 0);
    }

bool Mouse::WasButtonUp(MouseButtons button) const
{
    return ((mLastState.rgbButtons[button] & 0x80) == 0);
}

bool Mouse::WasButtonDown(MouseButtons button) const
{
    return ((mLastState.rgbButtons[button] & 0x80) != 0);
}

bool Mouse::WasButtonPressedThisFrame(MouseButtons button) const
{
    return (IsButtonDown(button) && WasButtonUp(button));
}

bool Mouse::WasButtonReleasedThisFrame(MouseButtons button) const
{
    return (IsButtonUp(button) && WasButtonDown(button));
}

bool Mouse::IsButtonHeldDown(MouseButtons button) const
{
    return (IsButtonDown(button) && WasButtonDown(button));
}
```

---

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(&mInstance, DIRECTINPUT_VERSION,
IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    mKeyboard = new Keyboard(*this, mDirectInput);
    mComponents.push_back(mKeyboard);

    mMouse = new Mouse(*this, mDirectInput);
    mComponents.push_back(mMouse);

    mFpsComponent = new FpsComponent(*this);
    mComponents.push_back(mFpsComponent);

    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

    mSpriteBatch = new SpriteBatch(mDirect3DDeviceContext);
    mSpriteFont = new SpriteFont(mDirect3DDevice,
L"Content\\Fonts\\Arial_14-Regular.spritefont");

    Game::Initialize();
}

void RenderingGame::Shutdown()
{
    DeleteObject(mKeyboard);
    DeleteObject(mMouse);
    DeleteObject(mFpsComponent);
```

```
        DeleteObject(mSpriteFont);
        DeleteObject(mSpriteBatch);

        ReleaseObject(mDirectInput);
    }

void RenderingGame::Draw(const GameTime &gameTime)
{
    mDirect3DDeviceContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext->ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    Game::Draw(gameTime);

    mSpriteBatch->Begin();

    std::wostringstream mouseLabel;
    mouseLabel << L"Mouse Position: " << mMouse->X() << ", " <<
mMouse->Y() << " Mouse Wheel: " << mMouse->Wheel();
    mSpriteFont->DrawString(mSpriteBatch, mouseLabel.str().c_str(),
mMouseTextPosition);

    mSpriteBatch->End();

    HRESULT hr = mSwapChain->Present(0, 0);
    if (FAILED(hr))
    {
        throw GameException("IDXGISwapChain::Present() failed.", hr);
    }
}
```

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class ServiceContainer
    {
public:
    ServiceContainer();

    void AddService(UINT typeID, void* service);
    void RemoveService(UINT typeID);
    void* GetService(UINT typeID) const;

private:
    ServiceContainer(const ServiceContainer& rhs);
    ServiceContainer& operator=(const ServiceContainer& rhs);

    std::map<UINT, void*> mServices;
    };
}
```

---

```
#include "ServiceContainer.h"

namespace Library
{
    ServiceContainer::ServiceContainer()
        : mServices()
    {
    }

    void ServiceContainer::AddService(UINT typeID, void* service)
    {
        mServices.insert(std::pair<UINT, void*>(typeID, service));
    }

    void ServiceContainer::RemoveService(UINT typeID)
    {
        mServices.erase(typeID);
    }

    void* ServiceContainer::GetService(UINT typeID) const
    {
        std::map<UINT, void*>::const_iterator it = mServices.
find(typeID);

        return (it != mServices.end() ? it->second : nullptr);
    }
}
```

---

```
void RenderingGame::Initialize()
{
    if (FAILED(DirectInput8Create(&mInstance, DIRECTINPUT_VERSION, IID_
IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
    {
        throw GameException("DirectInput8Create() failed");
    }

    mKeyboard = new Keyboard(*this, mDirectInput);
    mComponents.push_back(mKeyboard);
    mServices.AddService(Keyboard::TypeIdClass(), mKeyboard);

    mMouse = new Mouse(*this, mDirectInput);
    mComponents.push_back(mMouse);
    mServices.AddService(Mouse::TypeIdClass(), mMouse);

    Game::Initialize();
}
```

---

```
mKeyboard = (Keyboard*)mGame->Services().GetService(Keyboard::TypeIdClass());
```

```
#pragma once

#include "GameComponent.h"

namespace Library
{
    class GameTime;

    class Camera : public GameComponent
    {
        RTTI_DECLARATIONS(Camera, GameComponent)

    public:
        Camera(Game& game);
        Camera(Game& game, float fieldOfView, float aspectRatio, float
nearPlaneDistance, float farPlaneDistance);

        virtual ~Camera();

        const XMFLOAT3& Position() const;
        const XMFLOAT3& Direction() const;
        const XMFLOAT3& Up() const;
        const XMFLOAT3& Right() const;
    };
}
```

```
XMVECTOR PositionVector() const;
XMVECTOR DirectionVector() const;
XMVECTOR UpVector() const;
XMVECTOR RightVector() const;

float AspectRatio() const;
float FieldOfView() const;
float NearPlaneDistance() const;
float FarPlaneDistance() const;

XMMATRIX ViewMatrix() const;
XMMATRIX ProjectionMatrix() const;
XMMATRIX ViewProjectionMatrix() const;

virtual void SetPosition(FLOAT x, FLOAT y, FLOAT z);
virtual void SetPosition(FXMVECTOR position);
virtual void SetPosition(const XMFLOAT3& position);

virtual void Reset();
virtual void Initialize() override;
virtual void Update(const GameTime& gameTime) override;
virtual void UpdateViewMatrix();
virtual void UpdateProjectionMatrix();
void ApplyRotation(CXMMATRIX transform);
void ApplyRotation(const XMFLOAT4X4& transform);
```

```
    static const float DefaultFieldOfView;
    static const float DefaultAspectRatio;
    static const float DefaultNearPlaneDistance;
    static const float DefaultFarPlaneDistance;

protected:
    float mFieldOfView;
    float mAspectRatio;
    float mNearPlaneDistance;
    float mFarPlaneDistance;

    XMFLOAT3 mPosition;
    XMFLOAT3 mDirection;
    XMFLOAT3 mUp;
    XMFLOAT3 mRight;

    XMFLOAT4X4 mViewMatrix;
    XMFLOAT4X4 mProjectionMatrix;

private:
    Camera(const Camera& rhs);
    Camera& operator=(const Camera& rhs);
};

}
```

---

```
#include "Camera.h"
#include "Game.h"
#include "GameTime.h"
#include "VectorHelper.h"
#include "MatrixHelper.h"
namespace Library
{
    RTTI_DEFINITIONS(Camera)

    const float Camera::DefaultFieldOfView = XM_PIDIV4;
    const float Camera::DefaultNearPlaneDistance = 0.01f;
    const float Camera::DefaultFarPlaneDistance = 1000.0f;

    Camera::Camera(Game& game)
        : GameComponent(game),
          mFieldOfView(DefaultFieldOfView),
          mAspectRatio(game.AspectRatio()),
          mNearPlaneDistance(DefaultNearPlaneDistance),
          mFarPlaneDistance(DefaultFarPlaneDistance),
          mPosition(), mDirection(), mUp(), mRight(),
          mViewMatrix(), mProjectionMatrix()
    {
    }

    XMMATRIX Camera::ViewProjectionMatrix() const
    {
        XMMATRIX viewMatrix = XMLoadFloat4x4(&mViewMatrix);
        XMMATRIX projectionMatrix = XMLoadFloat4x4(&mProjectionMatrix);

        return XMMatrixMultiply(viewMatrix, projectionMatrix);
    }
}
```

```
void Camera::SetPosition(FLOAT x, FLOAT y, FLOAT z)
{
    XMVECTOR position = XMVectorSet(x, y, z, 1.0f);
    SetPosition(position);
}

void Camera::SetPosition(FXMVECTOR position)
{
    XMStoreFloat3(&mPosition, position);
}

void Camera::SetPosition(const XMFLOAT3& position)
{
    mPosition = position;
}

void Camera::Reset()
{
    mPosition = Vector3Helper::Zero;
    mDirection = Vector3Helper::Forward;
    mUp = Vector3Helper::Up;
    mRight = Vector3Helper::Right;

    UpdateViewMatrix();
}

void Camera::Initialize()
```

```
        UpdateProjectionMatrix();
        Reset();
    }

    void Camera::Update(const GameTime& gameTime)
    {
        UpdateViewMatrix();
    }

    void Camera::UpdateViewMatrix()
    {
        XMVECTOR eyePosition = XMLoadFloat3(&mPosition);
        XMVECTOR direction = XMLoadFloat3(&mDirection);
        XMVECTOR upDirection = XMLoadFloat3(&mUp);

        XMMATRIX viewMatrix = XMMatrixLookToRH(eyePosition, direction,
upDirection);
        XMStoreFloat4x4(&mViewMatrix, viewMatrix);
    }

    void Camera::UpdateProjectionMatrix()
    {
        XMMATRIX projectionMatrix = XMMatrixPerspectiveFovRH
(mFieldOfView, mAspectRatio, mNearPlaneDistance, mFarPlaneDistance);
        XMStoreFloat4x4(&mProjectionMatrix, projectionMatrix);
    }
}
```

```
void Camera::ApplyRotation(CXMMATRIX transform)
{
    XMVECTOR direction = XMLoadFloat3(&mDirection);
    XMVECTOR up = XMLoadFloat3(&mUp);

    direction = XMVector3TransformNormal(direction, transform);
    direction = XMVector3Normalize(direction);

    up = XMVector3TransformNormal(up, transform);
    up = XMVector3Normalize(up);

    XMVECTOR right = XMVector3Cross(direction, up);
    up = XMVector3Cross(right, direction);

    XMStoreFloat3(&mDirection, direction);
    XMStoreFloat3(&mUp, up);
    XMStoreFloat3(&mRight, right);
}

void Camera::ApplyRotation(const XMFLOAT4X4& transform)
{
    XMMATRIX transformMatrix = XMLoadFloat4x4(&transform);
    ApplyRotation(transformMatrix);
}
}
```

---

```
#pragma once

#include "Camera.h"

namespace Library
{
    class Keyboard;
    class Mouse;

    class FirstPersonCamera : public Camera
    {
        RTTI_DECLARATIONS(FirstPersonCamera, Camera)

    public:
        FirstPersonCamera(Game& game);
        FirstPersonCamera(Game& game, float fieldOfView, float
aspectRatio, float nearPlaneDistance, float farPlaneDistance);

        virtual ~FirstPersonCamera();

        const Keyboard& GetKeyboard() const;
        void SetKeyboard(Keyboard& keyboard);

        const Mouse& GetMouse() const;
        void SetMouse(Mouse& mouse);
    };
}
```

```
    float& MouseSensitivity();
    float& RotationRate();
    float& MovementRate();

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;

    static const float DefaultMouseSensitivity;
    static const float DefaultRotationRate;
    static const float DefaultMovementRate;

protected:
    float mMouseSensitivity;
    float mRotationRate;
    float mMovementRate;

    Keyboard* mKeyboard;
    Mouse* mMouse;

private:
    FirstPersonCamera(const FirstPersonCamera& rhs);
    FirstPersonCamera& operator=(const FirstPersonCamera& rhs);
};

}
```

---

```
const float FirstPersonCamera::DefaultRotationRate =
XMConvertToRadians(1.0f);
const float FirstPersonCamera::DefaultMovementRate = 10.0f;
const float FirstPersonCamera::DefaultMouseSensitivity = 100.0f;

void FirstPersonCamera::Initialize()
{
    mKeyboard = (Keyboard*)mGame->Services().GetService(Keyboard::Type
IdClass());
    mMouse = (Mouse*)mGame->Services().GetService
(Mouse::TypeIdClass());

    Camera::Initialize();
}

void FirstPersonCamera::Update(const GameTime& gameTime)
{
    XMFLOAT2 movementAmount = Vector2Helper::Zero;
    if (mKeyboard != nullptr)
    {
        if (mKeyboard->IsKeyDown(DIK_W))
        {
            movementAmount.y = 1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_S))
        {
            movementAmount.y = -1.0f;
        }
    }
}
```

```
    if (mKeyboard->IsKeyDown(DIK_A))
    {
        movementAmount.x = -1.0f;
    }

    if (mKeyboard->IsKeyDown(DIK_D))
    {
        movementAmount.x = 1.0f;
    }
}

XMFLOAT2 rotationAmount = Vector2Helper::Zero;
if ((mMouse != nullptr) && (mMouse->IsButtonHeldDown
(MouseButtonsLeft)))
{
    LPDIMOUSESTATE mouseState = mMouse->CurrentState();
    rotationAmount.x = -mouseState->lX * mMouseSensitivity;
    rotationAmount.y = -mouseState->lY * mMouseSensitivity;
}

float elapsedTime = (float)gameTime.ElapsedGameTime();
XMVECTOR rotationVector = XMLoadFloat2(&rotationAmount) *
mRotationRate * elapsedTime;
XMVECTOR right = XMLoadFloat3(&mRight);

XMMATRIX pitchMatrix = XMMatrixRotationAxis(right,
XMVectorGetY(rotationVector));
XMMATRIX yawMatrix = XMMatrixRotationY(XMVectorGetX
(rotationVector));

ApplyRotation(XMMatrixMultiply(pitchMatrix, yawMatrix));

XMVECTOR position = XMLoadFloat3(&mPosition);
XMVECTOR movement = XMLoadFloat2(&movementAmount) * mMovementRate *
elapsedTime;

XMVECTOR strafe = right * XMVectorGetX(movement);
position += strafe;

XMVECTOR forward = XMLoadFloat3(&mDirection) *
XMVectorGetY(movement);
position += forward;

XMStoreFloat3(&mPosition, position);

Camera::Update(gameTime);
}
```

```
#include "RenderingGame.h"
#include "GameException.h"
#include "Keyboard.h"
#include "Mouse.h"
#include "FpsComponent.h"
#include "ColorHelper.h"
#include "FirstPersonCamera.h"
#include "TriangleDemo.h"

namespace Rendering
{
    const XMVECTORF32 RenderingGame::BackgroundColor =
        ColorHelper::CornflowerBlue;

    RenderingGame::RenderingGame(HINSTANCE instance, const
        std::wstring& windowClass, const std::wstring& windowTitle, int
        showCommand)
        : Game(instance, windowClass, windowTitle, showCommand),
        mFpsComponent(nullptr),
        mDirectInput(nullptr), mKeyboard(nullptr), mMouse(nullptr),
        mDemo(nullptr)
    {
        mDepthStencilBufferEnabled = true;
        mMutiSamplingEnabled = true;
    }

    RenderingGame::~RenderingGame()
    {
    }

    void RenderingGame::Initialize()
    {
        if (FAILED(DirectInput8Create(mInstance, DIRECTINPUT_VERSION,
            IID_IDirectInput8, (LPVOID*)&mDirectInput, nullptr)))
        {
            throw GameException("DirectInput8Create() failed");
        }

        mKeyboard = new Keyboard(*this, mDirectInput);
        mComponents.push_back(mKeyboard);
        mServices.AddService(Keyboard::TypeIdClass(), mKeyboard);

        mMouse = new Mouse(*this, mDirectInput);
        mComponents.push_back(mMouse);
    }
}
```

```
mServices.AddService(Mouse::TypeIdClass(), mMouse);

mCamera = new FirstPersonCamera(*this);
mComponents.push_back(mCamera);
mServices.AddService(Camera::TypeIdClass(), mCamera);

mFpsComponent = new FpsComponent(*this);
mComponents.push_back(mFpsComponent);

mDemo = new TriangleDemo(*this, *mCamera);
mComponents.push_back(mDemo);

Game::Initialize();

mCamera->SetPosition(0.0f, 0.0f, 5.0f);
}

void RenderingGame::Shutdown()
{
    DeleteObject(mDemo);
    DeleteObject(mKeyboard);
    DeleteObject(mMouse);
    DeleteObject(mFpsComponent);
    DeleteObject(mCamera);

    ReleaseObject(mDirectInput);

    Game::Shutdown();
}

void RenderingGame::Update(const GameTime &gameTime)
{
    if (mKeyboard->WasKeyPressedThisFrame(DIK_ESCAPE))
    {
        Exit();
    }

    Game::Update(gameTime);
}

void RenderingGame::Draw(const GameTime &gameTime)
{
    mDirect3DDeviceContext->ClearRenderTargetView(mRenderTarget
View, reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext->ClearDepthStencilView(mDepthStencil
```

```
View, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

Game::Draw(gameTime);

HRESULT hr = mSwapChain->Present(0, 0);
if (FAILED(hr))
{
    throw GameException("IDXGISwapChain::Present() failed.", hr);
}
}
```

---

```
#pragma once

#include "DrawableGameComponent.h"

using namespace Library;

namespace Rendering
{
    class TriangleDemo : public DrawableGameComponent
    {
        RTTI_DECLARATIONS(TriangleDemo, DrawableGameComponent)

    public:
        TriangleDemo(Game& game, Camera& camera);
        ~TriangleDemo();

        virtual void Initialize() override;
        virtual void Draw(const GameTime& gameTime) override;

    private:
        typedef struct _BasicEffectVertex
        {
            XMFLOAT4 Position;
            XMFLOAT4 Color;

            _BasicEffectVertex() { }

            _BasicEffectVertex(XMFLOAT4 position, XMFLOAT4 color)
                : Position(position), Color(color) { }
        } BasicEffectVertex;

        TriangleDemo();
        TriangleDemo(const TriangleDemo& rhs);
        TriangleDemo& operator=(const TriangleDemo& rhs);

        ID3DX11Effect* mEffect;
        ID3DX11EffectTechnique* mTechnique;
        ID3DX11EffectPass* mPass;
        ID3DX11EffectMatrixVariable* mWvpVariable;

        ID3D11InputLayout* mInputLayout;
        ID3D11Buffer* mVertexBuffer;

        XMFLOAT4X4 mWorldMatrix;
    };
}
```

```
HRESULT D3DCompileFromFile(  
    LPCWSTR pFileName,  
    D3D_SHADER_MACRO* pDefines,  
    ID3DInclude* pInclude,  
    LPCSTR pEntryPoint,  
    LPCSTR pTarget,  
    UINT Flags1,  
    UINT Flags2,  
    ID3DBlob** ppCode,  
    ID3DBlob** ppErrorMsgs);
```

```
SetCurrentDirectory(utility::ExecutableDirectory().c_str());  
  
UINT shaderFlags = 0;  
  
#if defined( DEBUG ) || defined( _DEBUG )  
    shaderFlags |= D3DCOMPILE_DEBUG;  
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;  
#endif  
  
ID3D10Blob* compiledShader = nullptr;  
ID3D10Blob* errorMessages = nullptr;  
HRESULT hr = D3DCompileFromFile(L"Content\\Effects\\BasicEffect.fx",  
nullptr, nullptr, nullptr, "fx_5_0", shaderFlags, 0, &compiledShader,  
&errorMessages);  
if (errorMessages != nullptr)  
{  
    GameException ex((char*)errorMessages->GetBufferPointer(), hr);  
    ReleaseObject(errorMessages);  
  
    throw ex;  
}  
  
if (FAILED(hr))  
{  
    throw GameException("D3DX11CompileFromFile() failed.", hr);  
}
```

---

```
cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

struct VS_INPUT
{
    float4 ObjectPosition: POSITION;
    float4 Color : COLOR;
};

struct VS_OUTPUT
{
    float4 Position: SV_Position;
    float4 Color : COLOR;
};

RasterizerState DisableCulling
{
    CullMode = NONE;
};

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.Color = IN.Color;

    return OUT;
}

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return IN.Color;
}

technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));

        SetRasterizerState(DisableCulling);
    }
}
```

```
HRESULT D3DX11CreateEffectFromMemory(
    LPCVOID pData,
    SIZE_T DataLength,
    UINT FXFlags,
    ID3D11Device *pDevice,
    ID3DX11Effect **ppEffect );
```

```
hr = D3DX11CreateEffectFromMemory(compiledShader->GetBufferPointer(),
compiledShader->GetBufferSize(), 0, mGame->Direct3DDevice(), &mEffect);
if (FAILED(hr))
{
    throw GameException("D3DX11CreateEffectFromMemory() failed.", hr);
}

ReleaseObject(compiledShader);
```

---

```
mTechnique = mEffect->GetTechniqueByName("main11");
if (mTechnique == nullptr)
{
    throw GameException("ID3DX11Effect::GetTechniqueByName() could not
find the specified technique.", hr);
}

mPass = mTechnique->GetPassByName("p0");
if (mPass == nullptr)
{
    throw GameException("ID3DX11EffectTechnique::GetPassByName() could
not find the specified pass.", hr);
}

ID3DX11EffectVariable* variable = mEffect->GetVariableByName
("WorldViewProjection");
if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName() could not
find the specified variable.", hr);
}

mWvpVariable = variable->AsMatrix();
if (mWvpVariable->IsValid() == false)
{
    throw GameException("Invalid effect variable cast.");
}
```

---

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

typedef struct _BasicEffectVertex
{
    XMFBYTE4 Position;
    XMFBYTE4 Color;

    _BasicEffectVertex() { }

    _BasicEffectVertex(XMFBYTE4 position, XMFBYTE4 color)
        : Position(position), Color(color) { }
} BasicEffectVertex;
```

```
typedef struct D3D11_INPUT_ELEMENT_DESC
{
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D11_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D11_INPUT_ELEMENT_DESC;
```

```
HRESULT CreateInputLayout(
    const D3D11_INPUT_ELEMENT_DESC *pInputElementDescs,
    UINT NumElements,
    const void *pShaderBytecodeWithInputSignature,
    SIZE_T BytecodeLength,
    ID3D11InputLayout **ppInputLayout);
```

```
D3DX11_PASS_DESC passDesc;
mPass->GetDesc(&passDesc);

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR",     0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

if (FAILED(hr = mGame->Direct3DDevice()->CreateInputLayout
(inputElementDescriptions,ARRAYSIZE(inputElementDescriptions),
passDesc.pIAInputSignature,
passDesc.IAInputSignatureSize, &mInputLayout)))
{
    throw GameException("ID3D11Device::CreateInputLayout() failed.",
hr);
}
```

---

```
typedef struct D3D11_BUFFER_DESC
{
    UINT ByteWidth;
    D3D11_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
    UINT StructureByteStride;
} D3D11_BUFFER_DESC;
```

```
typedef struct D3D11_SUBRESOURCE_DATA
{
    const void *pSysMem;
    UINT SysMemPitch;
    UINT SysMemSlicePitch;
} D3D11_SUBRESOURCE_DATA;
```

```
HRESULT CreateBuffer(
    const D3D11_BUFFER_DESC *pDesc,
    const D3D11_SUBRESOURCE_DATA *pInitialData,
    ID3D11Buffer **ppBuffer);
```

```
BasicEffectVertex vertices[] =
{
    BasicEffectVertex(XMFLOAT4(-1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Red))),
    BasicEffectVertex(XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Green))),
    BasicEffectVertex(XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Blue)))
};

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) *
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, &mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}
```

---

```
void IASetVertexBuffers(  
    UINT StartSlot,  
    UINT NumBuffers,  
    ID3D11Buffer *const *ppVertexBuffers,  
    const UINT *pStrides,  
    const UINT *pOffsets);
```

```
ID3D11DeviceContext* direct3DDeviceContext =  
mGame->Direct3DDeviceContext();  
direct3DDeviceContext->IASetPrimitiveTopology(  
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);  
direct3DDeviceContext->IASetInputLayout(&mInputLayout);  
  
UINT stride = sizeof(BasicEffectVertex);  
UINT offset = 0;  
direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,  
&stride, &offset);
```

---

```
XMMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
XMMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
mWvpVariable->SetMatrix(reinterpret_cast<const float*>(&wvp));

mPass->Apply(0, direct3DDeviceContext);
```

---

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

```
direct3DDeviceContext->Draw(3, 0);
```

---

```
#include "TriangleDemo.h"
#include "Game.h"
#include "GameException.h"
#include "MatrixHelper.h"
#include "ColorHelper.h"
#include "Camera.h"
#include "Utility.h"
#include "D3DCompiler.h"

namespace Rendering
{
    RTTI_DEFINITIONS(TriangleDemo)

    TriangleDemo::TriangleDemo(Game& game, Camera& camera)
        : DrawableGameComponent(game, camera),
          mEffect(nullptr), mTechnique(nullptr), mPass(nullptr),
          mWvpVariable(nullptr),
          mInputLayout(nullptr), mWorldMatrix(MatrixHelper::Identity),
          mVertexBuffer(nullptr)
    {
    }

    TriangleDemo::~TriangleDemo()
    {
        ReleaseObject(mWvpVariable);
        ReleaseObject(mPass);
        ReleaseObject(mTechnique);
        ReleaseObject(mEffect);
        ReleaseObject(mInputLayout);
    }
}
```

```
        ReleaseObject(mVertexBuffer);
    }

    void TriangleDemo::Initialize()
    {
        SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

        // Compile the shader
        UINT shaderFlags = 0;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3DCOMPILE_DEBUG;
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

        ID3D10Blob* compiledShader = nullptr;
        ID3D10Blob* errorMessages = nullptr;
        HRESULT hr = D3DCompileFromFile(L"Content\\Effects\\
BasicEffect.fx",nullptr, nullptr, nullptr, "fx_5_0", shaderFlags, 0,
&compiledShader,
&errorMessages);
        if (errorMessages != nullptr)
        {
            GameException ex((char*)errorMessages->GetBufferPointer(),
hr);
            ReleaseObject(errorMessages);

            throw ex;
        }

        if (FAILED(hr))
        {
            throw GameException("D3DX11CompileFromFile() failed.", hr);
        }

        // Create an effect object from the compiled shader
        hr = D3DX11CreateEffectFromMemory(compiledShader->
GetBufferPointer(),
compiledShader->GetBufferSize(), 0, mGame->Direct3DDevice(), &mEffect);
        if (FAILED(hr))
        {
            throw GameException("D3DX11CreateEffectFromMemory()
failed.", hr);
        }

        ReleaseObject(compiledShader);

        // Look up the technique, pass, and WVP variable from the
effect
```

```
mTechnique = mEffect->GetTechniqueByName("main11");
if (mTechnique == nullptr)
{
    throw GameException("ID3DX11Effect::GetTechniqueByName()
could not find the specified technique.", hr);
}

mPass = mTechnique->GetPassByName("p0");
if (mPass == nullptr)
{
    throw GameException("ID3DX11EffectTechnique::GetPassBy
Name() could not find the specified pass.", hr);
}

ID3DX11EffectVariable* variable = mEffect->GetVariableByName
("WorldViewProjection");

if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName()
could not find the specified variable.", hr);
}

mWvpVariable = variable->AsMatrix();
if (mWvpVariable->IsValid() == false)
{
    throw GameException("Invalid effect variable cast.");
}
```

```
// Create the input layout
D3DX11_PASS_DESC passDesc;
mPass->GetDesc(&passDesc);

D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR",     0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

if (FAILED(hr = mGame->Direct3DDevice()->
CreateInputLayout(inputElementDescriptions,
ARRAYSIZE(inputElementDescriptions), passDesc.pIAInputSignature,
passDesc.IAInputSignatureSize, &mInputLayout)))
{
    throw GameException("ID3D11Device::CreateInputLayout() failed.", hr);
}

// Create the vertex buffer
BasicEffectVertex vertices[] =
{
    BasicEffectVertex(XMFLOAT4(-1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Red))),
    BasicEffectVertex(XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Green))),
    BasicEffectVertex(XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Blue)))
};
```

```
D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) * 
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof
(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()->CreateBuffer
(&vertexBufferDesc, &vertexSubResourceData, &mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer()
failed.");
}
}

void TriangleDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(BasicEffectVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mWvpVariable->SetMatrix(reinterpret_cast<const float*>(&wvp));

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->Draw(3, 0);
}
```

```
void TriangleDemo::Update(const GameTime& gameTime)
{
    mAngle += XM_PI * static_cast<float>(gameTime.ElapsedGameTime());
    XMStoreFloat4x4(&mWorldMatrix, XMMatrixRotationZ(mAngle));
}
```

---

```
BasicEffectVertex vertices[] =
{
    BasicEffectVertex(XMFLOAT4(-1.0f, +1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Green))),
    BasicEffectVertex(XMFLOAT4(+1.0f, +1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Yellow))),
    BasicEffectVertex(XMFLOAT4(+1.0f, +1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::White))),
    BasicEffectVertex(XMFLOAT4(-1.0f, +1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::BlueGreen))),

    BasicEffectVertex(XMFLOAT4(-1.0f, -1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Blue))),
    BasicEffectVertex(XMFLOAT4(+1.0f, -1.0f, +1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Purple))),
    BasicEffectVertex(XMFLOAT4(+1.0f, -1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Red))),
    BasicEffectVertex(XMFLOAT4(-1.0f, -1.0f, -1.0f, 1.0f),
XMFLOAT4(reinterpret_cast<const float*>(&ColorHelper::Black)))
};

D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) *
ARRAYSIZE(vertices);
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
```

```
D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(mGame->Direct3DDevice()->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, &mVertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}

UINT indices[] =
{
    0, 1, 2,
    0, 2, 3,

    4, 5, 6,
    4, 6, 7,

    3, 2, 5,
    3, 5, 4,

    2, 1, 6,
    2, 6, 5,

    1, 7, 6,
    1, 0, 7,

    0, 3, 4,
    0, 4, 7
};

D3D11_BUFFER_DESC indexBufferDesc;
ZeroMemory(&indexBufferDesc, sizeof(indexBufferDesc));
indexBufferDesc.ByteWidth = sizeof(UINT) * 36;
indexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

D3D11_SUBRESOURCE_DATA indexSubResourceData;
ZeroMemory(&indexSubResourceData, sizeof(indexSubResourceData));
indexSubResourceData.pSysMem = indices;
if (FAILED(mGame->Direct3DDevice()->CreateBuffer(&indexBufferDesc,
&indexSubResourceData, &mIndexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}
```

```
void IASetIndexBuffer(  
    ID3D11Buffer *pIndexBuffer,  
    DXGI_FORMAT Format,  
    UINT Offset);
```

```
void DrawIndexed(
    UINT IndexCount,
    UINT StartIndexLocation,
    INT BaseVertexLocation);
```

```
void CubeDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(BasicEffectVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_FORMAT_
R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mWvpVariable->SetMatrix(reinterpret_cast<const float*>(&wvp));

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(36, 0, 0);
}
```

---

```
ID3D11DeviceContext::RSGetState()
ID3D11DeviceContext::OMGetBlendState()
ID3D11DeviceContext::OMGetDepthStencilState()

ID3D11DeviceContext::RSSetState()
ID3D11DeviceContext::OMSetBlendState()
ID3D11DeviceContext::OMSetDepthStencilState()
```

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;

    class RenderStateHelper
    {
public:
    RenderStateHelper(Game& game);
    ~RenderStateHelper();

    static void ResetAll(ID3D11DeviceContext* deviceContext);

    ID3D11RasterizerState* RasterizerState();
    ID3D11BlendState* BlendState();
    ID3D11DepthStencilState* DepthStencilState();

    void SaveRasterizerState();
    void RestoreRasterizerState() const;

    void SaveBlendState();
    void RestoreBlendState() const;

    void SaveDepthStencilState();
    void RestoreDepthStencilState() const;

    void SaveAll();
    void RestoreAll() const;

private:
    RenderStateHelper(const RenderStateHelper& rhs);
    RenderStateHelper& operator=(const RenderStateHelper& rhs);

    Game& mGame;

    ID3D11RasterizerState* mRasterizerState;
    ID3D11BlendState* mBlendState;
    FLOAT* mBlendFactor;
    UINT mSampleMask;
    ID3D11DepthStencilState* mDepthStencilState;
    UINT mStencilRef;
};

}
```

```
#include "RenderStateHelper.h"
#include "Game.h"

namespace Library
{
    RenderStateHelper::RenderStateHelper(Game& game)
        : mGame(game), mRasterizerState(nullptr), mBlendState(nullptr),
          mBlendFactor(new FLOAT[4]), mSampleMask(UINT_MAX),
          mDepthStencilState(nullptr), mStencilRef(UINT_MAX)
    {
    }

    RenderStateHelper::~RenderStateHelper()
    {
        ReleaseObject(mRasterizerState);
        ReleaseObject(mBlendState);
        ReleaseObject(mDepthStencilState);
        DeleteObjects(mBlendFactor);
    }

    void RenderStateHelper::ResetAll(ID3D11DeviceContext*
deviceContext)
    {
        deviceContext->RSSetState(nullptr);
        deviceContext->OMSetBlendState(nullptr, nullptr, UINT_MAX);
        deviceContext->OMSetDepthStencilState(nullptr, UINT_MAX);
    }

    void RenderStateHelper::SaveRasterizerState()
    {
        ReleaseObject(mRasterizerState);
```

```
mGame.Direct3DDeviceContext () ->RSGetState (&mRasterizerState) ;  
}  
  
void RenderStateHelper::RestoreRasterizerState() const  
{  
    mGame.Direct3DDeviceContext () ->RSSetState (mRasterizerState) ;  
}  
  
void RenderStateHelper::SaveBlendState()  
{  
    ReleaseObject (mBlendState) ;  
    mGame.Direct3DDeviceContext () ->OMGetBlendState (&mBlendState,  
mBlendFactor, &mSampleMask) ;  
}  
  
void RenderStateHelper::RestoreBlendState() const  
{  
    mGame.Direct3DDeviceContext () ->OMSetBlendState (mBlendState,  
mBlendFactor, mSampleMask) ;  
}  
  
void RenderStateHelper::SaveDepthStencilState()  
{  
    ReleaseObject (mDepthStencilState) ;  
    mGame.Direct3DDeviceContext () ->OMGetDepthStencilState  
(&mDepthStencilState, &mStencilRef) ;  
}  
  
void RenderStateHelper::RestoreDepthStencilState() const  
{  
    mGame.Direct3DDeviceContext () ->OMSetDepthStencilState  
(mDepthStencilState, mStencilRef) ;  
}  
  
void RenderStateHelper::SaveAll()  
{  
    SaveRasterizerState () ;  
    SaveBlendState () ;  
    SaveDepthStencilState () ;  
}  
  
void RenderStateHelper::RestoreAll() const  
{  
    RestoreRasterizerState () ;  
    RestoreBlendState () ;  
    RestoreDepthStencilState () ;  
}
```

```
mRenderStateHelper->SaveAll();  
mFpsComponent->Draw(gameTime);  
mRenderStateHelper->RestoreAll();
```

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;
    class Mesh;
    class ModelMaterial;

    class Model
    {
    public:
        Model(Game& game, const std::string& filename, bool flipUVs =
false);
        ~Model();

        Game& GetGame();
        bool HasMeshes() const;
        bool HasMaterials() const;

        const std::vector<Mesh*>& Meshes() const;
        const std::vector<ModelMaterial*>& Materials() const;

    private:
        Model(const Model& rhs);
        Model& operator=(const Model& rhs);

        Game& mGame;
        std::vector<Mesh*> mMeshes;
        std::vector<ModelMaterial*> mMaterials;
    };
}
```

---

```
#pragma once

#include "Common.h"

struct aiMesh;

namespace Library
{
    class Material;
    class ModelMaterial;

    class Mesh
    {
        friend class Model;

    public:
        Mesh(Model& model, ModelMaterial* material);
        ~Mesh();

        Model& GetModel();
        ModelMaterial* GetMaterial();
        const std::string& Name() const;

        const std::vector<XMFLOAT3>& Vertices() const;
        const std::vector<XMFLOAT3>& Normals() const;
    };
}
```

```
    const std::vector<XMFLOAT3>& Tangents() const;
    const std::vector<XMFLOAT3>& BiNormals() const;
    const std::vector<std::vector<XMFLOAT3>*>& TextureCoordinates()
const;
    const std::vector<std::vector<XMFLOAT4>*>& VertexColors()
const;
    UINT FaceCount() const;
    const std::vector<UINT>& Indices() const;
    void CreateIndexBuffer(ID3D11Buffer** indexBuffer);

private:
    Mesh(Model& model, aiMesh& mesh);
    Mesh(const Mesh& rhs);
    Mesh& operator=(const Mesh& rhs);

    Model& mModel;
    ModelMaterial* mMaterial;
    std::string mName;
    std::vector<XMFLOAT3> mVertices;
    std::vector<XMFLOAT3> mNormals;
    std::vector<XMFLOAT3> mTangents;
    std::vector<XMFLOAT3> mBiNormals;
    std::vector<std::vector<XMFLOAT3>*> mTextureCoordinates;
    std::vector<std::vector<XMFLOAT4>*> mVertexColors;
    UINT mFaceCount;
    std::vector<UINT> mIndices;
};

}
```

---

```
#pragma once

#include "Common.h"

struct aiMaterial;

namespace Library
{
    enum TextureType
    {
        TextureTypeDiffuse = 0,
        TextureTypeSpecularMap,
        TextureTypeAmbient,
        TextureTypeEmissive,
        TextureTypeHeightmap,
        TextureTypeNormalMap,
        TextureTypeSpecularPowerMap,
        TextureTypeDisplacementMap,
        TextureTypeLightMap,
        TextureTypeEnd
    };
}

class ModelMaterial
{
```

```
friend class Model;

public:
    ModelMaterial(Model& model);
    ~ModelMaterial();

    Model& GetModel();
    const std::string& Name() const;
    const std::map<TextureType, std::vector<std::wstring>*>
Textures() const;

private:
    static void InitializeTextureTypeMappings();
    static std::map<TextureType, UINT> sTextureTypeMappings;

    ModelMaterial(Model& model, aiMaterial* material);
    ModelMaterial(const ModelMaterial& rhs);
    ModelMaterial& operator=(const ModelMaterial& rhs);

    Model& mModel;
    std::string mName;
    std::map<TextureType, std::vector<std::wstring>*> mTextures;
};

}
```

---

```
#include "Model.h"
#include "Game.h"
#include "GameException.h"
#include "Mesh.h"
#include "ModelMaterial.h"
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

namespace Library
{
    Model::Model(Game& game, const std::string& filename, bool flipUVs)
        : mGame(game), mMeshes(), mMaterials()
    {
        Assimp::Importer importer;

        UINT flags = aiProcess_Triangulate | aiProcess_
JoinIdenticalVertices | aiProcess_SortByPType | aiProcess_FlipWindingOrder;
        if (flipUVs)
        {
            flags |= aiProcess_FlipUVs;
        }
        const aiScene* scene = importer.ReadFile(filename, flags);
        if (scene == nullptr)
        {
            throw GameException(importer.GetErrorString());
        }

        if (scene->HasMaterials())
        {
```

```
    for (UINT i = 0; i < scene->mNumMaterials; i++)
    {
        mMaterials.push_back(new ModelMaterial(*this,
scene->mMaterials[i]));
    }
}

if (scene->HasMeshes())
{
    for (UINT i = 0; i < scene->mNumMeshes; i++)
    {
        Mesh* mesh = new Mesh(*this, *(scene->mMeshes[i]));
        mMeshes.push_back(mesh);
    }
}
}

Model::~Model()
{
    for (Mesh* mesh : mMeshes)
    {
        delete mesh;
    }

    for (ModelMaterial* material : mMaterials)
    {
```

```
        delete material;
    }

}

Game& Model::GetGame()
{
    return mGame;
}

bool Model::HasMeshes() const
{
    return (mMeshes.size() > 0);
}

bool Model::HasMaterials() const
{
    return (mMaterials.size() > 0);
}

const std::vector<Mesh*>& Model::Meshes() const
{
    return mMeshes;
}

const std::vector<ModelMaterial*>& Model::Materials() const
{
    return mMaterials;
}
```

---

```
#include "Mesh.h"
#include "Model.h"
#include "Game.h"
#include "GameException.h"
#include <assimp/scene.h>

namespace Library
{
    Mesh::Mesh(Model& model, aiMesh& mesh)
        : mModel(model), mMaterial(nullptr), mName(mesh.mName.C_Str()),
          mVertices(), mNormals(), mTangents(), mBiNormals(),
          mTextureCoordinates(), mVertexColors(), mFaceCount(0),
          mIndices()
    {
        mMaterial = mModel.Materials().at(mesh.mMaterialIndex);

        // Vertices
        mVertices.reserve(mesh.mNumVertices);
        for (UINT i = 0; i < mesh.mNumVertices; i++)
        {
            mVertices.push_back(XMFLOAT3(reinterpret_cast<const
float*>(&mesh.mVertices[i])));
        }

        // Normals
        if (mesh.HasNormals())
        {
```

```
mNormals.reserve(mesh.mNumVertices);
for (UINT i = 0; i < mesh.mNumVertices; i++)
{
    mNormals.push_back(XMFLOAT3(reinterpret_cast<const
float*>(&mesh.mNormals[i])));
}

// Tangents and Binormals
if (mesh.HasTangentsAndBitangents())
{
    mTangents.reserve(mesh.mNumVertices);
    mBiNormals.reserve(mesh.mNumVertices);
    for (UINT i = 0; i < mesh.mNumVertices; i++)
    {
        mTangents.push_back(XMFLOAT3(reinterpret_cast<const
float*>(&mesh.mTangents[i])));
        mBiNormals.push_back(XMFLOAT3(reinterpret_cast<const
float*>(&mesh.mBitangents[i])));
    }
}

// Texture Coordinates
UINT uvChannelCount = mesh.GetNumUVChannels();
for (UINT i = 0; i < uvChannelCount; i++)
{
```

```
    std::vector<XMFLOAT3>* textureCoordinates = new
std::vector<XMFLOAT3>();
    textureCoordinates->reserve(mesh.mNumVertices);
    mTextureCoordinates.push_back(textureCoordinates);

    aiVector3D* aiTextureCoordinates = mesh.mTextureCoords[i];
    for (UINT j = 0; j < mesh.mNumVertices; j++)
    {
        textureCoordinates->push_back(XMFLOAT3(reinterpret_
cast<const float*>(&aiTextureCoordinates[j])));
    }
}

// Vertex Colors
UINT colorChannelCount = mesh.GetNumColorChannels();
for (UINT i = 0; i < colorChannelCount; i++)
{
    std::vector<XMFLOAT4>* vertexColors = new
std::vector<XMFLOAT4>();
    vertexColors->reserve(mesh.mNumVertices);
    mVertexColors.push_back(vertexColors);

    aiColor4D* aiVertexColors = mesh.mColors[i];
    for (UINT j = 0; j < mesh.mNumVertices; j++)
    {
        vertexColors->push_back(XMFLOAT4(reinterpret_cast
<const float*>(&aiVertexColors[j])));
    }
}
```

```
}

// Faces (note: could pre-reserve if we limit primitive types)
if (mesh.HasFaces())
{
    mFaceCount = mesh.mNumFaces;
    for (UINT i = 0; i < mFaceCount; i++)
    {
        aiFace* face = &mesh.mFaces[i];

        for (UINT j = 0; j < face->mNumIndices; j++)
        {
            mIndices.push_back(face->mIndices[j]);
        }
    }
}

Mesh::~Mesh()
{
    for (std::vector<XMFLOAT3>* textureCoordinates :
mTextureCoordinates)
    {
        delete textureCoordinates;
    }

    for (std::vector<XMFLOAT4>* vertexColors : mVertexColors)
    {
        delete vertexColors;
    }
}
```

```
    }

void Mesh::CreateIndexBuffer(ID3D11Buffer** indexBuffer)
{
    assert(indexBuffer != nullptr);

    D3D11_BUFFER_DESC indexBufferDesc;
    ZeroMemory(&indexBufferDesc, sizeof(indexBufferDesc));
    indexBufferDesc.ByteWidth = sizeof(UINT) * mIndices.size();
    indexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
    indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

    D3D11_SUBRESOURCE_DATA indexSubResourceData;
    ZeroMemory(&indexSubResourceData, sizeof
(indexSubResourceData));
    indexSubResourceData.pSysMem = &mIndices[0];
    if (FAILED(mModel.GetGame().Direct3DDevice()->
CreateBuffer(&indexBufferDesc, &indexSubResourceData, indexBuffer)))
    {
        throw GameException("ID3D11Device::CreateBuffer()
failed.");
    }
}
}
```

---

```
#include "ModelMaterial.h"
#include "GameException.h"
#include "Utility.h"
#include <assimp/scene.h>

namespace Library
{
    std::map<TextureType, UINT> ModelMaterial::sTextureTypeMappings;

    ModelMaterial::ModelMaterial(Model& model)
        : mModel(model), mTextures()
    {
        InitializeTextureTypeMappings();
    }

    ModelMaterial::ModelMaterial(Model& model, aiMaterial* material)
        : mModel(model), mTextures()
    {
        InitializeTextureTypeMappings();

        aiString name;
        material->Get(AI_MATKEY_NAME, name);
        mName = name.C_Str();

        for (TextureType textureType = (TextureType)0; textureType < TextureTypeEnd; textureType = (TextureType)(textureType + 1))
        {
            aiTextureType mappedTextureType =
(aiTextureType)sTextureTypeMappings[textureType];

            UINT textureCount = material->GetTextureCount
(mappedTextureType);
```

```
    if (textureCount > 0)
    {
        std::vector<std::wstring>* textures = new std::vector<std::wstring>();
        mTextures.insert(std::pair<TextureType, std::vector<std::wstring>>(textureType, textures));

        textures->reserve(textureCount);
        for (UINT textureIndex = 0; textureIndex < textureCount; textureIndex++)
        {
            aiString path;
            if (material->GetTexture(mappedTextureType, textureIndex, &path) == AI_SUCCESS)
            {
                std::wstring wPath;
                Utility::ToWideString(path.C_Str(), wPath);

                textures->push_back(wPath);
            }
        }
    }

ModelMaterial::~ModelMaterial()
{
```

```
        for (std::pair<TextureType, std::vector<std::wstring>*>
textures : mTextures)
{
    DeleteObject(textures.second);
}
}

Model& ModelMaterial::GetModel()
{
    return mModel;
}

const std::string& ModelMaterial::Name() const
{
    return mName;
}

const std::map<TextureType, std::vector<std::wstring>*>
ModelMaterial::Textures() const
{
    return mTextures;
}

void ModelMaterial::InitializeTextureTypeMappings()
{
    if (sTextureTypeMappings.size() != TextureTypeEnd)
    {
        sTextureTypeMappings[TextureTypeDiffuse] =
aiTextureType_DIFFUSE;
        sTextureTypeMappings[TextureTypeSpecularMap] =
aiTextureType_SPECULAR;
        sTextureTypeMappings[TextureTypeAmbient] =
aiTextureType_AMBIENT;
        sTextureTypeMappings[TextureTypeHeightmap] =
aiTextureType_HEIGHT;
        sTextureTypeMappings[TextureTypeNormalMap] =
aiTextureType_NORMALS;
        sTextureTypeMappings[TextureTypeSpecularPowerMap] =
aiTextureType_SHININESS;
        sTextureTypeMappings[TextureTypeDisplacementMap] =
aiTextureType_DISPLACEMENT;
        sTextureTypeMappings[TextureTypeLightMap] =
aiTextureType_LIGHTMAP;
    }
}
}
```

```
void CreateVertexBuffer(ID3D11Device* device, const Mesh& mesh,  
ID3D11Buffer** vertexBuffer) const;
```

```
void ModelDemo::Initialize()
{
    // ... Previously presented code omitted for brevity ...

    // Load the model
    std::unique_ptr<Model> model(new Model(*mGame,
"Content\\Models\\Sphere.obj", true));

    // Create the vertex and index buffers
    Mesh* mesh = model->Meshes().at(0);
    CreateVertexBuffer(mGame->Direct3DDevice(), *mesh, &mVertexBuffer);
    mesh->CreateIndexBuffer(&mIndexBuffer);
    mIndexCount = mesh->Indices().size();
}

void ModelDemo::CreateVertexBuffer(ID3D11Device* device, const Mesh&
mesh, ID3D11Buffer** vertexBuffer) const
{
    const std::vector<XMFLOAT3>& sourceVertices = mesh.Vertices();

    std::vector<BasicEffectVertex> vertices;
    vertices.reserve(sourceVertices.size());
    if (mesh.VertexColors().size() > 0)
    {
        std::vector<XMFLOAT4*> vertexColors = mesh.VertexColors().
at(0);
        assert(vertexColors->size() == sourceVertices.size());

        for (UINT i = 0; i < sourceVertices.size(); i++)
        {
            XMFLOAT3 position = sourceVertices.at(i);
            XMFLOAT4 color = vertexColors->at(i);
            vertices.push_back(BasicEffectVertex(XMFLOAT4(position.x,
position.y, position.z, 1.0f), color));
        }
    }
    else
    {
        for (UINT i = 0; i < sourceVertices.size(); i++)
        {
            XMFLOAT3 position = sourceVertices.at(i);
            XMFLOAT4 color = ColorHelper::RandomColor();
            vertices.push_back(BasicEffectVertex(XMFLOAT4(position.x,
position.y, position.z, 1.0f), color));
        }
    }
}
```

```
D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(BasicEffectVertex) * vertices.
size();
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = &vertices[0];
if (FAILED(device->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, vertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}
```

---

```
ID3D11ShaderResourceView* mTextureShaderResourceView;  
ID3DX11EffectShaderResourceVariable* mColorTextureVariable;
```

```
typedef struct _TextureMappingVertex
{
    XMFLOAT4 Position;
    XMFLOAT2 TextureCoordinates;

    _TextureMappingVertex() { }

    _TextureMappingVertex(XMFLOAT4 position, XMFLOAT2
textureCoordinates)
        : Position(position), TextureCoordinates(textureCoordinates) {
    }
} TextureMappingVertex;
```

```
void TexturedModelDemo::Initialize()
{
    SetCurrentDirectory( Utility::ExecutableDirectory().c_str() );

    // Compile the shader
    UINT shaderFlags = 0;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3DCOMPILE_DEBUG;
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

    ID3D10Blob* compiledShader = nullptr;
    ID3D10Blob* errorMessages = nullptr;
    HRESULT hr = D3DCompileFromFile(L"Content\\Effects\\TextureMapping.
fx", nullptr, nullptr, nullptr, "fx_5_0", shaderFlags, 0,
&compiledShader, &errorMessages);
    if (errorMessages != nullptr)
    {
        GameException ex((char*)errorMessages->GetBufferPointer(), hr);
        ReleaseObject(errorMessages);

        throw ex;
    }
    if (FAILED(hr))
    {
        throw GameException("D3DX11CompileFromFile() failed.", hr);
    }
}
```

```
// Create an effect object from the compiled shader
hr = D3DX11CreateEffectFromMemory(compiledShader->
GetBufferPointer(), compiledShader->GetBufferSize(), 0, mGame->
Direct3DDevice(), &mEffect);
if (FAILED(hr))
{
    throw GameException("D3DX11CreateEffectFromMemory() failed.", hr);
}

ReleaseObject(compiledShader);

// Look up the technique, pass, and WVP variable from the effect
mTechnique = mEffect->GetTechniqueByName("main11");
if (mTechnique == nullptr)
{
    throw GameException("ID3DX11Effect::GetTechniqueByName() could not find the specified technique.", hr);
}

mPass = mTechnique->GetPassByName("p0");
if (mPass == nullptr)
{
    throw GameException("ID3DX11EffectTechnique::GetPassByName() could not find the specified pass.", hr);
}
```

```
ID3DX11EffectVariable* variable = mEffect->GetVariableByName
("WorldViewProjection");
if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName() could
not find the specified variable.", hr);
}

mWvpVariable = variable->AsMatrix();
if (mWvpVariable->IsValid() == false)
{
    throw GameException("Invalid effect variable cast.");
}

variable = mEffect->GetVariableByName("ColorTexture");
if (variable == nullptr)
{
    throw GameException("ID3DX11Effect::GetVariableByName() could
not find the specified variable.", hr);
}

mColorTextureVariable = variable->AsShaderResource();
if (mColorTextureVariable->IsValid() == false)
{
```

```
        throw GameException("Invalid effect variable cast.");
    }

    // Create the input layout
    D3DX11_PASS_DESC passDesc;
    mPass->GetDesc(&passDesc);

    D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
    };

    if (FAILED(hr = mGame->Direct3DDevice()->CreateInputLayout
(inputElementDescriptions, ARRSIZE(inputElementDescriptions),
passDesc.pIAInputSignature, passDesc.IAInputSignatureSize,
&mInputLayout)))
    {
        throw GameException("ID3D11Device::CreateInputLayout()
failed.", hr);
    }

    // Load the model
    std::unique_ptr<Model> model(new Model(*mGame, "Content\\Models\\
Sphere.obj", true));
}
```

```
// Create the vertex and index buffers
Mesh* mesh = model->Meshes().at(0);
CreateVertexBuffer(mGame->Direct3DDevice(), *mesh, &mVertexBuffer);
mesh->CreateIndexBuffer(&mIndexBuffer);
mIndexCount = mesh->Indices().size();

// Load the texture
std::wstring textureName = L"Content\\Textures\\EarthComposite.
jpg";
if (FAILED(hr = DirectX::CreateWICTextureFromFile(mGame->
Direct3DDevice(), mGame->Direct3DDeviceContext(), textureName.c_str(),
nullptr, &mTextureShaderResourceView)))
{
    throw GameException("CreateWICTextureFromFile() failed.", hr);
}
}

void TexturedModelDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(TextureMappingVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_FORMAT_
R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mWvpVariable->SetMatrix(reinterpret_cast<const float*>(&wvp));
    mColorTextureVariable->SetResource(mTextureShaderResourceView);

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}

void TexturedModelDemo::CreateVertexBuffer(ID3D11Device* device, const
Mesh& mesh, ID3D11Buffer** vertexBuffer) const
{
```

```
const std::vector<XMFLOAT3>& sourceVertices = mesh.Vertices();

std::vector<TextureMappingVertex> vertices;
vertices.reserve(sourceVertices.size());

std::vector<XMFLOAT3>* textureCoordinates = mesh.
TextureCoordinates().at(0);
assert(textureCoordinates->size() == sourceVertices.size());

for (UINT i = 0; i < sourceVertices.size(); i++)
{
    XMFLOAT3 position = sourceVertices.at(i);
    XMFLOAT3 uv = textureCoordinates->at(i);
    vertices.push_back(TextureMappingVertex(XMFLOAT4(position.x,
position.y, position.z, 1.0f), XMFLOAT2(uv.x, uv.y)));
}

D3D11_BUFFER_DESC vertexBufferDesc;

ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = sizeof(TextureMappingVertex) *
vertices.size();
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof(vertexSubResourceData));
vertexSubResourceData.pSysMem = &vertices[0];
if (FAILED(device->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, vertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer() failed.");
}
}
```

---

```
#pragma once

#include "Common.h"
#include "Technique.h"
#include "Variable.h"

namespace Library
{
    class Game;

    class Effect
    {
    public:
        Effect(Game& game);
        virtual ~Effect();

        static void CompileEffectFromFile(ID3D11Device* direct3DDevice,
ID3DX11Effect** effect, const std::wstring& filename);
        static void LoadCompiledEffect(ID3D11Device* direct3DDevice,
ID3DX11Effect** effect, const std::wstring& filename);

        Game& GetGame();
        ID3DX11Effect* GetEffect() const;
        void SetEffect(ID3DX11Effect* effect);
        const D3DX11_EFFECT_DESC& EffectDesc() const;
        const std::vector<Technique*>& Techniques() const;
        const std::map<std::string, Technique*>& TechniquesByName()
const;
        const std::vector<Variable*>& Variables() const;
        const std::map<std::string, Variable*>& VariablesByName()
const;

        void CompileFromFile(const std::wstring& filename);
        void LoadCompiledEffect(const std::wstring& filename);

    private:
        Effect(const Effect& rhs);
        Effect& operator=(const Effect& rhs);

        void Initialize();
    };
}
```

```
Game& mGame;
ID3DX11Effect* mEffect;
D3DX11_EFFECT_DESC mEffectDesc;
std::vector<Technique*> mTechniques;
std::map<std::string, Technique*> mTechniquesByName;
std::vector<Variable*> mVariables;
std::map<std::string, Variable*> mVariablesByName;
};

}
```

---

```
void Effect::CompileEffectFromFile(ID3D11Device* direct3DDevice,
ID3DX11Effect** effect, const std::wstring& filename)
{
    UINT shaderFlags = 0;

#if defined( DEBUG ) || defined( _DEBUG )
    shaderFlags |= D3DCOMPILE_DEBUG;
    shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

    ID3D10Blob* compiledShader = nullptr;
    ID3D10Blob* errorMessages = nullptr;
    HRESULT hr = D3DCompileFromFile(filename.c_str(), nullptr, nullptr,
nullptr, "fx_5_0", shaderFlags, 0, &compiledShader, &errorMessages);
    if (errorMessages != nullptr)
    {
        GameException ex((char*)errorMessages->GetBufferPointer(), hr);
        ReleaseObject(errorMessages);

        throw ex;
    }

    if (FAILED(hr))
    {
        throw GameException("D3DX11CompileFromFile() failed.", hr);
    }

    hr = D3DX11CreateEffectFromMemory(compiledShader-
>GetBufferPointer(), compiledShader->GetBufferSize(), NULL,
direct3DDevice, effect);
    if (FAILED(hr))
    {
        throw GameException("D3DX11CreateEffectFromMemory() failed.",
hr);
    }

    ReleaseObject(compiledShader);
}

void Effect::LoadCompiledEffect(ID3D11Device* direct3DDevice,
ID3DX11Effect** effect, const std::wstring& filename)
{
    std::vector<char> compiledShader;
    Utility::LoadBinaryFile(filename, compiledShader);
```

```
HRESULT hr = D3DX11CreateEffectFromMemory(&compiledShader.front(),
compiledShader.size(), NULL, direct3DDevice, effect);
if (FAILED(hr))
{
    throw GameException("D3DX11CreateEffectFromMemory() failed.",
hr);
}
}

void Effect::Initialize()
{
    HRESULT hr = mEffect->GetDesc(&mEffectDesc);
    if (FAILED(hr))
    {
        throw GameException("ID3DX11Effect::GetDesc() failed.", hr);
    }

    for (UINT i = 0; i < mEffectDesc.Techniques; i++)
    {
        Technique* technique = new Technique(mGame, *this,
mEffect->GetTechniqueByIndex(i));
        mTechniques.push_back(technique);
        mTechniquesByName.insert(std::pair<std::string,
Technique*>(technique->Name(), technique));
    }

    for (UINT i = 0; i < mEffectDesc.GlobalVariables; i++)
    {
        Variable* variable = new Variable(*this,
mEffect->GetVariableByIndex(i));
        mVariables.push_back(variable);
        mVariablesByName.insert(std::pair<std::string,
Variable*>(variable->Name(), variable));
    }
}
```

---

`$ (OutDir)Content\Effects\%(Filename).cso`

```
#pragma once

#include "Common.h"
#include "Pass.h"

namespace Library
{
    class Game;
    class Effect;

    class Technique
    {
    public:
        Technique(Game& game, Effect& effect, ID3DX11EffectTechnique* technique);
        ~Technique();

        Effect& GetEffect();
        ID3DX11EffectTechnique* GetTechnique() const;
        const D3DX11_TECHNIQUE_DESC& TechniqueDesc() const;
        const std::string& Name() const;
        const std::vector<Pass*>& Passes() const;
        const std::map<std::string, Pass*>& PassesByName() const;

    private:
        Technique(const Technique& rhs);
        Technique& operator=(const Technique& rhs);

        Effect& mEffect;
        ID3DX11EffectTechnique* mTechnique;
        D3DX11_TECHNIQUE_DESC mTechniqueDesc;
        std::string mName;
        std::vector<Pass*> mPasses;
        std::map<std::string, Pass*> mPassesByName;
    };
}
```

---

```
Technique::Technique(Game& game, Effect& effect,
ID3DX11EffectTechnique* technique)
: mEffect(effect), mTechnique(technique), mTechniqueDesc(),
mName(), mPasses(), mPassesByName()
{
    mTechnique->GetDesc(&mTechniqueDesc);
    mName = mTechniqueDesc.Name;

    for (UINT i = 0; i < mTechniqueDesc.Passes; i++)
    {
        Pass* pass = new Pass(game, *this,
mTechnique->GetPassByIndex(i));
        mPasses.push_back(pass);
        mPassesByName.insert(std::pair<std::string, Pass*>(pass-
>Name(), pass));
    }
}
```

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Game;
    class Technique;

    class Pass
    {
        public:
            Pass(Game& game, Technique& technique, ID3DX11EffectPass* pass);

            Technique& GetTechnique();
            ID3DX11EffectPass* GetPass() const;
            const D3DX11_PASS_DESC& PassDesc() const;
            const std::string& Name() const;

            void CreateInputLayout(const D3D11_INPUT_ELEMENT_DESC* inputElementDesc, UINT numElements, ID3D11InputLayout **inputLayout);
            void Apply(UINT flags, ID3D11DeviceContext* context);

        private:
            Pass(const Pass& rhs);
            Pass& operator=(const Pass& rhs);

            Game& mGame;
            Technique& mTechnique;
            ID3DX11EffectPass* mPass;
            D3DX11_PASS_DESC mPassDesc;
            std::string mName;
    };
}
```

```
Pass::Pass(Game& game, Technique& technique, ID3DX11EffectPass* pass)
    : mGame(game), mTechnique(technique), mPass(pass), mPassDesc(),
      mName()
{
    mPass->GetDesc(&mPassDesc);
    mName = mPassDesc.Name;
}

void Pass::CreateInputLayout(const D3D11_INPUT_ELEMENT_DESC*
inputElementDesc, UINT numElements, ID3D11InputLayout **inputLayout)
{
    HRESULT hr = mGame.Direct3DDevice()->CreateInputLayout(input
ElementDesc, numElements, mPassDesc.pIAInputSignature, mPassDesc.
IAInputSignatureSize, inputLayout);
    if (FAILED(hr))
    {
        throw GameException("ID3D11Device::CreateInputLayout() failed.", hr);
    }
}
```

---

```
#pragma once

#include "Common.h"

namespace Library
{
    class Effect;

    class Variable
    {
public:
    Variable(Effect& effect, ID3DX11EffectVariable* variable);

    Effect& GetEffect();
    ID3DX11EffectVariable* GetVariable() const;
    const D3DX11_EFFECT_VARIABLE_DESC& VariableDesc() const;
    ID3DX11EffectType* Type() const;
    const D3DX11_EFFECT_TYPE_DESC& TypeDesc() const;
    const std::string& Name() const;

    Variable& operator<<(CXMMATRIX value);
    Variable& operator<<(ID3D11ShaderResourceView* value);
    Variable& operator<<(FXMVECTOR value);
    Variable& operator<<(float value);

private:
    Variable(const Variable& rhs);
    Variable& operator=(const Variable& rhs);

    Effect& mEffect;
    ID3DX11EffectVariable* mVariable;
    D3DX11_EFFECT_VARIABLE_DESC mVariableDesc;
    ID3DX11EffectType* mType;
    D3DX11_EFFECT_TYPE_DESC mTypeDesc;
    std::string mName;
};

}
```

```
XMMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
XMMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
mBasicMaterial->WorldViewProjection() << wvp;
```

```
Variable& Variable::operator<<(CXMMATRIX value)
{
    ID3DX11EffectMatrixVariable* variable = mVariable->AsMatrix();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetMatrix(reinterpret_cast<const float*>(&value));

    return *this;
}

Variable& Variable::operator<<(ID3D11ShaderResourceView* value)
{
    ID3DX11EffectShaderResourceVariable* variable =
mVariable->AsShaderResource();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetResource(value);

    return *this;
}

Variable& Variable::operator<<(FXMVECTOR value)
{
    ID3DX11EffectVectorVariable* variable = mVariable->AsVector();
    if (variable->IsValid() == false)
    {
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetFloatVector(reinterpret_cast<const float*>(&value));

    return *this;
}

Variable& Variable::operator<<(float value)
{
    ID3DX11EffectScalarVariable* variable = mVariable->AsScalar();
    if (variable->IsValid() == false)
    {
```

```
        throw GameException("Invalid effect variable cast.");
    }

    variable->SetFloat(value);

    return *this;
}
```

---

```
#pragma once

#include "Common.h"
#include "Effect.h"

namespace Library
{
    class Model;
    class Mesh;

    class Material : public RTTI
    {
        RTTI_DECLARATIONS(Material, RTTI)

    public:
        Material();
        Material(const std::string& defaultTechniqueName);
        virtual ~Material();

        Variable* operator[](const std::string& variableName);
        Effect* GetEffect() const;
        Technique* CurrentTechnique() const;
        void SetCurrentTechnique(Technique* currentTechnique);
        const std::map<Pass*, ID3D11InputLayout*>& InputLayouts()
        const;

        virtual void Initialize(Effect* effect);
        virtual void CreateVertexBuffer(ID3D11Device* device, const
Model& model, std::vector<ID3D11Buffer*>& vertexBuffers) const;
        virtual void CreateVertexBuffer(ID3D11Device* device, const
Mesh& mesh, ID3D11Buffer** vertexBuffer) const = 0;
        virtual UINT VertexSize() const = 0;

    protected:
        Material(const Material& rhs);
        Material& operator=(const Material& rhs);

        virtual void CreateInputLayout(const std::string&
techniqueName, const std::string& passName, D3D11_INPUT_ELEMENT_DESC*
inputElementDescriptions, UINT inputElementDescriptionCount);

        Effect* mEffect;
        Technique* mCurrentTechnique;
        std::string mDefaultTechniqueName;
        std::map<Pass*, ID3D11InputLayout*> mInputLayouts;
    };
}
```

```
#define MATERIAL_VARIABLE_DECLARATION(VariableName) \
    public: \
        Variable& VariableName() const; \
    private: \
        Variable* m ## VariableName; \
 \
#define MATERIAL_VARIABLE_DEFINITION(Material, VariableName) \
    Variable& Material::VariableName() const \
{ \
    return *m ## VariableName; \
} \
 \
#define MATERIAL_VARIABLE_INITIALIZATION(VariableName) m ## \
VariableName(NULL) \
 \
#define MATERIAL_VARIABLE_RETRIEVE(VariableName) \
    m ## VariableName = mEffect->VariablesByName(). \
at(#VariableName); \
}
```

---

```
Pass* pass = mBasicMaterial->CurrentTechnique()->Passes().at(0);
ID3D11InputLayout* inputLayout = mBasicMaterial->InputLayouts().
at(pass);
direct3DDeviceContext->IASetInputLayout(inputLayout);
```

MATERIAL\_VARIABLE\_DECLARATION  
MATERIAL\_VARIABLE\_DEFINITION  
MATERIAL\_VARIABLE\_INITIALIZATION  
MATERIAL\_VARIABLE\_RETRIEVE

```
#include "Material.h"
#include "GameException.h"
#include "Model.h"

namespace Library
{
    RTTI_DEFINITIONS(Material)

    Material::Material()
        : mEffect(nullptr), mCurrentTechnique(nullptr),
          mDefaultTechniqueName(), mInputLayouts()
    {
    }

    Material::Material(const std::string& defaultTechniqueName)
        : mEffect(nullptr), mCurrentTechnique(nullptr),
          mDefaultTechniqueName(defaultTechniqueName), mInputLayouts()
    {
    }

    Material::~Material()
    {
        for (std::pair<Pass*, ID3D11InputLayout*> inputLayout :
            mInputLayouts)
        {
            ReleaseObject(inputLayout.second);
        }
    }

    Variable* Material::operator[](const std::string& variableName)
    {
        const std::map<std::string, Variable*>& variables =
mEffect->VariablesByName();
        Variable* foundVariable = nullptr;

        std::map<std::string, Variable*>::const_iterator found =
variables.find(variableName);
        if (found != variables.end())
        {
            foundVariable = found->second;
        }

        return foundVariable;
    }

    Effect* Material::GetEffect() const
```

```
{  
    return mEffect;  
}  
  
Technique* Material::CurrentTechnique() const  
{  
    return mCurrentTechnique;  
}  
  
void Material::SetCurrentTechnique(Technique* currentTechnique)  
{  
    mCurrentTechnique = currentTechnique;  
}  
  
const std::map<Pass*, ID3D11InputLayout*>& Material::InputLayouts()  
const  
{  
    return mInputLayouts;  
}  
  
void Material::Initialize(Effect* effect)  
{  
    mEffect = effect;  
    assert(mEffect != nullptr);  
  
    Technique* defaultTechnique = nullptr;  
    assert(mEffect->Techniques().size() > 0);  
    if (mDefaultTechniqueName.empty() == false)  
    {  
        defaultTechnique = mEffect->TechniquesByName().  
at(mDefaultTechniqueName);  
        assert(defaultTechnique != nullptr);  
    }  
    else  
    {  
        defaultTechnique = mEffect->Techniques().at(0);  
    }  
  
    SetCurrentTechnique(defaultTechnique);  
}  
  
void Material::CreateVertexBuffer(ID3D11Device* device, const  
Model& model, std::vector<ID3D11Buffer*>& vertexBuffers) const  
{  
    vertexBuffers.reserve(model.Meshes().size());
```

```
for (Mesh* mesh : model.Meshes())
{
    ID3D11Buffer* vertexBuffer;
    CreateVertexBuffer(device, *mesh, &vertexBuffer);
    vertexBuffers.push_back(vertexBuffer);
}
}

void Material::CreateInputLayout(const std::string&
techniqueName, const std::string& passName, D3D11_INPUT_ELEMENT_DESC*
inputElementDescriptions, UINT inputElementDescriptionCount)
{
    Technique* technique = mEffect->TechniquesByName().
at(techniqueName);
    assert(technique != nullptr);

    Pass* pass = technique->PassesByName().at(passName);
    assert(pass != nullptr);

    ID3D11InputLayout* inputLayout;
    pass->CreateInputLayout(inputElementDescriptions,
inputElementDescriptionCount, &inputLayout);

    mInputLayouts.insert(std::pair<Pass*, ID3D11InputLayout*>(pass,
inputLayout));
}
}
```

---

```
#pragma once

#include "Common.h"
#include "Material.h"

namespace Library
{
    typedef struct _BasicMaterialVertex
    {
        XMFLOAT4 Position;
        XMFLOAT4 Color;

        _BasicMaterialVertex() { }

        _BasicMaterialVertex(const XMFLOAT4& position, const XMFLOAT4&
color)
            : Position(position), Color(color) { }
    } BasicMaterialVertex;

    class BasicMaterial : public Material
    {
        RTTI_DECLARATIONS(BasicMaterial, Material)

        MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)

        public:
            BasicMaterial();

            virtual void Initialize(Effect* effect) override;
            virtual void CreateVertexBuffer(ID3D11Device* device, const
Mesh& mesh, ID3D11Buffer** vertexBuffer) const override;
            void CreateVertexBuffer(ID3D11Device* device,
BasicMaterialVertex* vertices, UINT vertexCount, ID3D11Buffer**
vertexBuffer) const;
            virtual UINT VertexSize() const override;
    };
}
```

```
#include "BasicMaterial.h"
#include "GameException.h"
#include "Mesh.h"
#include "ColorHelper.h"

namespace Library
{
    RTTI_DEFINITIONS(BasicMaterial)

    BasicMaterial::BasicMaterial()
        : Material("main11"),
          MATERIAL_VARIABLE_INITIALIZATION(WorldViewProjection)
    {
    }

    MATERIAL_VARIABLE_DEFINITION(BasicMaterial, WorldViewProjection)

    void BasicMaterial::Initialize(Effect* effect)
    {
        Material::Initialize(effect);

        MATERIAL_VARIABLE_RETRIEVE(WorldViewProjection)

        D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
        {
            { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
            { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_
APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
        };

        CreateInputLayout("main11", "p0", inputElementDescriptions, ARR
AYSIZE(inputElementDescriptions));
    }

    void BasicMaterial::CreateVertexBuffer(ID3D11Device* device, const
Mesh& mesh, ID3D11Buffer** vertexBuffer) const
    {
        const std::vector<XMFLOAT3>& sourceVertices = mesh.Vertices();
        std::vector<BasicMaterialVertex> vertices;
        vertices.reserve(sourceVertices.size());
        if (mesh.VertexColors().size() > 0)
        {
            std::vector<XMFLOAT4>* vertexColors = mesh.VertexColors().
at(0);

```

```
assert(vertexColors->size() == sourceVertices.size());  
  
    for (UINT i = 0; i < sourceVertices.size(); i++)  
    {  
        XMFLOAT3 position = sourceVertices.at(i);  
        XMFLOAT4 color = vertexColors->at(i);  
        vertices.push_back(BasicMaterialVertex(XMFLOAT4  
(position.x, position.y, position.z, 1.0f), color));  
    }  
}  
else  
{  
    XMFLOAT4 color = XMFLOAT4(reinterpret_cast<const  
float*>(&ColorHelper::White));  
    for (UINT i = 0; i < sourceVertices.size(); i++)  
    {  
        XMFLOAT3 position = sourceVertices.at(i);  
        vertices.push_back(BasicMaterialVertex(XMFLOAT4  
(position.x, position.y, position.z, 1.0f), color));  
    }  
}  
  
CreateVertexBuffer(device, &vertices[0], vertices.size(),  
vertexBuffer);  
}  
  
void BasicMaterial::CreateVertexBuffer(ID3D11Device* device,  
BasicMaterialVertex* vertices, UINT vertexCount, ID3D11Buffer**  
vertexBuffer) const  
{  
    D3D11_BUFFER_DESC vertexBufferDesc;  
    ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));  
    vertexBufferDesc.ByteWidth = VertexSize() * vertexCount;  
    vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;  
    vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;  
  
    D3D11_SUBRESOURCE_DATA vertexSubResourceData;  
    ZeroMemory(&vertexSubResourceData, sizeof  
(vertexSubResourceData));  
    vertexSubResourceData.pSysMem = vertices;  
    if (FAILED(device->CreateBuffer(&vertexBufferDesc,  
&vertexSubResourceData, vertexBuffer)))  
    {
```

```
        throw GameException("ID3D11Device::CreateBuffer()\n"
failed.");  
    }  
}  
  
UINT BasicMaterial::VertexSize() const  
{  
    return sizeof(BasicMaterialVertex);  
}  
}
```

---

```
#pragma once

#include "DrawableGameComponent.h"

using namespace Library;

namespace Library
{
    class Effect;
    class BasicMaterial;
}

namespace Rendering
{
    class MaterialDemo : public DrawableGameComponent
    {
        RTTI_DECLARATIONS(MaterialDemo, DrawableGameComponent)

    public:
        MaterialDemo(Game& game, Camera& camera);
        ~MaterialDemo();

        virtual void Initialize() override;
        virtual void Draw(const GameTime& gameTime) override;

    private:
        MaterialDemo();
        MaterialDemo(const MaterialDemo& rhs);
        MaterialDemo& operator=(const MaterialDemo& rhs);

        Effect* mBasicEffect;
        BasicMaterial* mBasicMaterial;
        ID3D11Buffer* mVertexBuffer;
        ID3D11Buffer* mIndexBuffer;
        UINT mIndexCount;

        XMFLOAT4X4 mWorldMatrix;
    };
}

```

```
#include "MaterialDemo.h"
#include "Game.h"
#include "GameException.h"
#include "MatrixHelper.h"
#include "Camera.h"
#include "Utility.h"
#include "Model.h"
#include "Mesh.h"
#include "BasicMaterial.h"

namespace Rendering
{
    RTTI_DEFINITIONS(MaterialDemo)

    MaterialDemo::MaterialDemo(Game& game, Camera& camera)
        : DrawableGameComponent(game, camera),
          mBasicMaterial(nullptr), mBasicEffect(nullptr), mWorldMatrix
(MatrixHelper::Identity),
          mVertexBuffer(nullptr), mIndexBuffer(nullptr), mIndexCount(0)
    {
    }

    MaterialDemo::~MaterialDemo()
    {
        DeleteObject(mBasicMaterial);
        DeleteObject(mBasicEffect);
        ReleaseObject(mVertexBuffer);
        ReleaseObject(mIndexBuffer);
    }

    void MaterialDemo::Initialize()
    {
        SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

        // Load the model
        std::unique_ptr<Model> model(new Model(*mGame, "Content\\
Models\\Sphere.obj", true));

        // Initialize the material
        mBasicEffect = new Effect(*mGame);
        mBasicEffect->LoadCompiledEffect(L"Content\\Effects\\
BasicEffect.cso");
        mBasicMaterial = new BasicMaterial();
        mBasicMaterial->Initialize(mBasicEffect);

        // Create the vertex and index buffers
```

```
Mesh* mesh = model->Meshes().at(0);

mBasicMaterial->CreateVertexBuffer(mGame->Direct3DDevice() ,
*mesh, &mVertexBuffer);
mesh->CreateIndexBuffer(&mIndexBuffer);
mIndexCount = mesh->Indices().size();
}

void MaterialDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mBasicMaterial->CurrentTechnique()->Passes() .
at(0);
    ID3D11InputLayout* inputLayout = mBasicMaterial-
>InputLayouts().at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);

    UINT stride = mBasicMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_-
FORMAT_R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mBasicMaterial->WorldViewProjection() << wvp;

    pass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}
}
```

---

```
class SkyboxMaterial : public Material
{
    RTTI_DECLARATIONS(SkyboxMaterial, Material)

    MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)
    MATERIAL_VARIABLE_DECLARATION(SkyboxTexture)

public:
    SkyboxMaterial();

    virtual void Initialize(Effect* effect) override;
    virtual void CreateVertexBuffer(ID3D11Device* device, const Mesh&
mesh, ID3D11Buffer** vertexBuffer) const override;
    void CreateVertexBuffer(ID3D11Device* device, XMFLOAT4* vertices,
UINT vertexCount, ID3D11Buffer** vertexBuffer) const;
    virtual UINT VertexSize() const override;
};
```

---

```
SkyboxMaterial::SkyboxMaterial()
    : Material("main11"),
      MATERIAL_VARIABLE_INITIALIZATION(WorldViewProjection),
      MATERIAL_VARIABLE_INITIALIZATION(SkyboxTexture)
{
}

MATERIAL_VARIABLE_DEFINITION(SkyboxMaterial, WorldViewProjection)
MATERIAL_VARIABLE_DEFINITION(SkyboxMaterial, SkyboxTexture)

void SkyboxMaterial::Initialize(Effect* effect)
{
    Material::Initialize(effect);

    MATERIAL_VARIABLE_RETRIEVE(WorldViewProjection)
    MATERIAL_VARIABLE_RETRIEVE(SkyboxTexture)

    D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 }
    };

    CreateInputLayout("main11", "p0", inputElementDescriptions,
ARRAYSIZE(inputElementDescriptions));
}
```

---

```
class Skybox : public DrawableGameComponent
{
    RTTI_DECLARATIONS(Skybox, DrawableGameComponent)

public:
    Skybox(Game& game, Camera& camera, const std::wstring&
cubeMapFileName, float scale);
    ~Skybox();

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    Skybox();
    Skybox(const Skybox& rhs);
    Skybox& operator=(const Skybox& rhs);

    std::wstring mCubeMapFileName;
    Effect* mEffect;
    SkyboxMaterial* mMaterial;
    ID3D11ShaderResourceView* mCubeMapShaderResourceView;
    ID3D11Buffer* mVertexBuffer;
    ID3D11Buffer* mIndexBuffer;
    UINT mIndexCount;

    XMFLOAT4X4 mWorldMatrix;
    XMFLOAT4X4 mScaleMatrix;
};
```

---

```
void Skybox::Initialize()
{
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

    std::unique_ptr<Model> model(new Model(*mGame, "Content\\Models\\Sphere.obj", true));

    mEffect = new Effect(*mGame);
    mEffect->LoadCompiledEffect(L"Content\\Effects\\Skybox.cso");

    mMaterial = new SkyboxMaterial();
    mMaterial->Initialize(mEffect);

    Mesh* mesh = model->Meshes().at(0);
    mMaterial->CreateVertexBuffer(mGame->Direct3DDevice(), *mesh,
&mVertexBuffer);
    mesh->CreateIndexBuffer(&mIndexBuffer);
    mIndexCount = mesh->Indices().size();

    HRESULT hr = DirectX::CreateDDSTextureFromFile
(mGame->Direct3DDevice(), mCubeMapFileName.c_str(), nullptr,
&mCubeMapShaderResourceView);
    if (FAILED(hr))
    {
        throw GameException("CreateDDSTextureFromFile() failed.", hr);
    }
}

void Skybox::Update(const GameTime& gameTime)
{
    const XMFLOAT3& position = mCamera->Position();

    XMStoreFloat4x4(&mWorldMatrix, XMLoadFloat4x4(&mScaleMatrix) *
XMMatrixTranslation(position.x, position.y, position.z));
}

void Skybox::Draw(const GameTime& gametime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mMaterial->CurrentTechnique()->Passes().at(0);
    ID3D11InputLayout* inputLayout = mMaterial->InputLayouts().
```

```
at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);

    UINT stride = mMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_FORMAT_
R32_UINT, 0);

    XMATRIX wvp = XMLoadFloat4x4(&mWorldMatrix) * mCamera->
ViewMatrix() * mCamera->ProjectionMatrix();
    mMaterial->WorldViewProjection() << wvp;
    mMaterial->SkyboxTexture() << mCubeMapShaderResourceView;

    pass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}
```

---

```
mSkybox = new Skybox(*this, *mCamera, L"Content\\Textures\\  
Maskonaive2_1024.dds", 500.0f);  
mComponents.push_back(mSkybox);
```

```
typedef struct _DiffuseLightingMaterialVertex
{
    XMFLOAT4 Position;
    XMFLOAT2 TextureCoordinates;
    XMFLOAT3 Normal;

    _DiffuseLightingMaterialVertex() { }

    _DiffuseLightingMaterialVertex(XMFLOAT4 position, XMFLOAT2
textureCoordinates, XMFLOAT3 normal)
        : Position(position), TextureCoordinates(textureCoordinates),
Normal(normal) { }
} DiffuseLightingMaterialVertex;

class DiffuseLightingMaterial : public Material
{
    RTTI_DECLARATIONS(DiffuseLightingMaterial, Material)

    MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)
    MATERIAL_VARIABLE_DECLARATION(World)
    MATERIAL_VARIABLE_DECLARATION(AmbientColor)
    MATERIAL_VARIABLE_DECLARATION(LightColor)
    MATERIAL_VARIABLE_DECLARATION(LightDirection)
    MATERIAL_VARIABLE_DECLARATION(ColorTexture)

public:
    DiffuseLightingMaterial();

    virtual void Initialize(Effect* effect) override;
    virtual void CreateVertexBuffer(ID3D11Device* device, const Mesh&
mesh, ID3D11Buffer** vertexBuffer) const override;
    void CreateVertexBuffer(ID3D11Device* device,
DiffuseLightingMaterialVertex* vertices, UINT vertexCount,
ID3D11Buffer** vertexBuffer) const;
    virtual UINT VertexSize() const override;
};
```

```
#include "DiffuseLightingMaterial.h"
#include "GameException.h"
#include "Mesh.h"

namespace Rendering
{
    RTTI_DEFINITIONS(DiffuseLightingMaterial)

    DiffuseLightingMaterial::DiffuseLightingMaterial()
        : Material("main11"),
          MATERIAL_VARIABLE_INITIALIZATION(WorldViewProjection),
          MATERIAL_VARIABLE_INITIALIZATION(World),
          MATERIAL_VARIABLE_INITIALIZATION(AmbientColor),
          MATERIAL_VARIABLE_INITIALIZATION(LightColor),
          MATERIAL_VARIABLE_INITIALIZATION(LightDirection),
          MATERIAL_VARIABLE_INITIALIZATION(ColorTexture)
    {
    }

    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
WorldViewProjection)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial, World)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial, AmbientColor)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial, LightColor)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial,
LightDirection)
    MATERIAL_VARIABLE_DEFINITION(DiffuseLightingMaterial, ColorTexture)

    void DiffuseLightingMaterial::Initialize(Effect* effect)
    {
        Material::Initialize(effect);

        MATERIAL_VARIABLE_RETRIEVE(WorldViewProjection)
        MATERIAL_VARIABLE_RETRIEVE(World)
        MATERIAL_VARIABLE_RETRIEVE(AmbientColor)
        MATERIAL_VARIABLE_RETRIEVE(LightColor)
        MATERIAL_VARIABLE_RETRIEVE(LightDirection)
        MATERIAL_VARIABLE_RETRIEVE(ColorTexture)

        D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
```

```

D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_
ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D11_
APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

CreateInputLayout("main11", "p0", inputElementDescriptions,
ARRAYSIZE(inputElementDescriptions));
}

void DiffuseLightingMaterial::CreateVertexBuffer(ID3D11Device*
device, const Mesh& mesh, ID3D11Buffer** vertexBuffer) const
{
    const std::vector<XMFLOAT3>& sourceVertices = mesh.Vertices();
    std::vector<XMFLOAT3>* textureCoordinates = mesh.
TextureCoordinates().at(0);
    assert(textureCoordinates->size() == sourceVertices.size());
    const std::vector<XMFLOAT3>& normals = mesh.Normals();
    assert(textureCoordinates->size() == sourceVertices.size());

    std::vector<DiffuseLightingMaterialVertex> vertices;
    vertices.reserve(sourceVertices.size());
    for (UINT i = 0; i < sourceVertices.size(); i++)
    {
        XMFLOAT3 position = sourceVertices.at(i);
        XMFLOAT3 uv = textureCoordinates->at(i);
        XMFLOAT3 normal = normals.at(i);

        vertices.push_back(DiffuseLightingMaterialVertex(XMFLOAT4(position.x,
position.y, position.z, 1.0f), XMFLOAT2(uv.x, uv.y), normal));
    }

    CreateVertexBuffer(device, &vertices[0], vertices.size(),
vertexBuffer);
}

void DiffuseLightingMaterial::CreateVertexBuffer(ID3D11Device*
device, DiffuseLightingMaterialVertex* vertices, UINT vertexCount,
ID3D11Buffer** vertexBuffer) const
{
}

```

```
D3D11_BUFFER_DESC vertexBufferDesc;
ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
vertexBufferDesc.ByteWidth = VertexSize() * vertexCount;
vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

D3D11_SUBRESOURCE_DATA vertexSubResourceData;
ZeroMemory(&vertexSubResourceData, sizeof
(vertexSubResourceData));
vertexSubResourceData.pSysMem = vertices;
if (FAILED(device->CreateBuffer(&vertexBufferDesc,
&vertexSubResourceData, vertexBuffer)))
{
    throw GameException("ID3D11Device::CreateBuffer()
failed.");
}
}

UINT DiffuseLightingMaterial::VertexSize() const
{
    return sizeof(DiffuseLightingMaterialVertex);
}
```

---

```
class DiffuseLightingDemo : public DrawableGameComponent
{
    RTTI_DECLARATIONS(DiffuseLightingDemo, DrawableGameComponent)

public:
    DiffuseLightingDemo(Game& game, Camera& camera);
    ~DiffuseLightingDemo();

    virtual void Initialize() override;
    virtual void Update(const GameTime& gameTime) override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    DiffuseLightingDemo();
    DiffuseLightingDemo(const DiffuseLightingDemo& rhs);
    DiffuseLightingDemo& operator=(const DiffuseLightingDemo& rhs);

    void UpdateAmbientLight(const GameTime& gameTime);
    void UpdateDirectionalLight(const GameTime& gameTime);

    static const float AmbientModulationRate;
    static const XMFLOAT2 RotationRate;

    Effect* mEffect;
    DiffuseLightingMaterial* mMaterial;
    ID3D11ShaderResourceView* mTextureShaderResourceView;
    ID3D11Buffer* mVertexBuffer;
    ID3D11Buffer* mIndexBuffer;
    UINT mIndexCount;

    XMCOLOR mAmbientColor;
    DirectionalLight* mDirectionalLight;
    Keyboard* mKeyboard;
    XMFLOAT4X4 mWorldMatrix;

    ProxyModel* mProxyModel;
};
```

```
void DiffuseLightingDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mMaterial->CurrentTechnique()->Passes().at(0);
    ID3D11InputLayout* inputLayout = mMaterial->InputLayouts().
at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);

    UINT stride = mMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_FORMAT_
R32_UINT, 0);

    XMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
    XMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    XMVECTOR ambientColor = XMLoadColor(&mAmbientColor);

    mMaterial->WorldViewProjection() << wvp;
    mMaterial->World() << worldMatrix;
    mMaterial->AmbientColor() << ambientColor;
    mMaterial->LightColor() << mDirectionalLight->ColorVector();
    mMaterial->LightDirection() <<
mDirectionalLight->DirectionVector();
    mMaterial->ColorTexture() << mTextureShaderResourceView;

    pass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);

    mProxyModel->Draw(gameTime);
}
```

```
void DiffuseLightingDemo::UpdateAmbientLight(const GameTime& gameTime)
{
    static float ambientIntensity = mAmbientColor.a;

    if (mKeyboard != nullptr)
    {

        if (mKeyboard->IsKeyDown(DIK_PGUP) && ambientIntensity <
UCHAR_MAX)
        {
            ambientIntensity += AmbientModulationRate *
(float)gameTime.ElapsedGameTime();
            mAmbientColor.a = XMMin<UCHAR>(ambientIntensity,
UCHAR_MAX);
        }

        if (mKeyboard->IsKeyDown(DIK_PGDN) && ambientIntensity > 0)
        {
            ambientIntensity -= LightModulationRate * (float)gameTime.
ElapsedGameTime();
            mAmbientColor.a = XMMax<UCHAR>(ambientIntensity, 0);
        }
    }
}
```

---

```
void DiffuseLightingDemo::UpdateDirectionalLight(const GameTime&
gameTime)
{
    float elapsedTime = (float)gameTime.ElapsedGameTime();

    XMFLOAT2 rotationAmount = Vector2Helper::Zero;
    if (mKeyboard->IsKeyDown(DIK_LEFTARROW))
    {
        rotationAmount.x += LightRotationRate.x * elapsedTime;
    }
    if (mKeyboard->IsKeyDown(DIK_RIGHTARROW))
    {
        rotationAmount.x -= LightRotationRate.x * elapsedTime;
    }
    if (mKeyboard->IsKeyDown(DIK_UPARROW))
    {
        rotationAmount.y += LightRotationRate.y * elapsedTime;
    }
    if (mKeyboard->IsKeyDown(DIK_DOWNARROW))
    {
        rotationAmount.y -= LightRotationRate.y * elapsedTime;
    }

    XMMATRIX lightRotationMatrix = XMMatrixIdentity();
    if (rotationAmount.x != 0)
    {
        lightRotationMatrix = XMMatrixRotationY(rotationAmount.x);
    }

    if (rotationAmount.y != 0)
    {
        XMMATRIX lightRotationAxisMatrix = XMMatrixRotationAxis
(mDirectionalLight->RightVector(), rotationAmount.y);
        lightRotationMatrix *= lightRotationAxisMatrix;
    }

    if (rotationAmount.x != 0.0f || rotationAmount.y != 0.0f)
    {
        mDirectionalLight->ApplyRotation(lightRotationMatrix);
        mProxyModel->ApplyRotation(lightRotationMatrix);
    }
}
```

```
MATERIAL_VARIABLE_DECLARATION(WorldViewProjection)
MATERIAL_VARIABLE_DECLARATION(World)
MATERIAL_VARIABLE_DECLARATION(SpecularColor)
MATERIAL_VARIABLE_DECLARATION(SpecularPower)
MATERIAL_VARIABLE_DECLARATION(AmbientColor)
MATERIAL_VARIABLE_DECLARATION(LightColor)
MATERIAL_VARIABLE_DECLARATION(LightPosition)
MATERIAL_VARIABLE_DECLARATION(LightRadius)
MATERIAL_VARIABLE_DECLARATION(CameraPosition)
MATERIAL_VARIABLE_DECLARATION(ColorTexture)
```

```
XMMATRIX worldMatrix = XMLoadFloat4x4(&mWorldMatrix);
XMMATRIX wvp = worldMatrix * mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
XMVECTOR ambientColor = XMLoadColor(&mAmbientColor);
XMVECTOR specularColor = XMLoadColor(&mSpecularColor);

mMaterial->WorldViewProjection() << wvp;
mMaterial->World() << worldMatrix;
mMaterial->SpecularColor() << specularColor;
mMaterial->SpecularPower() << mSpecularPower;
mMaterial->AmbientColor() << ambientColor;
mMaterial->LightColor() << mPointLight->ColorVector();
mMaterial->LightPosition() << mPointLight->PositionVector();
mMaterial->LightRadius() << mPointLight->Radius();
mMaterial->ColorTexture() << mTextureShaderResourceView;
mMaterial->CameraPosition() << mCamera->PositionVector();
```

```
void PointLightDemo::UpdatePointLight(const GameTime& gameTime)
{
    XMFLOAT3 movementAmount = Vector3Helper::Zero;
    if (mKeyboard != nullptr)
    {
        if (mKeyboard->IsKeyDown(DIK_NUMPAD4))
        {
            movementAmount.x = -1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_NUMPAD6))
        {
            movementAmount.x = 1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_NUMPAD9))
        {
            movementAmount.y = 1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_NUMPAD3))
        {
            movementAmount.y = -1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_NUMPAD8))
        {
            movementAmount.z = -1.0f;
        }

        if (mKeyboard->IsKeyDown(DIK_NUMPAD2))
        {
            movementAmount.z = 1.0f;
        }
    }

    XMVECTOR movement = XMLoadFloat3(&movementAmount) * MovementRate *
(float)gameTime.ElapsedGameTime();
    mPointLight->SetPosition(mPointLight->PositionVector() + movement);
    mProxyModel->SetPosition(mPointLight->Position());
}
```

```
class FullScreenRenderTarget
{
public:
    FullScreenRenderTarget(Game& game);
    ~FullScreenRenderTarget();

    ID3D11ShaderResourceView* OutputTexture() const;
    ID3D11RenderTargetView* RenderTargetView() const;
    ID3D11DepthStencilView* DepthStencilView() const;

    void Begin();
    void End();

private:
    FullScreenRenderTarget();
    FullScreenRenderTarget(const FullScreenRenderTarget& rhs);
    FullScreenRenderTarget& operator=(const FullScreenRenderTarget& rhs);

    Game* mGame;
    ID3D11RenderTargetView* mRenderTargetView;
    ID3D11DepthStencilView* mDepthStencilView;
    ID3D11ShaderResourceView* mOutputTexture;
};


```

---

```
#include "FullScreenRenderTarget.h"
#include "Game.h"
#include "GameException.h"

namespace Library
{
    FullScreenRenderTarget::FullScreenRenderTarget(Game& game)
        : mGame(&game), mRenderTargetView(nullptr),
        mDepthStencilView(nullptr), mOutputTexture(nullptr)
    {
        D3D11_TEXTURE2D_DESC fullScreenTextureDesc;
        ZeroMemory(&fullScreenTextureDesc, sizeof
(fullScreenTextureDesc));
        fullScreenTextureDesc.Width = game.ScreenWidth();
        fullScreenTextureDesc.Height = game.ScreenHeight();
        fullScreenTextureDesc.MipLevels = 1;
        fullScreenTextureDesc.ArraySize = 1;
        fullScreenTextureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        fullScreenTextureDesc.SampleDesc.Count = 1;
        fullScreenTextureDesc.SampleDesc.Quality = 0;
        fullScreenTextureDesc.BindFlags = D3D11_BIND_RENDER_TARGET |
D3D11_BIND_SHADER_RESOURCE;
        fullScreenTextureDesc.Usage = D3D11_USAGE_DEFAULT;

        HRESULT hr;
        ID3D11Texture2D* fullScreenTexture = nullptr;
        if (FAILED(hr = game.Direct3DDevice()->CreateTexture2D
(&fullScreenTextureDesc, nullptr, &fullScreenTexture)))
        {
            throw GameException("IDXGIDevice::CreateTexture2D()
failed.", hr);
        }

        if (FAILED(hr = game.Direct3DDevice()->CreateShaderResourceView
(fullScreenTexture, nullptr, &mOutputTexture)))
        {
            ReleaseObject(fullScreenTexture);
            throw GameException("IDXGIDevice::CreateShaderResource
View() failed.", hr);
        }

        if (FAILED(hr = game.Direct3DDevice()->CreateRenderTargetView
```

```
(fullScreenTexture, nullptr, &mRenderTargetView));
{
    ReleaseObject(fullScreenTexture);
    throw GameException("IDXGIDevice::CreateRenderTargetView() failed.", hr);
}

ReleaseObject(fullScreenTexture);

D3D11_TEXTURE2D_DESC depthStencilDesc;
ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));
depthStencilDesc.Width = game.ScreenWidth();
depthStencilDesc.Height = game.ScreenHeight();
depthStencilDesc.MipLevels = 1;
depthStencilDesc.ArraySize = 1;
depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthStencilDesc.SampleDesc.Count = 1;
depthStencilDesc.SampleDesc.Quality = 0;
depthStencilDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;

ID3D11Texture2D* depthStencilBuffer = nullptr;
if (FAILED(hr = game.Direct3DDevice() ->CreateTexture2D(&depthStencilDesc, nullptr, &depthStencilBuffer)))
{
    throw GameException("IDXGIDevice::CreateTexture2D() failed.", hr);
}

if (FAILED(hr = game.Direct3DDevice() ->CreateDepthStencilView(depthStencilBuffer, nullptr, &mDepthStencilView)))
{
    ReleaseObject(depthStencilBuffer);
    throw GameException("IDXGIDevice::CreateDepthStencilView() failed.", hr);
}

ReleaseObject(depthStencilBuffer);
}

FullScreenRenderTarget::~FullScreenRenderTarget()
{
    ReleaseObject(mOutputTexture);
    ReleaseObject(mDepthStencilView);
    ReleaseObject(mRenderTargetView);
```

```
        }
```

```
        ID3D11ShaderResourceView* FullScreenRenderTarget::OutputTexture()
```

```
const
```

```
    {
```

```
        return mOutputTexture;
```

```
    }
```

```
        ID3D11RenderTargetView* FullScreenRenderTarget::RenderTargetView()
```

```
const
```

```
    {
```

```
        return mRenderTargetView;
```

```
    }
```

```
        ID3D11DepthStencilView* FullScreenRenderTarget::DepthStencilView()
```

```
const
```

```
    {
```

```
        return mDepthStencilView;
```

```
    }
```

```
        void FullScreenRenderTarget::Begin()
```

```
{
```

```
        mGame->Direct3DDeviceContext()->OMSetRenderTargets(1,
```

```
&mRenderTargetView, mDepthStencilView);
```

```
}
```

```
        void FullScreenRenderTarget::End()
```

```
{
```

```
        mGame->ResetRenderTargets();
```

```
}
```

```
}
```

---

```
mRenderTarget->Begin();
// 1. Clear mRenderTarget->RenderTargetView()
// 2. Clear mRenderTarget->DepthStencilView()
// 3. Draw Objects
mRenderTarget->End();
```

```
class FullScreenQuad : public DrawableGameComponent
{
    RTTI_DECLARATIONS(FullScreenQuad, DrawableGameComponent)

public:
    FullScreenQuad(Game& game);
    FullScreenQuad(Game& game, Material& material);
    ~FullScreenQuad();

    Material* GetMaterial();
    void SetMaterial(Material& material, const std::string&
techniqueName, const std::string& passName);
    void SetActiveTechnique(const std::string& techniqueName, const
std::string& passName);
    void SetCustomUpdateMaterial(std::function<void()> callback);

    virtual void Initialize() override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    FullScreenQuad();
    FullScreenQuad(const FullScreenQuad& rhs);
    FullScreenQuad& operator=(const FullScreenQuad& rhs);

    Material* mMaterial;
    Pass* mPass;
    ID3D11InputLayout* mInputLayout;
    ID3D11Buffer* mVertexBuffer;
    ID3D11Buffer* mIndexBuffer;
    UINT mIndexCount;
    std::function<void()> mCustomUpdateMaterial;
};


```

---

```
#include "FullScreenQuad.h"
#include "Game.h"
#include "GameException.h"
#include "Material.h"
#include "VertexDeclarations.h"

namespace Library
{
    RTTI_DEFINITIONS(FullScreenQuad)

    FullScreenQuad::FullScreenQuad(Game& game)
        : DrawableGameComponent(game),
          mMaterial(nullptr), mPass(nullptr), mInputLayout(nullptr),
          mVertexBuffer(nullptr), mIndexBuffer(nullptr),
          mIndexCount(0), mCustomUpdateMaterial(nullptr)
    {
    }

    FullScreenQuad::FullScreenQuad(Game& game, Material& material)
        : DrawableGameComponent(game),
          mMaterial(&material), mPass(nullptr), mInputLayout(nullptr),
          mVertexBuffer(nullptr), mIndexBuffer(nullptr),
          mIndexCount(0), mCustomUpdateMaterial(nullptr)
    {
    }

    FullScreenQuad::~FullScreenQuad()
    {
        ReleaseObject(mIndexBuffer);
        ReleaseObject(mVertexBuffer);
    }

    Material* FullScreenQuad::GetMaterial()
    {
        return mMaterial;
    }

    void FullScreenQuad::SetMaterial(Material& material, const
std::string& techniqueName, const std::string& passName)
    {
        mMaterial = &material;
        SetActiveTechnique(techniqueName, passName);
    }
}
```

```
void FullScreenQuad::SetActiveTechnique(const std::string&
techniqueName, const std::string& passName)
{
    Technique* technique = mMaterial->GetEffect()
->TechniquesByName().at(techniqueName);
    assert(technique != nullptr);

    mPass = technique->PassesByName().at(passName);
    assert(mPass != nullptr);
    mInputLayout = mMaterial->InputLayouts().at(mPass);
}

void FullScreenQuad::SetCustomUpdateMaterial(std::function<void()>
callback)
{
    mCustomUpdateMaterial = callback;
}

void FullScreenQuad::Initialize()
{
    PositionTextureVertex vertices[] =
    {
        PositionTextureVertex(XMFLOAT4(-1.0f, -1.0f, 0.0f, 1.0f),
XMFLOAT2(0.0f, 1.0f)),
        PositionTextureVertex(XMFLOAT4(-1.0f, 1.0f, 0.0f, 1.0f),
XMFLOAT2(0.0f, 0.0f)),
        PositionTextureVertex(XMFLOAT4(1.0f, 1.0f, 0.0f, 1.0f),
XMFLOAT2(1.0f, 0.0f)),
        PositionTextureVertex(XMFLOAT4(1.0f, -1.0f, 0.0f, 1.0f),
XMFLOAT2(1.0f, 1.0f)),
    };

    D3D11_BUFFER_DESC vertexBufferDesc;
    ZeroMemory(&vertexBufferDesc, sizeof(vertexBufferDesc));
    vertexBufferDesc.ByteWidth = sizeof(PositionTextureVertex) *
ARRAYSIZE(vertices);
    vertexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
    vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

    D3D11_SUBRESOURCE_DATA vertexSubResourceData;
    ZeroMemory(&vertexSubResourceData, sizeof
(vertexSubResourceData));
    vertexSubResourceData.pSysMem = vertices;
```

```
    if (FAILED(mGame->Direct3DDevice()->CreateBuffer
(&vertexBufferDesc, &vertexSubResourceData, &mVertexBuffer)))
    {
        throw GameException("ID3D11Device::CreateBuffer()
failed.");
    }

    UINT indices[] =
    {
        0, 1, 2,
        0, 2, 3
    };

    mIndexCount = ARRAYSIZE(indices);

    D3D11_BUFFER_DESC indexBufferDesc;
    ZeroMemory(&indexBufferDesc, sizeof(indexBufferDesc));
    indexBufferDesc.ByteWidth = sizeof(UINT) * mIndexCount;
    indexBufferDesc.Usage = D3D11_USAGE_IMMUTABLE;
    indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;
```

```
    D3D11_SUBRESOURCE_DATA indexSubResourceData;
    ZeroMemory(&indexSubResourceData, sizeof
(indexSubResourceData));
    indexSubResourceData.pSysMem = indices;
    if (FAILED(mGame->Direct3DDevice()-
>CreateBuffer(&indexBufferDesc, &indexSubResourceData, &mIndexBuffer)))
    {
        throw GameException("ID3D11Device::CreateBuffer()
failed.");
    }
}

void FullScreenQuad::Draw(const GameTime& gameTime)
{
    assert(mPass != nullptr);
    assert(mInputLayout != nullptr);

    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = sizeof(PositionTextureVertex);
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_
FORMAT_R32_UINT, 0);

    if (mCustomUpdateMaterial != nullptr)
    {
        mCustomUpdateMaterial();
    }

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexed(mIndexCount, 0, 0);
}
}
```

```
static const float3 GrayScaleIntensity = { 0.299f, 0.587f, 0.114f };

/********************* Resources ********************/

Texture2D ColorTexture;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/********************* Data Structures *****/

struct VS_INPUT
{
    float4 Position : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/********************* Vertex Shader *****/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}

/********************* Pixel Shader *****/

float4 grayscale_pixel_shader(VS_OUTPUT IN) : SV_Target
{
```

```
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float intensity = dot(color.rgb, GrayScaleIntensity);

    return float4(intensity.rrr, color.a);
}

/***** Techniques *****/
technique11 grayscale_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
grayscale_pixel_shader()));
    }
}
```

---

```
FullScreenRenderTarget* mRenderTarget;
FullScreenQuad* mFullScreenQuad;
Effect* mColorFilterEffect;
ColorFilterMaterial* mColorFilterMaterial;
```

```
mRenderTarget = new FullScreenRenderTarget(*this);

SetCurrentDirectory(Utility::ExecutableDirectory().c_str());
mColorFilterEffect = new Effect(*this);
mColorFilterEffect->LoadCompiledEffect(L"Content\\Effects\\ColorFilter.
cso");

mColorFilterMaterial = new ColorFilterMaterial();
mColorFilterMaterial->Initialize(*mColorFilterEffect);

mFullScreenQuad = new FullScreenQuad(*this, *mColorFilterMaterial);
mFullScreenQuad->Initialize();
mFullScreenQuad->SetActiveTechnique("grayscale_filter", "p0");
mFullScreenQuad->SetCustomUpdateMaterial(std::bind(&ColorFilteringGame::
UpdateColorFilterMaterial, this));
```

```
void ColorFilteringGame::UpdateColorFilterMaterial()
{
    mColorFilterMaterial->ColorTexture()
<< mRenderTarget->OutputTexture();
}
```

```
void ColorFilteringGame::Draw(const GameTime &gameTime)
{
    // Render the scene to an off-screen texture.
    mRenderTarget->Begin();

    mDirect3DDeviceContext->ClearRenderTargetView(mRenderTarget
->RenderTargetView() , reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext->ClearDepthStencilView(mRenderTarget
->DepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    Game::Draw(gameTime);

    mRenderTarget->End();

    // Render a full-screen quad with a post processing effect.
    mDirect3DDeviceContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&BackgroundColor));
    mDirect3DDeviceContext->ClearDepthStencilView(mDepthStencilView,
D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    mFullScreenQuad->Draw(gameTime);

    HRESULT hr = mSwapChain->Present(0, 0);
    if (FAILED(hr))
    {
        throw GameException("IDXGISwapChain::Present() failed.", hr);
    }
}
```

```
float4 inverse_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return float4(1 - color.rgb, color.a);
}

technique11 inverse_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, inverse_pixel_shader()));
    }
}
```

```
static const float3x3 SepiaFilter = { 0.393f, 0.349f, 0.272f,
                                      0.769f, 0.686f, 0.534f,
                                      0.189f, 0.168f, 0.131f };

float4 sepia_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return float4(mul(color.rgb, SepiaFilter), color.a);
}
technique11 sepia_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, sepia_pixel_shader()));
    }
}
```

---

```
cbuffer CBufferPerObject
{
    float4x4 ColorFilter = { 1, 0, 0, 0,
                             0, 1, 0, 0,
                             0, 0, 1, 0,
                             0, 0, 0, 1 };
}

float4 genericfilter_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return float4(mul(color, ColorFilter).rgb, color.a);
}

technique11 generic_filter
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
genericfilter_pixel_shader()));
    }
}
```

---

```
void ColorFilteringGame::UpdateColorFilterMaterial()
{
    XMATRIX colorFilter = XMLoadFloat4x4(&mGenericColorFilter);

    mColorFilterMaterial->ColorTexture() << mRenderTarget->
OutputTexture();
    mColorFilterMaterial->ColorFilter() << colorFilter;
}
```

```
***** Resources *****

#define SAMPLE_COUNT 9

cbuffer CBufferPerFrame
{
    float2 SampleOffsets[SAMPLE_COUNT];
    float SampleWeights[SAMPLE_COUNT];
}

Texture2D ColorTexture;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = CLAMP;
    AddressV = CLAMP;
};

***** Data Structures *****

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

***** Vertex Shader *****

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.ObjectPosition;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}
```

```
***** Pixel Shaders *****

float4 blur_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = (float4)0;

    for (int i = 0; i < SAMPLE_COUNT; i++)
    {
        color += ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate + SampleOffsets[i]) * SampleWeights[i];
    }

    return color;
}

float4 no_blur_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(TrilinearSampler, IN.TextureCoordinate);
}

***** Techniques *****

technique1 blur
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, blur_pixel_shader()));
    }
}

technique1 no_blur
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, no_blur_pixel_shader()));
    }
}
```

---

```
class GaussianBlur : public DrawableGameComponent
{
    RTTI_DECLARATIONS(GaussianBlur, DrawableGameComponent)

public:
    GaussianBlur(Game& game, Camera& camera);
    GaussianBlur(Game& game, Camera& camera, float blurAmount);
    ~GaussianBlur();

    ID3D11ShaderResourceView* SceneTexture();
    void SetSceneTexture(ID3D11ShaderResourceView& sceneTexture);

    ID3D11ShaderResourceView* OutputTexture();

    float BlurAmount() const;
    void SetBlurAmount(float blurAmount);

    virtual void Initialize() override;
    virtual void Draw(const GameTime& gameTime) override;

private:
    GaussianBlur();
    GaussianBlur(const GaussianBlur& rhs);
    GaussianBlur& operator=(const GaussianBlur& rhs);

    void InitializeSampleWeights();
    void InitializeSampleOffsets();
    float GetWeight(float x) const;
    void UpdateGaussianMaterialWithHorizontalOffsets();
    void UpdateGaussianMaterialWithVerticalOffsets();
    void UpdateGaussianMaterialNoBlur();

    static const float DefaultBlurAmount;

    Effect* mEffect;
    GaussianBlurMaterial* mMaterial;
    ID3D11ShaderResourceView* mSceneTexture;
    FullScreenRenderTarget* mHorizontalBlurTarget;
    FullScreenQuad* mFullScreenQuad;

    std::vector<XMFLOAT2> mHorizontalSampleOffsets;
    std::vector<XMFLOAT2> mVerticalSampleOffsets;
    std::vector<float> mSampleWeights;
    float mBlurAmount;
};

};
```

```
void GaussianBlur::InitializeSampleOffsets()
{
    float horizontalPixelSize = 1.0f / mGame->ScreenWidth();
    float verticalPixelSize = 1.0f / mGame->ScreenHeight();

    UINT sampleCount = mMaterial->SampleOffsets().TypeDesc().Elements;

    mHorizontalSampleOffsets.resize(sampleCount);
    mVerticalSampleOffsets.resize(sampleCount);
    mHorizontalSampleOffsets[0] = Vector2Helper::Zero;
    mVerticalSampleOffsets[0] = Vector2Helper::Zero;

    for (UINT i = 0; i < sampleCount / 2; i++)
    {
        float sampleOffset = i * 2 + 1.5f;
        float horizontalOffset = horizontalPixelSize * sampleOffset;
        float verticalOffset = verticalPixelSize * sampleOffset;

        mHorizontalSampleOffsets[i * 2 + 1] =
XMFLOAT2(horizontalOffset, 0.0f);
        mHorizontalSampleOffsets[i * 2 + 2] = XMFLOAT2
(-horizontalOffset, 0.0f);

        mVerticalSampleOffsets[i * 2 + 1] = XMFLOAT2
(0.0f, verticalOffset);
        mVerticalSampleOffsets[i * 2 + 2] = XMFLOAT2
(0.0f, -verticalOffset);
    }
}

void GaussianBlur::InitializeSampleWeights()
{
    UINT sampleCount = mMaterial->SampleOffsets().TypeDesc().Elements;

    mSampleWeights.resize(sampleCount);
    mSampleWeights[0] = GetWeight(0);

    float totalWeight = mSampleWeights[0];
    for (UINT i = 0; i < sampleCount / 2; i++)
    {
        float weight = GetWeight((float)i + 1);
        mSampleWeights[i * 2 + 1] = weight;
        mSampleWeights[i * 2 + 2] = weight;
        totalWeight += weight * 2;
    }
}
```

```
// Normalize the weights so that they sum to one
for (UINT i = 0; i < mSampleWeights.size(); i++)
{
    mSampleWeights[i] /= totalWeight;
}
}

float GaussianBlur::GetWeight(float x) const
{
    return (float)(exp(-(x * x) / (2 * mBlurAmount * mBlurAmount)));
}
```

---

```
void GaussianBlur::Draw(const GameTime& gameTime)
{
    if (mBlurAmount > 0.0f)
    {
        // Horizontal blur
        mHorizontalBlurTarget->Begin();
        mGame->Direct3DDeviceContext()->ClearRenderTargetView
(mHorizontalBlurTarget->RenderTargetView() , reinterpret_cast<const
float*>(&ColorHelper::Purple));
        mGame->Direct3DDeviceContext()->ClearDepthStencilView
(mHorizontalBlurTarget->DepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_
CLEAR_STENCIL, 1.0f, 0);
        mFullScreenQuad->SetActiveTechnique("blur", "p0");
        mFullScreenQuad->SetCustomUpdateMaterial(std::bind
(&GaussianBlur::UpdateGaussianMaterialWithHorizontalOffsets, this));
        mFullScreenQuad->Draw(gameTime);
        mHorizontalBlurTarget->End();

        // Vertical blur for the final image
        mFullScreenQuad->SetCustomUpdateMaterial(std::bind
(&GaussianBlur::UpdateGaussianMaterialWithVerticalOffsets, this));
        mFullScreenQuad->Draw(gameTime);
    }
    else
    {
        mFullScreenQuad->SetActiveTechnique("no_blur", "p0");
        mFullScreenQuad->SetCustomUpdateMaterial(std::bind
(&GaussianBlur::UpdateGaussianMaterialNoBlur, this));
        mFullScreenQuad->Draw(gameTime);
    }
}

void GaussianBlur::UpdateGaussianMaterialWithHorizontalOffsets()
{
    mMaterial->ColorTexture() << mSceneTexture;
    mMaterial->SampleWeights() << mSampleWeights;
    mMaterial->SampleOffsets() << mHorizontalSampleOffsets;
}

void GaussianBlur::UpdateGaussianMaterialWithVerticalOffsets()
{
```

```
mMaterial->ColorTexture() <<
mHorizontalBlurTarget->OutputTexture();
mMaterial->SampleWeights() << mSampleWeights;
mMaterial->SampleOffsets() << mVerticalSampleOffsets;
}

void GaussianBlur::UpdateGaussianMaterialNoBlur()
{
    mMaterial->ColorTexture() << mSceneTexture;
}
```

---

```
mGaussianBlur = new GaussianBlur(*this, *mCamera);
mGaussianBlur->SetSceneTexture(* (mRenderTarget->OutputTexture()));
mGaussianBlur->Initialize();
```

```
mRenderTarget->Begin();

mDirect3DDeviceContext->ClearRenderTargetView(mRenderTarget->
RenderTargetView() , reinterpret_cast<const float*>(&BackgroundColor));
mDirect3DDeviceContext->ClearDepthStencilView(mRenderTarget->
DepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

Game::Draw(gameTime);

mRenderTarget->End();

mDirect3DDeviceContext->ClearRenderTargetView(mRenderTargetView,
reinterpret_cast<const float*>(&BackgroundColor));
mDirect3DDeviceContext->ClearDepthStencilView(mDepthStencilView, D3D11_
CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

mGaussianBlur->Draw(gameTime);
```

```
***** Resources *****/
static const float3 GrayScaleIntensity = { 0.299f, 0.587f, 0.114f };

Texture2D ColorTexture;
Texture2D BloomTexture;

cbuffer CBufferPerObject
{
    float BloomThreshold = 0.45f;
    float BloomIntensity = 1.25f;
    float BloomSaturation = 1.0f;
    float SceneIntensity = 1.0f;
    float SceneSaturation = 1.0f;
};

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 Position : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/***** Utility Functions *****/
float4 AdjustSaturation(float4 color, float saturation)
{
    float intensity = dot(color.rgb, GrayScaleIntensity);

    return float4(lerp(intensity.rrr, color.rgb, saturation), color.a);
}
```

```
***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}

***** Pixel Shaders *****/
float4 bloom_extract_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    return saturate((color - BloomThreshold) / (1 - BloomThreshold));
}

float4 bloom_composite_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 sceneColor = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float4 bloomColor = BloomTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);

    sceneColor = AdjustSaturation(sceneColor, SceneSaturation) *
SceneIntensity;
    bloomColor = AdjustSaturation(bloomColor, BloomSaturation) *
BloomIntensity;

    sceneColor *= (1 - saturate(bloomColor));

    return sceneColor + bloomColor;
}

float4 no_bloom_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(TrilinearSampler, IN.TextureCoordinate);
}
```

```
***** Techniques *****

technique11 bloom_extract
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
bloom_extract_pixel_shader()));
    }
}

technique11 bloom_composite
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
bloom_composite_pixel_shader()));
    }
}

technique11 no_bloom
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, no_bloom_pixel_shader()));
    }
}
```

---

```
float4 bloom_extract_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float intensity = dot(color.rgb, GrayScaleIntensity);

    return (intensity > BloomThreshold ? color : float4(0, 0, 0, 1));
}
```

```
cbuffer CBufferPerObject
{
    float DisplacementScale = 1.0f;
}

Texture2D SceneTexture;
Texture2D DistortionMap;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = CLAMP;
    AddressV = CLAMP;
};

float4 displacement_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy - 0.5;
    OUT = SceneTexture.Sample(TrilinearSampler, IN.TextureCoordinate +
(DisplacementScale * displacement));

    return OUT;
}
```

---

```
static const float ZeroCorrection = 0.5f / 255.0f;

float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy - 0.5 + ZeroCorrection;
```

```
***** Resources *****/
static const float ZeroCorrection = 0.5f / 255.0f;

cbuffer CBufferPerObjectCutout
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
}

cbuffer CBufferPerObjectComposite
{
    float DisplacementScale = 1.0f;
}

Texture2D SceneTexture;
Texture2D DistortionMap;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/***** Data Structures *****/

struct VS_INPUT
{
    float4 Position : POSITION;
    float2 TextureCoordinate : TEXCOORD;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/***** Cutout *****/

VS_OUTPUT distortion_cutout_vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.Position, WorldViewProjection);
    OUT.TextureCoordinate = IN.TextureCoordinate;
}
```

```
        return OUT;
    }

float4 distortion_cutout_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy;

    return float4(displacement.xy, 0, 1);
}

/***** Distortion Post Processing *****/
VS_OUTPUT distortion_vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.TextureCoordinate = IN.TextureCoordinate;

    return OUT;
}

float4 distortion_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float2 displacement = DistortionMap.Sample(TrilinearSampler,
IN.TextureCoordinate).xy;
    if (displacement.x == 0 && displacement.y == 0)
    {
        OUT = SceneTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    }
    else
    {
        displacement -= 0.5f + ZeroCorrection;
        OUT = SceneTexture.Sample(TrilinearSampler,
IN.TextureCoordinate + (DisplacementScale * displacement));
    }

    return OUT;
}

float4 no_distortion_pixel_shader(VS_OUTPUT IN) : SV_Target
{
```

```
        return SceneTexture.Sample(TrilinearSampler, IN.TextureCoordinate);
    }

/***** Techniques *****/
technique11 distortion_cutout
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
distortion_cutout_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
distortion_cutout_pixel_shader()));
    }
}

technique11 distortion
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
distortion_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
distortion_pixel_shader()));
    }
}

technique11 no_distortion
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
distortion_composite_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
no_distortion_pixel_shader()));
    }
}
```

---

```
#include "include\\Common.fxh"

/********************* Resources *****/
static const float4 ColorWhite = { 1, 1, 1, 1 };

cbuffer CBufferPerFrame
{
    float4 AmbientColor = { 1.0f, 1.0f, 1.0f, 0.0f };
    float4 LightColor = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 LightPosition = { 0.0f, 0.0f, 0.0f };
    float LightRadius = 10.0f;
    float3 CameraPosition;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
    float4 SpecularColor : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
    float SpecularPower : SPECULARPOWER = 25.0f;

    float4x4 ProjectiveTextureMatrix;
}
```

```
Texture2D ColorTexture;
Texture2D ProjectedTexture;

SamplerState ProjectedTextureSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
};

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState BackFaceCulling
{
    CullMode = BACK;
};

/***************** Data Structures ********************/

struct VS_INPUT
```

```
{  
    float4 ObjectPosition : POSITION;  
    float2 TextureCoordinate : TEXCOORD;  
    float3 Normal : NORMAL;  
};  
  
struct VS_OUTPUT  
{  
    float4 Position : SV_Position;  
    float3 Normal : NORMAL;  
    float2 TextureCoordinate : TEXCOORD0;  
    float3 WorldPosition : TEXCOORD1;  
    float Attenuation : TEXCOORD2;  
    float4 ProjectedTextureCoordinate : TEXCOORD3;  
};  
  
/***************** Vertex Shader *****/  
  
VS_OUTPUT project_texture_vertex_shader(VS_INPUT IN)  
{  
    VS_OUTPUT OUT = (VS_OUTPUT)0;  
  
    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);  
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;  
    OUT.TextureCoordinate = IN.TextureCoordinate;  
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);  
    float3 lightDirection = LightPosition - OUT.WorldPosition;  
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /  
LightRadius));
```

```

    OUT.ProjecteTextureCoordinate = mul(IN.ObjectPosition,
ProjectiveTextureMatrix);

    return OUT;
}

/***** Pixel Shaders *****/
float4 project_texture_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor,
color.rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    IN.ProjecteTextureCoordinate.xy /= IN.ProjecteTextureCoordinate.w;
    float3 projectedColor = ProjectedTexture.
Sample(ProjectedTextureSampler, IN.ProjecteTextureCoordinate.xy).rgb;

    OUT.rgb *= projectedColor;
}

```

```
        return OUT;
    }

/****** Techniques *****/

technique11 project_texture
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
project_texture_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
project_texture_pixel_shader()));

        SetRasterizerState(BackFaceCulling);
    }
}
```

---

```
if (IN.ProjectedTextureCoordinate.w >= 0.0f)
{
    IN.ProjectedTextureCoordinate.xy /= IN.ProjectedTextureCoordinate.w;
    float3 projectedColor = ProjectedTexture.
Sample(ProjectedTextureSampler, IN.ProjectedTextureCoordinate.xy).rgb;

    OUT.rgb *= projectedColor;
}
```

```
class DepthMap : public RenderTarget
{
    RTTI_DECLARATIONS(DepthMap, RenderTarget)

public:
    DepthMap(Game& game, UINT width, UINT height);
    ~DepthMap();

    ID3D11ShaderResourceView* OutputTexture() const;
    ID3D11DepthStencilView* DepthStencilView() const;

    virtual void Begin() override;
    virtual void End() override;

private:
    DepthMap();
    DepthMap(const DepthMap& rhs);
    DepthMap& operator=(const DepthMap& rhs);

    Game* mGame;
    ID3D11DepthStencilView* mDepthStencilView;
    ID3D11ShaderResourceView* mOutputTexture;
    D3D11_VIEWPORT mViewPort;
};


```

---

```
#include "DepthMap.h"
#include "Game.h"
#include "GameException.h"

namespace Library
{
    RTTI_DEFINITIONS(DepthMap)

    DepthMap::DepthMap(Game& game, UINT width, UINT height)
        : RenderTarget(), mGame(&game), mDepthStencilView(nullptr),
          mOutputTexture(nullptr), mViewport()
    {
        D3D11_TEXTURE2D_DESC textureDesc;
        ZeroMemory(&textureDesc, sizeof(textureDesc));
        textureDesc.Width = width;
        textureDesc.Height = height;
        textureDesc.MipLevels = 1;
        textureDesc.ArraySize = 1;
        textureDesc.Format = DXGI_FORMAT_R24G8_TYPELESS;
        textureDesc.SampleDesc.Count = 1;
        textureDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL |
D3D11_BIND_SHADER_RESOURCE;

        HRESULT hr;
        ID3D11Texture2D* texture = nullptr;
        if (FAILED(hr = game.Direct3DDevice()->CreateTexture2D
(&textureDesc, nullptr, &texture)))
        {
            throw GameException("IDXGIDevice::CreateTexture2D() failed.", hr);
        }

        D3D11_SHADER_RESOURCE_VIEW_DESC resourceViewDesc;
        ZeroMemory(&resourceViewDesc, sizeof(resourceViewDesc));
        resourceViewDesc.Format = DXGI_FORMAT_R24_UNORM_X8_TYPELESS;
        resourceViewDesc.ViewDimension = D3D_SRV_DIMENSION_TEXTURE2D;
        resourceViewDesc.Texture2D.MipLevels = 1;

        if (FAILED(hr = game.Direct3DDevice()
->CreateShaderResourceView(texture, &resourceViewDesc,
&mOutputTexture)))
        {
            ReleaseObject(texture);
            throw GameException("IDXGIDevice::CreateShaderResource
View() failed.", hr);
        }
    }
}
```

```
D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc;
ZeroMemory(&depthStencilViewDesc, sizeof
(depthStencilViewDesc));
    depthStencilViewDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
    depthStencilViewDesc.ViewDimension =
D3D11_DSV_DIMENSION_TEXTURE2D;
    depthStencilViewDesc.Texture2D.MipSlice = 0;

    if (FAILED(hr = game.Direct3DDevice() ->CreateDepthStencilView
(texture, &depthStencilViewDesc, &mDepthStencilView)))
{
    ReleaseObject(texture);
    throw GameException("IDXGIDevice::CreateDepthStencilView()
failed.", hr);
}

ReleaseObject(texture);

mViewport.TopLeftX = 0.0f;
mViewport.TopLeftY = 0.0f;
mViewport.Width = static_cast<float>(width);
mViewport.Height = static_cast<float>(height);
mViewport.MinDepth = 0.0f;
mViewport.MaxDepth = 1.0f;
```

```
}

DepthMap::~DepthMap()
{
    ReleaseObject(mOutputTexture);
    ReleaseObject(mDepthStencilView);
}

ID3D11ShaderResourceView* DepthMap::OutputTexture() const
{
    return mOutputTexture;
}

ID3D11DepthStencilView* DepthMap::DepthStencilView() const
{
    return mDepthStencilView;
}

void DepthMap::Begin()
{
    static ID3D11RenderTargetView* nullRenderTargetView = nullptr;
    RenderTarget::Begin(mGame->Direct3DDeviceContext(), 1,
&nullRenderTargetView, mDepthStencilView, mViewport);
}
void DepthMap::End()
{
    RenderTarget::End(mGame->Direct3DDeviceContext());
}
}
```

---

```
class RenderTarget : public RTTI
{
    RTTI_DECLARATIONS(RenderTarget, RTTI)

public:
    RenderTarget();
    virtual ~RenderTarget();

    virtual void Begin() = 0;
    virtual void End() = 0;

protected:
    typedef struct _RenderTargetData
    {
        UINT ViewCount;
        ID3D11RenderTargetView** RenderTargetViews;
        ID3D11DepthStencilView* DepthStencilView;
        D3D11_VIEWPORT Viewport;

        _RenderTargetData(UINT viewCount, ID3D11RenderTargetView** renderTargetViews, ID3D11DepthStencilView* depthStencilView, const D3D11_VIEWPORT& viewport)
            : ViewCount(viewCount), RenderTargetViews(renderTargetViews), DepthStencilView(depthStencilView), Viewport(viewport) { }
    } RenderTargetData;

    void Begin(ID3D11DeviceContext* deviceContext, UINT viewCount, ID3D11RenderTargetView** renderTargetViews, ID3D11DepthStencilView* depthStencilView, const D3D11_VIEWPORT& viewport);
    void End(ID3D11DeviceContext* deviceContext);

private:
    RenderTarget(const RenderTarget& rhs);
    RenderTarget& operator=(const RenderTarget& rhs);

    static std::stack<RenderTargetData> sRenderTargetStack;
};
```

```
#include "RenderTarget.h"
#include "Game.h"

namespace Library
{
    RTTI_DEFINITIONS(RenderTarget)

    std::stack<RenderTarget::RenderTargetData> RenderTarget::
sRenderTargetStack;

    RenderTarget::RenderTarget()
    {
    }

    RenderTarget::~RenderTarget()
    {
    }

    void RenderTarget::Begin(ID3D11DeviceContext* deviceContext,
    UINT viewCount, ID3D11RenderTargetView** renderTargetViews,
    ID3D11DepthStencilView* depthStencilView, const D3D11_VIEWPORT&
viewport)
    {
        sRenderTargetStack.push(RenderTargetData(viewCount,
renderTargetViews, depthStencilView, viewport));
        deviceContext->OMSetRenderTargets(viewCount, renderTargetViews,
depthStencilView);
    }
}
```

```
    deviceContext->RSSetViewports(1, &viewport);
}

void RenderTarget::End(ID3D11DeviceContext* deviceContext)
{
    sRenderTargetStack.pop();

    if (sRenderTargetStack.size() > 0)
    {
        RenderTargetData renderTargetData = sRenderTargetStack.
top();
        deviceContext->OMSetRenderTargets(renderTargetData.
ViewCount, renderTargetData.RenderTargetViews, renderTargetData.
DepthStencilView);
        deviceContext->RSSetViewports(1, &renderTargetData.
Viewport);
    }
    else
    {
        static ID3D11RenderTargetView* nullRenderTargetView =
nullptr;
        deviceContext->OMSetRenderTargets(1, &nullRenderTargetView,
nullptr);
    }
}
```

---

```
mDepthMap->Begin();

ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_
TRIANGLELIST);

direct3DDeviceContext->ClearDepthStencilView(mDepthMap
->DepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
Pass* pass = mDepthMapMaterial->CurrentTechnique()->Passes().at(0);
ID3D11InputLayout* inputLayout = mDepthMapMaterial->InputLayouts().
at(pass);
direct3DDeviceContext->IASetInputLayout(inputLayout);

UINT stride = mDepthMapMaterial->VertexSize();
UINT offset = 0;
direct3DDeviceContext->IASetVertexBuffers(0, 1,
&mModelPositionVertexBuffer, &stride, &offset);
direct3DDeviceContext->IASetIndexBuffer(mModelIndexBuffer, DXGI_FORMAT_
R32_UINT, 0);

XMMATRIX modelWorldMatrix = XMLoadFloat4x4(&mModelWorldMatrix);
mDepthMapMaterial->WorldLightViewProjection() << modelWorldMatrix *
mProjector->ViewMatrix() * mProjector->ProjectionMatrix();

pass->Apply(0, direct3DDeviceContext);
direct3DDeviceContext->DrawIndexed(mModelIndexCount, 0, 0);

mDepthMap->End();
```

```
cbuffer CBufferPerObject
{
    float4x4 WorldLightViewProjection;
}

float4 create_depthmap_vertex_shader(float4 ObjectPosition : POSITION)
: SV_Position
{
    return mul(ObjectPosition, WorldLightViewProjection);
}

technique1 create_depthmap
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
create_depthmap_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(NULL);
    }
}
```

---

```
cbuffer CBufferPerFrame
{
    /* ... */
    float DepthBias = 0.005;
}

Texture2D ColorTexture;
Texture2D ProjectedTexture;
Texture2D DepthMap;

SamplerState DepthMapSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
};

float4 project_texture_w_depthmap_pixel_shader(VS_OUTPUT IN) :
SV_Target
{
    float4 OUT = (float4)0;
```

```
float3 lightDirection = LightPosition - IN.WorldPosition;
lightDirection = normalize(lightDirection);

float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

float3 normal = normalize(IN.Normal);
float n_dot_l = dot(normal, lightDirection);
float3 halfVector = normalize(lightDirection + viewDirection);
float n_dot_h = dot(normal, halfVector);

float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

float3 ambient = get_vector_color_contribution(AmbientColor, color.rgb);

float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

OUT.rgb = ambient + diffuse + specular;
OUT.a = 1.0f;

if (IN.ProjectedTextureCoordinate.w >= 0.0f)
{
    IN.ProjectedTextureCoordinate.xyz /= IN.ProjectedTexture
Coordinate.w;
    float pixelDepth = IN.ProjectedTextureCoordinate.z;
    float sampledDepth = DepthMap.Sample(DepthMapSampler,
IN.ProjectedTextureCoordinate.xy).x + DepthBias;

    float3 projectedColor = (pixelDepth > sampledDepth ?
ColorWhite : ProjectedTexture.Sample(ProjectedTextureSampler,
IN.ProjectedTextureCoordinate.xy).rgb);
    OUT.rgb *= projectedColor;
}

return OUT;
}

/**************** Techniques *****/

```

```
technique11 project_texture_w_depthmap
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
project_texture_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0,
project_texture_w_depthmap_pixel_shader()));

        SetRasterizerState(BackFaceCulling);
    }
}
```

---

```
#include "include\\Common.fxh"

/********************* Resources *****/
static const float4 ColorWhite = { 1, 1, 1, 1 };
static const float3 ColorBlack = { 0, 0, 0 };
static const float DepthBias = 0.005;

cbuffer CBufferPerFrame
{
    float4 AmbientColor = { 1.0f, 1.0f, 1.0f, 0.0f };
    float4 LightColor = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 LightPosition = { 0.0f, 0.0f, 0.0f };
    float LightRadius = 10.0f;
    float3 CameraPosition;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
    float4 SpecularColor : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
    float SpecularPower : SPECULARPOWER = 25.0f;

    float4x4 ProjectiveTextureMatrix;
}
```

```
Texture2D ColorTexture;
Texture2D ShadowMap;

SamplerState ShadowMapSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
};

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

RasterizerState BackFaceCulling
{
    CullMode = BACK;
};

/***************** Data Structures *******/

struct VS_INPUT
{
```

```
float4 ObjectPosition : POSITION;
float2 TextureCoordinate : TEXCOORD;
float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
    float4 ShadowTextureCoordinate : TEXCOORD3;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.WorldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.TextureCoordinate = IN.TextureCoordinate;
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
```

```
    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));

    OUT.ShadowTextureCoordinate = mul(IN.ObjectPosition,
ProjectiveTextureMatrix);

    return OUT;
}

/***************** Pixel Shader *****/
float4 shadow_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);
```

```
    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    if (IN.ShadowTextureCoordinate.w >= 0.0f)
    {
        IN.ShadowTextureCoordinate.xyz /= IN.ShadowTextureCoordinate.w;
        float pixelDepth = IN.ShadowTextureCoordinate.z;
        float sampledDepth = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy).x + DepthBias;

        // Shadow applied in a boolean fashion -- either in shadow or not
        float3 shadow = (pixelDepth > sampledDepth ? ColorBlack :
ColorWhite.rgb);

        diffuse *= shadow;
        specular *= shadow;
    }

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

/**************** Techniques *****/
technique11 shadow_mapping
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, shadow_pixel_shader()));

        SetRasterizerState(BackFaceCulling);
    }
}
```

```
cbuffer CBufferPerFrame
{
    /* ... */
    float2 ShadowMapSize = { 1024.0f, 1024.0f };
}

float4 shadow_manual_pcf_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);
```

```
    float3 ambient = get_vector_color_contribution(AmbientColor, color.rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    if (IN.ShadowTextureCoordinate.w >= 0.0f)
    {
        IN.ShadowTextureCoordinate.xyz /= IN.ShadowTextureCoordinate.w;

        float2 texelSize = 1.0f / ShadowMapSize;
        float sampledDepth1 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy).x + DepthBias;
        float sampledDepth2 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy + float2(texelSize.x, 0)).x + DepthBias;
        float sampledDepth3 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy + float2(0, texelSize.y)).x + DepthBias;
        float sampledDepth4 = ShadowMap.Sample(ShadowMapSampler,
IN.ShadowTextureCoordinate.xy + float2(texelSize.x, texelSize.y)).x +
DepthBias;

        float pixelDepth = IN.ShadowTextureCoordinate.z;
        float shadowFactor1 = (pixelDepth > sampledDepth1 ? 0.0f :
1.0f);
        float shadowFactor2 = (pixelDepth > sampledDepth2 ? 0.0f :
1.0f);
        float shadowFactor3 = (pixelDepth > sampledDepth3 ? 0.0f :
1.0f);
        float shadowFactor4 = (pixelDepth > sampledDepth4 ? 0.0f :
1.0f);

        float2 lerpValues = frac(IN.ShadowTextureCoordinate.xy *
ShadowMapSize);
        float shadow = lerp(lerp(shadowFactor1, shadowFactor2,
lerpValues.x), lerp(shadowFactor3, shadowFactor4, lerpValues.x),
lerpValues.y);
        diffuse *= shadow;
        specular *= shadow;
    }

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}
```

```
SamplerComparisonState PcfShadowMapSampler
{
    Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = ColorWhite;
    ComparisonFunc = LESS_EQUAL;
};

float4 shadow_pcf_pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);
```

```
    float4 color = ColorTexture.Sample(ColorSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    IN.ShadowTextureCoordinate.xyz /= IN.ShadowTextureCoordinate.w;
    float pixelDepth = IN.ShadowTextureCoordinate.z;

    float shadow = ShadowMap.SampleCmpLevelZero(PcfShadowMapSampler,
IN.ShadowTextureCoordinate.xy, pixelDepth).x;
    diffuse *= shadow;
    specular *= shadow;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}
```

---

```
RasterizerState DepthBias
{
    DepthBias = 84000;
};

float4 create_depthmap_vertex_shader(float4 ObjectPosition : POSITION)
: SV_Position
{
    return mul(ObjectPosition, WorldLightViewProjection);
}

technique11 create_depthmap_w_bias
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0,
create_depthmap_vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(NULL);

        SetRasterizerState(DepthBias);
    }
}
```

---

```
#include "include\\Common.fch"

#define MaxBones 60

/********************* Resources *****/
cbuffer CBufferPerFrame
{
    float4 AmbientColor = { 1.0f, 1.0f, 1.0f, 0.0f };
    float4 LightColor = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 LightPosition = { 0.0f, 0.0f, 0.0f };
    float LightRadius = 10.0f;
    float3 CameraPosition;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
    float4 SpecularColor : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
    float SpecularPower : SPECULARPOWER = 25.0f;
}

cbuffer CBufferSkinning
{
    float4x4 BoneTransforms[MaxBones];
}
```

```
Texture2D ColorTexture;

SamplerState ColorSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

/***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
    uint4 BoneIndices : BONEINDICES;
    float4 BoneWeights : WEIGHTS;
};
struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 WorldPosition : TEXCOORD1;
    float Attenuation : TEXCOORD2;
};
```

```
***** Vertex Shader *****

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    float4x4 skinTransform = (float4x4)0;
    skinTransform += BoneTransforms[IN.BoneIndices.x] *
IN.BoneWeights.x;
    skinTransform += BoneTransforms[IN.BoneIndices.y] *
IN.BoneWeights.y;
    skinTransform += BoneTransforms[IN.BoneIndices.z] *
IN.BoneWeights.z;
    skinTransform += BoneTransforms[IN.BoneIndices.w] *
IN.BoneWeights.w;

    float4 position = mul(IN.ObjectPosition, skinTransform);
    OUT.Position = mul(position, WorldViewProjection);
    OUT.WorldPosition = mul(position, World).xyz;

    float4 normal = mul(float4(IN.Normal, 0), skinTransform);
    OUT.Normal = normalize(mul(normal, World).xyz);

    OUT.TextureCoordinate = IN.TextureCoordinate;
```

```
    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));

    return OUT;
}

/***************** Pixel Shaders *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(ColorSampler,
```

```
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

/***** Techniques *****/
technique1 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}
```

---

```
float4x4 skinTransform = (float4x4)0;
skinTransform += BoneTransforms[IN.BoneIndices.x] * IN.BoneWeights.x;
skinTransform += BoneTransforms[IN.BoneIndices.y] * IN.BoneWeights.y;
skinTransform += BoneTransforms[IN.BoneIndices.z] * IN.BoneWeights.z;
skinTransform += BoneTransforms[IN.BoneIndices.w] * IN.BoneWeights.w;
```

```
class SceneNode : public RTTI
{
    RTTI_DECLARATIONS(SceneNode, RTTI)

public:
    const std::string& Name() const;
    SceneNode* Parent();
    std::vector<SceneNode*>& Children();
    const XMFLOAT4X4& Transform() const;
    XMMATRIX TransformMatrix() const;

    void SetParent(SceneNode* parent);
    void SetTransform(XMFLOAT4X4& transform);
    void SetTransform(CXMMATRIX transform);

    SceneNode(const std::string& name);
    SceneNode(const std::string& name, const XMFLOAT4X4& transform);

protected:
    std::string mName;
    SceneNode* mParent;
    std::vector<SceneNode*> mChildren;
    XMFLOAT4X4 mTransform;

private:
    SceneNode();
    SceneNode(const SceneNode& rhs);
    SceneNode& operator=(const SceneNode& rhs);
};
```

---

```
class Bone : public SceneNode
{
    RTTI_DECLARATIONS(Bone, SceneNode)

public:
    UINT Index() const;
    void SetIndex(UINT index);

    const XMFLOAT4X4& OffsetTransform() const;
    XMMATRIX OffsetTransformMatrix() const;

    Bone(const std::string& name, UINT index, const XMFLOAT4X4&
offsetTransform);

private:
    Bone();
    Bone(const Bone& rhs);
    Bone& operator=(const Bone& rhs);

    UINT mIndex;                                // Index into the model's bone
container
    XMFLOAT4X4 mOffsetTransform; // Transforms from mesh space to bone
space
};
```

---

```
std::vector<Bone*> mBones;
std::map<std::string, UINT> mBoneIndexMapping;
SceneNode* mRootNode;
```

```
class BoneVertexWeights
{
public:

    typedef struct _VertexWeight
    {
        float Weight;
        UINT BoneIndex;

        _VertexWeight(float weight, UINT boneIndex)
            : Weight(weight), BoneIndex(boneIndex) { }

    } VertexWeight;

    const std::vector<VertexWeight>& Weights();

    void AddWeight(float weight, UINT boneIndex);

    static const UINT MaxBoneWeightsPerVertex = 4;

private:
    std::vector<VertexWeight> mWeights;
};
```

---

```
if (mesh.HasBones())
{
    mBoneWeights.resize(mesh.mNumVertices);

    for (UINT i = 0; i < mesh.mNumBones; i++)
    {
        aiBone* meshBone = mesh.mBones[i];

        // Look up the bone in the model's hierarchy, or add it if not
        found.

        UINT boneIndex = 0U;
        std::string boneName = meshBone->mName.C_Str();
        auto boneMappingIterator = mModel.mBoneIndexMapping.
        find(boneName);
        if (boneMappingIterator != mModel.mBoneIndexMapping.end())
        {
            boneIndex = boneMappingIterator->second;
        }
        else
        {
            boneIndex = mModel.mBones.size();

            XMATRIX offsetMatrix = XMLoadFloat4x4(&(XMFLOAT4X4
(reinterpret_cast<const float*>(meshBone->mOffsetMatrix[0]))));
            XMFLOAT4X4 offset;
            XMStoreFloat4x4(&offset, XMMatrixTranspose(offsetMatrix));

            Bone* modelBone = new Bone(boneName, boneIndex, offset);
            mModel.mBones.push_back(modelBone);
            mModel.mBoneIndexMapping[boneName] = boneIndex;
        }

        for (UINT i = 0; i < meshBone->mNumWeights; i++)
        {
            aiVertexWeight vertexWeight = meshBone->mWeights[i];
            mBoneWeights[vertexWeight.mVertexId].
            AddWeight(vertexWeight.mWeight, boneIndex);
        }
    }
}
```

```
SceneNode* Model::BuildSkeleton(aiNode& node, SceneNode* parentSceneNode)
{
    SceneNode* sceneNode = nullptr;

    auto boneMapping = mBoneIndexMapping.find(node.mName.C_Str());
    if (boneMapping == mBoneIndexMapping.end())
    {
        sceneNode = new SceneNode(node.mName.C_Str());
    }
    else
    {
        sceneNode = mBones[boneMapping->second];
    }

    XMATRIX transform = XMLoadFloat4x4(&(XMFLOAT4X4(reinterpret_cast<const float*>(node.mTransformation[0]))));
    sceneNode->SetTransform(XMMatrixTranspose(transform));
    sceneNode->SetParent(parentSceneNode);

    for (UINT i = 0; i < node.mNumChildren; i++)
    {
        SceneNode* childSceneNode = BuildSkeleton(*(node.mChildren[i]),
                                                sceneNode);
        sceneNode->Children().push_back(childSceneNode);
    }

    return sceneNode;
}
```

---

```
class AnimationClip
{
    friend class Model;

public:
    ~AnimationClip();

    const std::string& Name() const;
    float Duration() const;
    float TicksPerSecond() const;
    const std::vector<BoneAnimation*>& BoneAnimations() const;
    const std::map<Bone*, BoneAnimation*>& BoneAnimationsByBone() const;
    const UINT KeyframeCount() const;

    UINT GetTransform(float time, Bone& bone, XMFLOAT4X4& transform) const;
    void GetTransforms(float time, std::vector<XMFLOAT4X4>& boneTransforms) const;

    void GetTransformAtKeyframe(UINT keyframe, Bone& bone, XMFLOAT4X4& transform) const;
    void GetTransformsAtKeyframe(UINT keyframe, std::vector<XMFLOAT4X4>& boneTransforms) const;

    void GetInterpolatedTransform(float time, Bone& bone, XMFLOAT4X4& transform) const;
    void GetInterpolatedTransforms(float time, std::vector<XMFLOAT4X4>& boneTransforms) const;

private:
    AnimationClip(Model& model, aiAnimation& animation);

    AnimationClip();
    AnimationClip(const AnimationClip& rhs);
    AnimationClip& operator=(const AnimationClip& rhs);

    std::string mName;
    float mDuration;
    float mTicksPerSecond;
    std::vector<BoneAnimation*> mBoneAnimations;
    std::map<Bone*, BoneAnimation*> mBoneAnimationsByBone;
    UINT mKeyframeCount;
};
```

```
AnimationClip::AnimationClip(Model& model, aiAnimation& animation)
    : mName(animation.mName.C_Str()),
      mDuration(static_cast<float>(animation.mDuration)),
      mTicksPerSecond(static_cast<float>(animation.mTicksPerSecond)),
      mBoneAnimations(), mBoneAnimationsByBone(), mKeyframeCount(0)
{
    assert(animation.mNumChannels > 0);

    if (mTicksPerSecond <= 0.0f)
    {
        mTicksPerSecond = 1.0f;
    }

    for (UINT i = 0; i < animation.mNumChannels; i++)
    {
        BoneAnimation* boneAnimation = new BoneAnimation(model,
*(animation.mChannels[i]));
        mBoneAnimations.push_back(boneAnimation);

        assert(mBoneAnimationsByBone.find(&(boneAnimation->GetBone())))
== mBoneAnimationsByBone.end();
        mBoneAnimationsByBone[&(boneAnimation->GetBone())] =
boneAnimation;
    }

    for (BoneAnimation* boneAnimation : mBoneAnimations)
    {
        if (boneAnimation->Keyframes().size() > mKeyframeCount)
        {
            mKeyframeCount = boneAnimation->Keyframes().size();
        }
    }
}
```

```
class BoneAnimation
{
    friend class AnimationClip;

public:
    ~BoneAnimation();

    Bone& GetBone();
    const std::vector<Keyframe*> Keyframes() const;

    UINT GetTransform(float time, XMFLOAT4X4& transform) const;
    void GetTransformAtKeyframe(UINT keyframeIndex, XMFLOAT4X4&
transform) const;
    void GetInterpolatedTransform(float time, XMFLOAT4X4& transform)
const;

private:
    BoneAnimation(Model& model, aiNodeAnim& nodeAnim);

    BoneAnimation();
    BoneAnimation(const BoneAnimation& rhs);
    BoneAnimation& operator=(const BoneAnimation& rhs);

    UINT FindKeyframeIndex(float time) const;

    Model* mModel;
    Bone* mBone;
    std::vector<Keyframe*> mKeyframes;
};
```

---

```
BoneAnimation::BoneAnimation(Model& model, aiNodeAnim& nodeAnim)
    : mModel(&model), mBone(nullptr), mKeyframes()
{
    UINT boneIndex = model.BoneIndexMapping().at(nodeAnim.
mNodeName.C_Str());
    mBone = model.Bones().at(boneIndex);

    assert(nodeAnim.mNumPositionKeys == nodeAnim.mNumRotationKeys);
    assert(nodeAnim.mNumPositionKeys == nodeAnim.mNumScalingKeys);

    for (UINT i = 0; i < nodeAnim.mNumPositionKeys; i++)
    {
        aiVectorKey positionKey = nodeAnim.mPositionKeys[i];
        aiQuatKey rotationKey = nodeAnim.mRotationKeys[i];
        aiVectorKey scaleKey = nodeAnim.mScalingKeys[i];

        assert(positionKey.mTime == rotationKey.mTime);
        assert(positionKey.mTime == scaleKey.mTime);

        Keyframe* keyframe = new Keyframe(static_
cast<float>(positionKey.mTime), XMFLOAT3(positionKey.mValue.x,
positionKey.mValue.y, positionKey.mValue.z), XMFLOAT4(rotationKey.
mValue.x, rotationKey.mValue.y, rotationKey.mValue.z, rotationKey.
mValue.w), XMFLOAT3(scaleKey.mValue.x, scaleKey.mValue.y, scaleKey.
mValue.z));
        mKeyframes.push_back(keyframe);
    }
}
```

---

```
class Keyframe
{
    friend class BoneAnimation;

public:
    float Time() const;
    const XMFLOAT3& Translation() const;
    const XMFLOAT4& RotationQuaternion() const;
    const XMFLOAT3& Scale() const;

    XMVECTOR TranslationVector() const;
    XMVECTOR RotationQuaternionVector() const;
    XMVECTOR ScaleVector() const;

    XMMATRIX Transform() const;

private:
    Keyframe(float time, const XMFLOAT3& translation, const XMFLOAT4&
rotationQuaternion, const XMFLOAT3& scale);

    Keyframe();
    Keyframe(const Keyframe& rhs);
    Keyframe& operator=(const Keyframe& rhs);

    float mTime;
    XMFLOAT3 mTranslation;
    XMFLOAT4 mRotationQuaternion;
    XMFLOAT3 mScale;
};


```

---



```
UINT BoneAnimation::GetTransform(float time, XMFLOAT4X4& transform)
const
{
    UINT keyframeIndex = FindKeyframeIndex(time);
    Keyframe* keyframe = mKeyframes[keyframeIndex];

    XMStoreFloat4x4(&transform, keyframe->Transform());
}

void BoneAnimation::GetTransformAtKeyframe(UINT keyframeIndex,
XMFLOAT4X4& transform) const
{
    // Clamp the keyframe
    if (keyframeIndex >= mKeyframes.size() )
    {
        keyframeIndex = mKeyframes.size() - 1;
    }

    Keyframe* keyframe = mKeyframes[keyframeIndex];

    XMStoreFloat4x4(&transform, keyframe->Transform());
}

void BoneAnimation::GetInterpolatedTransform(float time, XMFLOAT4X4&
transform) const
{
```

```
Keyframe* firstKeyframe = mKeyframes.front();
Keyframe* lastKeyframe = mKeyframes.back();

if (time <= firstKeyframe->Time())
{
    // Specified time is before the start time of the animation, so
return the first keyframe
    XMStoreFloat4x4(&transform, firstKeyframe->Transform());
}

else if (time >= lastKeyframe->Time())
{
    // Specified time is after the end time of the animation, so
return the last keyframe
    XMStoreFloat4x4(&transform, lastKeyframe->Transform());
}

else
{
    // Interpolate the transform between keyframes
    UINT keyframeIndex = FindKeyframeIndex(time);

    Keyframe* keyframeOne = mKeyframes[keyframeIndex];
    Keyframe* keyframeTwo = mKeyframes[keyframeIndex + 1];

    XMVECTOR translationOne = keyframeOne->TranslationVector();
    XMVECTOR rotationQuaternionOne =
keyframeOne->RotationQuaternionVector();
    XMVECTOR scaleOne = keyframeOne->ScaleVector();
```

```
XMVECTOR translationTwo = keyframeTwo->TranslationVector();
XMVECTOR rotationQuaternionTwo =
keyframeTwo->RotationQuaternionVector();
XMVECTOR scaleTwo = keyframeTwo->ScaleVector();

float lerpValue = ((time - keyframeOne->Time()) / (keyframeTwo-
>Time() - keyframeOne->Time()));
XMVECTOR translation = XMVectorLerp(translationOne,
translationTwo, lerpValue);
XMVECTOR rotationQuaternion = XMQuaternionSlerp
(rotationQuaternionOne, rotationQuaternionTwo, lerpValue);
XMVECTOR scale = XMVectorLerp(scaleOne, scaleTwo, lerpValue);

XMVECTOR rotationOrigin = XMLoadFloat4(&Vector4Helper::Zero);
XMStoreFloat4x4(&transform, XMMatrixAffineTransformation(scale,
rotationOrigin, rotationQuaternion, translation));
}

}

UINT BoneAnimation::FindKeyframeIndex(float time) const
{
    Keyframe* firstKeyframe = mKeyframes.front();
    if (time <= firstKeyframe->Time())
    {
        return 0;
    }

    Keyframe* lastKeyframe = mKeyframes.back();
    if (time >= lastKeyframe->Time())
    {
        return mKeyframes.size() - 1;
    }

    UINT keyframeIndex = 1;

    for (; keyframeIndex < mKeyframes.size() - 1 && time >=
mKeyframes[keyframeIndex]->Time(); keyframeIndex++);
    return keyframeIndex - 1;
}
```

```
class AnimationPlayer : GameComponent
{
    RTTI_DECLARATIONS(AnimationPlayer, GameComponent)

public:
    AnimationPlayer(Game& game, Model& model, bool interpolationEnabled
= true);

    const Model& GetModel() const;
    const AnimationClip* CurrentClip() const;
    float CurrentTime() const;
    UINT CurrentKeyframe() const;
    const std::vector<XMFLOAT4X4>& BoneTransforms() const;

    bool InterpolationEnabled() const;
    bool IsPlayingClip() const;
    bool IsClipLooped() const;

    void SetInterpolationEnabled(bool interpolationEnabled);

    void StartClip(AnimationClip& clip);
    void PauseClip();
    void ResumeClip();
    virtual void Update(const GameTime& gameTime) override;
    void SetCurrentKeyFrame(UINT keyframe);

private:
    AnimationPlayer();
    AnimationPlayer(const AnimationPlayer& rhs);
    AnimationPlayer& operator=(const AnimationPlayer& rhs);

    void GetBindPose(SceneNode& sceneNode);
    void GetPose(float time, SceneNode& sceneNode);
    void GetPoseAtKeyframe(UINT keyframe, SceneNode& sceneNode);
    void GetInterpolatedPose(float time, SceneNode& sceneNode);

    Model* mModel;
    AnimationClip* mCurrentClip;
    float mCurrentTime;
    UINT mCurrentKeyframe;
    std::map<SceneNode*, XMFLOAT4X4> mToRootTransforms;
    std::vector<XMFLOAT4X4> mFinalTransforms;
    XMFLOAT4X4 mInverseRootTransform;
    bool mInterpolationEnabled;
    bool mIsPlayingClip;
    bool mIsClipLooped;
};

};
```

```
void AnimationPlayer::StartClip(AnimationClip& clip)
{
    mCurrentClip = &clip;
    mCurrentTime = 0.0f;
    mCurrentKeyframe = 0;
    mIsPlayingClip = true;

    XMATRIX inverseRootTransform = XMMatrixInverse
    (&XMMatrixDeterminant(mModel->RootNode()->TransformMatrix()),
     mModel->RootNode()->TransformMatrix());
    XMStoreFloat4x4(&mInverseRootTransform, inverseRootTransform);
    GetBindPose(* (mModel->RootNode()));
}
```

```
void AnimationPlayer::GetBindPose(SceneNode& sceneNode)
{
    XMATRIX toRootTransform = sceneNode.TransformMatrix();

    SceneNode* parentNode = sceneNode.Parent();
    while (parentNode != nullptr)
    {
        toRootTransform = toRootTransform *
parentNode->TransformMatrix();
        parentNode = parentNode->Parent();
    }

    Bone* bone = sceneNode.As<Bone>();
    if (bone != nullptr)
    {
        XMStoreFloat4x4(&(mFinalTransforms[bone->Index()]),
bone->OffsetTransformMatrix() * toRootTransform * XMLoadFloat4x4
(&mInverseRootTransform));
    }

    for (SceneNode* childNode : sceneNode.Children())
    {
        GetBindPose(*childNode);
    }
}
```

---

```
void AnimationPlayer::GetBindPose(SceneNode& sceneNode)
{
    XMATRIX toParentTransform = sceneNode.TransformMatrix();
    XMATRIX toRootTransform = (sceneNode.Parent() != nullptr ?
toParentTransform * XMLoadFloat4x4(&(mToRootTransforms.at(sceneNode.
Parent())))) : toParentTransform);
    XMStoreFloat4x4(&(mToRootTransforms[&sceneNode]), toRootTransform);

    Bone* bone = sceneNode.As<Bone>();
    if (bone != nullptr)
    {
        XMStoreFloat4x4(&(mFinalTransforms[bone->Index()]),
bone->OffsetTransformMatrix() * toRootTransform * XMLoadFloat4x4
(&mInverseRootTransform));
    }

    for (SceneNode* childNode : sceneNode.Children())
    {
        GetBindPose(*childNode);
    }
}
```

---

```
void AnimationPlayer::Update(const GameTime& gameTime)
{
    if (mIsPlayingClip)
    {
        assert(mCurrentClip != nullptr);

        mCurrentTime += static_cast<float>(gameTime.ElapsedGameTime())
        * mCurrentClip->TicksPerSecond();
        if (mCurrentTime >= mCurrentClip->Duration())
        {
            if (mIsClipLooped)
            {
                mCurrentTime = 0.0f;
            }
            else
            {
                mIsPlayingClip = false;
                return;
            }
        }

        if (mInterpolationEnabled)
        {
            GetInterpolatedPose(mCurrentTime, *(mModel->RootNode()));
        }
        else
        {
            GetPose(mCurrentTime, *(mModel->RootNode()));
        }
    }
}
```

```
void AnimationPlayer::GetInterpolatedPose(float time, SceneNode&
sceneNode)
{
    XMFLOAT4X4 toParentTransform;
    Bone* bone = sceneNode.As<Bone>();
    if (bone != nullptr)
    {
        mCurrentClip->GetInterpolatedTransform(time, *bone,
toParentTransform);
    }
    else
    {
        toParentTransform = sceneNode.Transform();
    }

    XMMATRIX toRootTransform = (sceneNode.Parent() != nullptr ?
XMLoadFloat4x4(&toParentTransform) * XMLoadFloat4x4(&
(mToRootTransforms.at(sceneNode.Parent()))) : XMLoadFloat4x4
(&toParentTransform));
    XMStoreFloat4x4(&(mToRootTransforms[&sceneNode]), toRootTransform);

    if (bone != nullptr)
    {
        XMStoreFloat4x4(&(mFinalTransforms[bone->Index()]),
bone->OffsetTransformMatrix() * toRootTransform * XMLoadFloat4x4
(&mInverseRootTransform));
    }

    for (SceneNode* childNode : sceneNode.Children())
    {
        GetInterpolatedPose(time, *childNode);
    }
}
```

---

```
void AnimationPlayer::SetCurrentKeyFrame(UINT keyframe)
{
    mCurrentKeyframe = keyframe;
    GetPoseAtKeyframe(mCurrentKeyframe, * (mModel->RootNode()) );
}
```

```
D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "BONEINDICES", 0, DXGI_FORMAT_R32G32B32A32_UINT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "WEIGHTS", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};
```

```
mMaterial->BoneTransforms() << mAnimationPlayer->BoneTransforms();
```

```
[maxvertexcount(3)]  
void geometry_shader(point VS_OUTPUT IN[1], inout TriangleStream<GS_  
OUTPUT> triStream) { // shader body }
```

```
struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct GS_OUTPUT
{
    float4 Position : SV_Position;
};

[maxvertexcount(3)]
void geometry_shader(point VS_OUTPUT IN[1], inout TriangleStream<GS_OUTPUT> triStream)
{
    GS_OUTPUT OUT = (GS_OUTPUT)0;

    for (int i = 0; i < 3; i++)
    {
        OUT.Position = // Some modification of the input point, followed
by transformation into homogeneous-clip space
        triStream.Append(OUT);
    }
}
```

```
***** Resources *****
```

```
static const float2 QuadUVs[4] = { float2(0.0f, 1.0f), // v0,  
    lower-left  
    float2(0.0f, 0.0f), // v1,  
    upper-left  
    float2(1.0f, 0.0f), // v2,  
    upper-right  
    float2(1.0f, 1.0f) // v3,  
    lower-right  
};
```

```
cbuffer CBufferPerFrame  
{  
    float3 CameraPosition : CAMERAPOSITION;  
    float3 CameraUp;  
}
```

```
cbuffer CBufferPerObject  
{  
    float4x4 ViewProjection;  
}
```

```
Texture2D ColorTexture;
```

```
SamplerState ColorSampler  
{  
    Filter = MIN_MAG_MIP_LINEAR;  
    AddressU = WRAP;  
    AddressV = WRAP;  
};
```

```
***** Data Structures *****
```

```
struct VS_INPUT  
{  
    float4 Position : POSITION;  
    float2 Size : SIZE;  
};
```

```
struct VS_OUTPUT  
{  
    float4 Position : POSITION;  
    float2 Size : SIZE;  
};
```

```
struct GS_OUTPUT
{
    float4 Position : SV_Position;
    float2 TextureCoordinate : TEXCOORD;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = IN.Position;
    OUT.Size = IN.Size;

    return OUT;
}

/***** Geometry Shader *****/
[maxvertexcount(6)]
void geometry_shader(point VS_OUTPUT IN[1], inout TriangleStream<GS_OUTPUT> triStream)
{
    GS_OUTPUT OUT = (GS_OUTPUT)0;

    float2 halfSize = IN[0].Size / 2.0f;
    float3 direction = CameraPosition - IN[0].Position.xyz;
    float3 right = cross(normalize(direction), CameraUp);

    float3 offsetX = halfSize.x * right;
    float3 offsetY = halfSize.y * CameraUp;

    float4 vertices[4];
    vertices[0] = float4(IN[0].Position.xyz + offsetX - offsetY, 1.0f);
// lower-left
    vertices[1] = float4(IN[0].Position.xyz + offsetX + offsetY, 1.0f);
// upper-left
    vertices[2] = float4(IN[0].Position.xyz - offsetX + offsetY, 1.0f);
// upper-right
    vertices[3] = float4(IN[0].Position.xyz - offsetX - offsetY, 1.0f);
// lower-right

    // tri: 0, 1, 2
    OUT.Position = mul(vertices[0], ViewProjection);
```

```
OUT.TextureCoordinate = QuadUVs[0];
triStream.Append(OUT);

OUT.Position = mul(vertices[1], ViewProjection);
OUT.TextureCoordinate = QuadUVs[1];
triStream.Append(OUT);

OUT.Position = mul(vertices[2], ViewProjection);
OUT.TextureCoordinate = QuadUVs[2];
triStream.Append(OUT);
triStream.RestartStrip();

// tri: 0, 2, 3
OUT.Position = mul(vertices[0], ViewProjection);
OUT.TextureCoordinate = QuadUVs[0];
triStream.Append(OUT);

OUT.Position = mul(vertices[2], ViewProjection);
OUT.TextureCoordinate = QuadUVs[2];
triStream.Append(OUT);

OUT.Position = mul(vertices[3], ViewProjection);
OUT.TextureCoordinate = QuadUVs[3];
triStream.Append(OUT);
}

/****************** Pixel Shader *****/
float4 pixel_shader(GS_OUTPUT IN) : SV_Target
{
    return ColorTexture.Sample(ColorSampler, IN.TextureCoordinate);
}

/****************** Techniques *****/
technique1 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(CompileShader(gs_5_0, geometry_shader()));
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}
```

---

```
static const float2 QuadStripUVs[4] = { float2(0.0f, 1.0f), // v0,
                                         lower-left
                                         float2(0.0f, 0.0f), // v1,
                                         upper-left
                                         float2(1.0f, 1.0f), // v2,
                                         lower-right
                                         float2(1.0f, 0.0f) // v3,
                                         upper-right
                                       };
```

[maxvertexcount(4)]

```
void geometry_shader(point VS_OUTPUT IN[1], inout TriangleStream<GS_OUTPUT> triStream)
{
    GS_OUTPUT OUT = (GS_OUTPUT)0;

    float2 halfSize = IN[0].Size / 2.0f;
    float3 direction = CameraPosition - IN[0].Position.xyz;
    float3 right = cross(normalize(direction), CameraUp);

    float3 offsetX = halfSize.x * right;
    float3 offsetY = halfSize.y * CameraUp;
    float4 vertices[4];
    vertices[0] = float4(IN[0].Position.xyz + offsetX - offsetY, 1.0f);
// lower-left
    vertices[1] = float4(IN[0].Position.xyz + offsetX + offsetY, 1.0f);
// upper-left
    vertices[2] = float4(IN[0].Position.xyz - offsetX - offsetY, 1.0f);
// lower-right
    vertices[3] = float4(IN[0].Position.xyz - offsetX + offsetY, 1.0f);
// upper-right

    [unroll]
    for (int i = 0; i < 4; i++)
    {
        OUT.Position = mul(vertices[i], ViewProjection);
        OUT.TextureCoordinate = QuadStripUVs[i];

        triStream.Append(OUT);
    }
}
```

```
void GeometryShaderDemo::Initialize()
{
    SetCurrentDirectory(Utility::ExecutableDirectory().c_str());

    // Initialize the material
    mEffect = new Effect(*mGame);
    mEffect->LoadCompiledEffect(L"Content\\Effects\\PointSprite.cso");
    mMaterial = new PointSpriteMaterial();
    mMaterial->Initialize(*mEffect);

    mPass = mMaterial->CurrentTechnique()->Passes().at(0);
    mInputLayout = mMaterial->InputLayouts().at(mPass);

    UINT maxRandomPoints = 100;
    float maxDistance = 10;
    float minSize = 2;
    float maxSize = 2;

    std::random_device randomDevice;
    std::default_random_engine randomGenerator(randomDevice());
    std::uniform_real_distribution<float> distanceDistribution
(-maxDistance, maxDistance);
    std::uniform_real_distribution<float> sizeDistribution(minSize,
maxSize);

    // Randomly generate points
    std::vector<VertexPositionSize> vertices;
    vertices.reserve(maxRandomPoints);
    for (UINT i = 0; i < maxRandomPoints; i++)
    {
        float x = distanceDistribution(randomGenerator);
        float y = distanceDistribution(randomGenerator);
        float z = distanceDistribution(randomGenerator);

        float size = sizeDistribution(randomGenerator);

        vertices.push_back(VertexPositionSize(XMFLOAT4(x, y, z, 1.0f),
XMFLOAT2(size, size)));
    }

    mVertexCount = vertices.size();
    mMaterial->CreateVertexBuffer(mGame->Direct3DDevice(),
&vertices[0], mVertexCount, &mVertexBuffer);

    std::wstring textureName = L"Content\\Textures\\BookCover.png";
    HRESULT hr = DirectX::CreateWICTextureFromFile(mGame->
Direct3DDevice(), mGame->Direct3DDeviceContext(), textureName.c_str(),
nullptr, &mColorTexture);
```

```
if (FAILED(hr))
{
    throw GameException("CreateWICTextureFromFile() failed.", hr);
}

void GeometryShaderDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_POINTLIST);
    direct3DDeviceContext->IASetInputLayout(mInputLayout);

    UINT stride = mMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mVertexBuffer,
&stride, &offset);

    mMaterial->ViewProjection() << mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mMaterial->CameraPosition() << mCamera->PositionVector();
    mMaterial->CameraUp() << mCamera->UpVector();
    mMaterial->ColorTexture() << mColorTexture;

    mPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->Draw(mVertexCount, 0);

    direct3DDeviceContext->GSSetShader(nullptr, nullptr, 0);
}
```

---

```
[maxvertexcount(4)]  
void geometry_shader_nosize(point VS_NOSIZE_OUTPUT IN[1], uint  
primitiveID : SV_PrimitiveID, inout TriangleStream<GS_OUTPUT>  
triStream)  
{  
    GS_OUTPUT OUT = (GS_OUTPUT)0;  
  
    float size = primitiveID + 1;  
  
    float2 halfSize = size / 2.0f;  
  
    // Remaining code identical to previous iteration, and removed for  
    brevity  
}
```

---

```
struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[3] : SV_TessFactor;
    float InsideFactor : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("constant_hull_shader")]
HS_OUTPUT hull_shader(InputPatch<VS_OUTPUT, 3> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;

    return OUT;
}
```

---

```
cbuffer CBufferPerFrame
{
    float TessellationEdgeFactors[3];
    float TessellationInsideFactor;
}

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[3] : SV_TessFactor;
    float InsideFactor : SV_InsideTessFactor;
};

HS_CONSTANT_OUTPUT constant_hull_shader(InputPatch<VS_OUTPUT, 3> patch)
{
    HS_CONSTANT_OUTPUT OUT = (HS_CONSTANT_OUTPUT)0;

    [unroll]
    for (int i = 0; i < 3; i++)
    {
        OUT.EdgeFactors[i] = TessellationEdgeFactors[i];
    }

    OUT.InsideFactor = TessellationInsideFactor;

    return OUT;
}
```

---

```
struct DS_OUTPUT
{
    float4 Position : SV_Position;
};

[domain("tri")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float3 uvw : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 3> patch)
{
    DS_OUTPUT OUT = (DS_OUTPUT)0;

    float3 objectPosition = uvw.x * patch[0].ObjectPosition.xyz + uvw.y
* patch[1].ObjectPosition.xyz + uvw.z * patch[2].ObjectPosition.xyz;

    OUT.Position = mul(float4(objectPosition, 1.0f),
WorldViewProjection);

    return OUT;
}
```

---

```
***** Resources *****/
static const float4 ColorWheat = { 0.961f, 0.871f, 0.702f, 1.0f };

cbuffer CBufferPerFrame
{
    float TessellationEdgeFactors[4];
    float TessellationInsideFactors[2];
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection;
}

***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
};

struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[4] : SV_TessFactor;
    float InsideFactors[2] : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
    float4 ObjectPosition : POSITION;
};

struct DS_OUTPUT
{
    float4 Position : SV_Position;
};

***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
```

```

VS_OUTPUT OUT = (VS_OUTPUT)0;
OUT.ObjectPosition = IN.ObjectPosition;

return OUT;
}

/***** Hull Shaders *****/
HS_CONSTANT_OUTPUT constant_hull_shader(InputPatch<VS_OUTPUT, 4> patch,
uint patchID : SV_PrimitiveID)
{
    HS_CONSTANT_OUTPUT OUT = (HS_CONSTANT_OUTPUT)0;

    [unroll]
    for (int i = 0; i < 4; i++)
    {
        OUT.EdgeFactors[i] = TessellationEdgeFactors[i];
    }

    OUT.InsideFactors[0] = TessellationInsideFactors[0];
    OUT.InsideFactors[1] = TessellationInsideFactors[1];

    return OUT;
}

[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("constant_hull_shader")]
HS_OUTPUT hull_shader(InputPatch<VS_OUTPUT, 4> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;

    return OUT;
}

/***** Domain Shader *****/
[domain("quad")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float2 uv : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 4> patch)
{

```

```
    DS_OUTPUT OUT;

    float4 v0 = lerp(patch[0].ObjectPosition, patch[1].ObjectPosition,
uv.x);
    float4 v1 = lerp(patch[2].ObjectPosition, patch[3].ObjectPosition,
uv.x);
    float4 objectPosition = lerp(v0, v1, uv.y);

    OUT.Position = mul(float4(objectPosition.xyz, 1.0f),
WorldViewProjection);

    return OUT;
}

/****************** Pixel Shader *****/
float4 pixel_shader(DS_OUTPUT IN) : SV_Target
{
    return ColorWheat;
}

/****************** Techniques *****/
technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetHullShader(CompileShader(hs_5_0, hull_shader()));
        SetDomainShader(CompileShader(ds_5_0, domain_shader()));
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}
```

---

```
ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
direct3DDeviceContext->RSSetState(RasterizerStates::Wireframe);

if (mShowQuadTopology)
{
    direct3DDeviceContext->IASetInputLayout(mQuadInputLayout);
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_4_CONTROL_POINT_PATCHLIST);

    UINT stride = mQuadMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mQuadVertexBuffer,
&stride, &offset);

    mQuadMaterial->WorldViewProjection() << mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mQuadMaterial->TessellationEdgeFactors()
<< mTessellationEdgeFactors;
    mQuadMaterial->TessellationInsideFactors()
<< mTessellationInsideFactors;
    mQuadPass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->Draw(4, 0);
}

else
{
    direct3DDeviceContext->IASetInputLayout(mTriInputLayout);
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_3_CONTROL_POINT_PATCHLIST);

    UINT stride = mTriMaterial->VertexSize();
    UINT offset = 0;
    direct3DDeviceContext->IASetVertexBuffers(0, 1, &mTriVertexBuffer,
&stride, &offset);

    std::vector<float> tessellationEdgeFactors(mTessellationEdge
Factors.begin(), mTessellationEdgeFactors.end() - 1);

    mTriMaterial->WorldViewProjection() << mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mTriMaterial->TessellationEdgeFactors() << tessellationEdgeFactors;
```

```
mTriMaterial->TessellationInsideFactor()
<< mTessellationInsideFactors[0];
mTriPass->Apply(0, direct3DDeviceContext);

direct3DDeviceContext->Draw(3, 0);
}
```

---

```
***** Resources *****/
static const float4 ColorWheat = { 0.961f, 0.871f, 0.702f, 1.0f };

cbuffer CBufferPerFrame
{
    float TessellationEdgeFactors[4];
    float TessellationInsideFactors[2];
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection;
    float4x4 TextureMatrix;

    float DisplacementScale;
}

Texture2D Heightmap;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

***** Data Structures *****/
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate: TEXCOORD;
};

struct VS_OUTPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate: TEXCOORD;
};

struct HS_CONSTANT_OUTPUT
{
    float EdgeFactors[4] : SV_TessFactor;
    float InsideFactors[2] : SV_InsideTessFactor;
};
```

```

struct HS_OUTPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate: TEXCOORD;
};

struct DS_OUTPUT
{
    float4 Position : SV_Position;
};

/***** Vertex Shader *****/
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.ObjectPosition = IN.ObjectPosition;
    OUT.TextureCoordinate = mul(float4(IN.TextureCoordinate, 0, 1),
TextureMatrix);

    return OUT;
}

/***** Hull Shaders *****/
HS_CONSTANT_OUTPUT constant_hull_shader(InputPatch<VS_OUTPUT, 4> patch,
uint patchID : SV_PrimitiveID)
{
    HS_CONSTANT_OUTPUT OUT = (HS_CONSTANT_OUTPUT)0;

    [unroll]
    for (int i = 0; i < 4; i++)
    {
        OUT.EdgeFactors[i] = TessellationEdgeFactors[i];
    }

    OUT.InsideFactors[0] = TessellationInsideFactors[0];
    OUT.InsideFactors[1] = TessellationInsideFactors[1];

    return OUT;
}

[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]

```

```
[outputcontrolpoints(4)]
[patchconstantfunc("constant_hull_shader")]
HS_OUTPUT hull_shader(InputPatch<VS_OUTPUT, 4> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;
    OUT.TextureCoordinate = patch[controlPointID].TextureCoordinate;

    return OUT;
}

/***** Domain Shader *****/
[domain("quad")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float2 uv : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 4> patch)
{
    DS_OUTPUT OUT;

    float4 v0 = lerp(patch[0].ObjectPosition, patch[1].ObjectPosition,
uv.x);
    float4 v1 = lerp(patch[2].ObjectPosition, patch[3].ObjectPosition,
uv.x);
    float4 objectPosition = lerp(v0, v1, uv.y);

    float2 texCoord0 = lerp(patch[0].TextureCoordinate, patch[1].
TextureCoordinate, uv.x);
    float2 texCoord1 = lerp(patch[2].TextureCoordinate, patch[3].
TextureCoordinate, uv.x);
    float2 textureCoordinate = lerp(texCoord0, texCoord1, uv.y);

    objectPosition.y = (2 * Heightmap.SampleLevel(TrilinearSampler,
textureCoordinate, 0).x - 1) * DisplacementScale;

    OUT.Position = mul(float4(objectPosition.xyz, 1.0f),
WorldViewProjection);

    return OUT;
}

/***** Pixel Shader *****/
float4 pixel_shader(DS_OUTPUT IN) : SV_Target
```

```
{  
    return ColorWheat;  
}  
  
/************* Techniques *****/  
  
technique11 main11  
{  
    pass p0  
    {  
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));  
        SetHullShader(CompileShader(hs_5_0, hull_shader()));  
        SetDomainShader(CompileShader(ds_5_0, domain_shader()));  
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));  
    }  
}
```

---

```
***** Resources *****
```

```
cbuffer CBufferPerFrame
{
    float3 CameraPosition : CAMERAPOSITION;
    int MaxTessellationFactor = 64;
    float MinTessellationDistance = 2.0f;
    float MaxTessellationDistance = 20.0f;
}

cbuffer CBufferPerObject
{
    float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
    float4x4 World : WORLD;
}
```

```
***** Hull Shaders *****
```

```
HS_CONSTANT_OUTPUT distance_constant_hull_shader(InputPatch<VS_OUTPUT,
3> patch, uint patchID : SV_PrimitiveID)
{
    HS_CONSTANT_OUTPUT OUT = (HS_CONSTANT_OUTPUT)0;

    // Calculate the center of the patch
    float3 objectCenter = (patch[0].ObjectPosition.xyz + patch[1].
ObjectPosition.xyz + patch[2].ObjectPosition.xyz) / 3.0f;
    float3 worldCenter = mul(float4(objectCenter, 1.0f), World).xyz;

    // Calculate uniform tessellation factor based on distance from the
camera
    float tessellationFactor = max(min(MaxTessellationFactor,
(MaxTessellationDistance - distance(worldCenter, CameraPosition)) /
(MaxTessellationDistance - MinTessellationDistance) *
MaxTessellationFactor), 1);
    [unroll]
    for (int i = 0; i < 3; i++)
    {
        OUT.EdgeFactors[i] = tessellationFactor;
    }

    OUT.InsideFactor = tessellationFactor;

    return OUT;
}
```

```
[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("distance_constant_hull_shader")]
HS_OUTPUT distance_hull_shader(InputPatch<VS_OUTPUT, 3> patch, uint
controlPointID : SV_OutputControlPointID)
{
    HS_OUTPUT OUT = (HS_OUTPUT)0;

    OUT.ObjectPosition = patch[controlPointID].ObjectPosition;
    OUT.TextureCoordinate = patch[controlPointID].TextureCoordinate;

    return OUT;
}

/***** Domain Shader *****/
[domain("tri")]
DS_OUTPUT domain_shader(HS_CONSTANT_OUTPUT IN, float3 uvw : SV_
DomainLocation, const OutputPatch<HS_OUTPUT, 3> patch)
{
    DS_OUTPUT OUT = (DS_OUTPUT)0;

    float3 objectPosition = uvw.x * patch[0].ObjectPosition.xyz + uvw.y
* patch[1].ObjectPosition.xyz + uvw.z * patch[2].ObjectPosition.xyz;
    float2 textureCoordinate = uvw.x * patch[0].TextureCoordinate
+ uvw.y * patch[1].TextureCoordinate + uvw.z * patch[2].
TextureCoordinate;

    OUT.Position = mul(float4(objectPosition, 1.0f),
WorldViewProjection);
    OUT.TextureCoordinate = textureCoordinate;

    return OUT;
}
```

---

/Fc"\$(OutDir)Content\Effects\%(Filename).asm"

```
PixelShader = asm {
    ps_5_0
    dcl_globalFlags refactoringAllowed
    dcl_constantbuffer cb0[2], immediateIndexed
    dcl_constantbuffer cb1[10], immediateIndexed
    dcl_sampler s0, mode_default
    dcl_resource_texture2d (float,float,float,float) t0
    dcl_input_ps linear v1.xyz
    dcl_input_ps linear v2.xy
    dcl_input_ps linear v3.xyz
    dcl_input_ps linear v4.xyz
    dcl_output o0.xyzw
    dcl_temps 3
    dp3 r0.x, v4.yzxx, v4.yzxz
    rsq r0.x, r0.x
    dp3 r0.y, v3.yzxx, v3.yzxz
    rsq r0.y, r0.y
    mul r0.yzw, r0.yyyy, v3.xxyz
    mad r1.xyz, v4.yzxx, r0.xxxx, r0.yzwy
    dp3 r0.x, r1.yzxx, r1.yzxz
    rsq r0.x, r0.x
    mul r1.xyz, r0.xxxx, r1.yzxx
    dp3 r0.x, v1.yzxx, v1.yzxz
    rsq r0.x, r0.x
```

```
mul r2.xyz, r0.xxxx, v1.xyzx
dp3 r0.x, r2.xyzx, r1.xyzx
dp3 r0.y, r2.xyzx, r0.yzwy
ge r0.zw, r0.xyyy, l(0.000000, 0.000000, 0.000000, 0.000000)
log r0.x, r0.x
mul r0.x, r0.x, cb1[9].x
exp r0.x, r0.x
max r0.y, r0.y, l(0.000000)
and r0.z, r0.w, r0.z
and r0.x, r0.x, r0.z
sample_indexable(texture2d) (float,float,float,float) r1.xyzw,
v2.xyxx, t0.xyzw, s0
min r0.x, r0.x, r1.w
mul r0.yzw, r0.yyyy, r1.xxyz
mul r2.xyz, cb0[1].wwww, cb0[1].xyzx
mul r0.yzw, r0.yyzw, r2.xxyz
mul r2.xyz, cb0[0].wwww, cb0[0].xyzx
mad r0.yzw, r2.xxyz, r1.xxyz, r0.yyzw
mul r1.xyz, cb1[8].wwww, cb1[8].xyzx
mad o0.xyz, r1.xyzx, r0.xxxx, r0.yzwy
mov o0.w, l(1.000000)
ret
// Approximately 32 instruction slots used
};
```

---

```
#include "include\\Common.fxh"

/********************* Resources ********************/

cbuffer CBufferPerFrame
{
    float4 AmbientColor = {1.0f, 1.0f, 1.0f, 0.0f};
    float4 LightColor = {1.0f, 1.0f, 1.0f, 1.0f};
    float3 LightPosition = {0.0f, 0.0f, 0.0f};
    float LightRadius = 10.0f;
    float3 CameraPosition : CAMERAPosition;
}

cbuffer CBufferPerObject
{
    float4x4 ViewProjection : VIEWPROJECTION;
}

Texture2D ColorTexture;

SamplerState TrilinearSampler
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};
```

```
***** Data Structures *****
```

```
struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
    row_major float4x4 World : WORLD;
    float4 SpecularColor : SPECULARCOLOR;
    float SpecularPower : SPECULARPOWER;
};
```

```
struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD;
    float3 WorldPosition : WORLD;
    float Attenuation : ATTENUATION;
    float4 SpecularColor : SPECULAR;
    float SpecularPower : SPECULARPOWER;
};
```

```
***** Vertex Shader *****
```

```
VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.WorldPosition = mul(IN.ObjectPosition, IN.World).xyz;
    OUT.Position = mul(float4(OUT.WorldPosition, 1.0f),
ViewProjection);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), IN.World).xyz);
    OUT.TextureCoordinate = IN.TextureCoordinate;

    float3 lightDirection = LightPosition - OUT.WorldPosition;
    OUT.Attenuation = saturate(1.0f - (length(lightDirection) /
LightRadius));
    OUT.SpecularColor = IN.SpecularColor;
    OUT.SpecularPower = IN.SpecularPower;

    return OUT;
}

/***** Pixel Shader *****/
float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 lightDirection = LightPosition - IN.WorldPosition;
    lightDirection = normalize(lightDirection);

    float3 viewDirection = normalize(CameraPosition
- IN.WorldPosition);

    float3 normal = normalize(IN.Normal);
    float n_dot_l = dot(normal, lightDirection);
    float3 halfVector = normalize(lightDirection + viewDirection);
    float n_dot_h = dot(normal, halfVector);

    float4 color = ColorTexture.Sample(TrilinearSampler,
IN.TextureCoordinate);
    float4 lightCoefficients = lit(n_dot_l, n_dot_h, IN.SpecularPower);

    float3 ambient = get_vector_color_contribution(AmbientColor, color.
rgb);
    float3 diffuse = get_vector_color_contribution(LightColor,
lightCoefficients.y * color.rgb) * IN.Attenuation;
    float3 specular = get_scalar_color_contribution(IN.SpecularColor,
min(lightCoefficients.z, color.w)) * IN.Attenuation;
```

```
    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}

/***** Techniques *****/
technique11 main11
{
    pass p0
    {
        SetVertexShader(CompileShader(vs_5_0, vertex_shader()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_5_0, pixel_shader()));
    }
}
```

---

```
D3D11_INPUT_ELEMENT_DESC inputElementDescriptions[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 16, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "WORLD", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "WORLD", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "WORLD", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "WORLD", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "SPECULARCOLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 64, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "SPECULARPOWER", 0, DXGI_FORMAT_R32_FLOAT, 1, 80, D3D11_INPUT_PER_INSTANCE_DATA, 1 }
};
```

```
void InstancingDemo::Initialize()
{
    SetCurrentDirectory(utility::ExecutableDirectory().c_str());

    std::unique_ptr<Model> model(new Model(*mGame, "Content\\Models\\Sphere.obj", true));

    // Initialize the material
    mEffect = new Effect(*mGame);
    mEffect->LoadCompiledEffect(L"Content\\Effects\\Instancing.cso");
    mMaterial = new InstancingMaterial();
    mMaterial->Initialize(*mEffect);

    // Create vertex buffer
    Mesh* mesh = model->Meshes().at(0);
    ID3D11Buffer* vertexBuffer = nullptr;
    mMaterial->CreateVertexBuffer(mGame->Direct3DDevice(), *mesh,
&vertexBuffer);
    mVertexBuffers.push_back(new VertexBufferData(vertexBuffer,
mMaterial->VertexSize(), 0));
}
```

```
// Create instance buffer
std::vector<InstancingMaterial::InstanceData> instanceData;
UINT axisInstanceCount = 5;
float offset = 20.0f;
for (UINT x = 0; x < axisInstanceCount; x++)
{
    float xPosition = x * offset;

    for (UINT z = 0; z < axisInstanceCount; z++)
    {
        float zPosition = z * offset;

        instanceData.push_back(InstancingMaterial::InstanceData
        ( XMMatrixTranslation(-xPosition, 0, -zPosition), ColorHelper::ToFloat4
        (mSpecularColor), mSpecularPower));
        instanceData.push_back(InstancingMaterial::InstanceData
        ( XMMatrixTranslation(xPosition, 0, -zPosition), ColorHelper::ToFloat4
        (mSpecularColor), mSpecularPower));
    }
}

ID3D11Buffer* instanceBuffer = nullptr;
mMaterial->CreateInstanceBuffer(mGame->Direct3DDevice(),
instanceData, &instanceBuffer);
mInstanceCount = instanceData.size();
mVertexBuffers.push_back(new VertexBufferData(instanceBuffer,
mMaterial->InstanceSize(), 0));
```

```
// Create index buffer
mesh->CreateIndexBuffer(&mIndexBuffer);
mIndexCount = mesh->Indices().size();

std::wstring textureName = L"Content\\Textures\\EarthComposite.
jpg";
HRESULT hr = DirectX::CreateWICTextureFromFile(mGame-
>Direct3DDevice(), mGame->Direct3DDeviceContext(), textureName.c_str(),
nullptr, &mColorTexture);
if (FAILED(hr))
{
    throw GameException("CreateWICTextureFromFile() failed.", hr);
}

mPointLight = new PointLight(*mGame);
mPointLight->SetRadius(500.0f);
mPointLight->SetPosition(5.0f, 0.0f, 10.0f);

mKeyboard = (Keyboard*)mGame->Services().GetService
(Keyboard::TypeIdClass());
assert(mKeyboard != nullptr);

mProxyModel = new ProxyModel(*mGame, *mCamera, "Content\\Models\\
PointLightProxy.obj", 0.5f);
mProxyModel->Initialize();
}
```

```
void InstancingDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();
    direct3DDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_
TOPOLOGY_TRIANGLELIST);

    Pass* pass = mMaterial->CurrentTechnique()->Passes().at(0);
    ID3D11InputLayout* inputLayout = mMaterial->InputLayouts().
at(pass);
    direct3DDeviceContext->IASetInputLayout(inputLayout);

    ID3D11Buffer* vertexBuffers[2] = { mVertexBuffers[0]->VertexBuffer,
mVertexBuffers[1]->VertexBuffer };
    UINT strides[2] = { mVertexBuffers[0]->Stride, mVertexBuffers[1]
->Stride };
    UINT offsets[2] = { mVertexBuffers[0]->Offset, mVertexBuffers[1]
->Offset };

    direct3DDeviceContext->IASetVertexBuffers(0, 2, vertexBuffers,
strides, offsets);
    direct3DDeviceContext->IASetIndexBuffer(mIndexBuffer, DXGI_FORMAT_
R32_UINT, 0);

    mMaterial->ViewProjection() << mCamera->ViewMatrix() *
mCamera->ProjectionMatrix();
    mMaterial->AmbientColor() << XMLoadColor(&mAmbientColor);
    mMaterial->LightColor() << mPointLight->ColorVector();
    mMaterial->LightPosition() << mPointLight->PositionVector();
    mMaterial->LightRadius() << mPointLight->Radius();
    mMaterial->ColorTexture() << mColorTexture;
    mMaterial->CameraPosition() << mCamera->PositionVector();

    pass->Apply(0, direct3DDeviceContext);

    direct3DDeviceContext->DrawIndexedInstanced(mIndexCount,
mInstanceCount, 0, 0, 0);

    mProxyModel->Draw(gameTime);
}
```

---

```
RWTexture2D<float4> OutputTexture;

cbuffer CBufferPerFrame
{
    float2 TextureSize;
    float BlueColor;
};

[numthreads(32, 32, 1)]
void compute_shader(uint3 threadID : SV_DispatchThreadID)
{
    float4 color = float4((threadID.xy / TextureSize), BlueColor, 1);
    OutputTexture[threadID.xy] = color;
}

technique11 compute
{
    pass p0
    {
        SetVertexShader(NULL);
        SetGeometryShader(NULL);
        SetPixelShader(NULL);
        SetComputeShader(CompileShader(cs_5_0, compute_shader()));
    }
}
```

---

```
D3D11_TEXTURE2D_DESC textureDesc;
ZeroMemory(&textureDesc, sizeof(textureDesc));
textureDesc.Width = mGame->ScreenWidth();
textureDesc.Height = mGame->ScreenHeight();
textureDesc.MipLevels = 1;
textureDesc.ArraySize = 1;
textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.SampleDesc.Count = 1;
textureDesc.SampleDesc.Quality = 0;
textureDesc.BindFlags = D3D11_BIND_UNORDERED_ACCESS |
D3D11_BIND_SHADER_RESOURCE;

HRESULT hr;
ID3D11Texture2D* texture = nullptr;
if (FAILED(hr = mGame->Direct3DDevice()->CreateTexture2D(&textureDesc,
nullptr, &texture)))
{
    throw GameException("IDXGIDevice::CreateTexture2D() failed.", hr);
}

D3D11_UNORDERED_ACCESS_VIEW_DESC uavDesc;
ZeroMemory(&uavDesc, sizeof(uavDesc));
uavDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
uavDesc.ViewDimension = D3D11_UAV_DIMENSION_TEXTURE2D;
uavDesc.Texture2D.MipSlice = 0;
```

```
if (FAILED(hr = mGame->Direct3DDevice()->CreateUnorderedAccessView
(texture, &uavDesc, &mOutputTexture)))
{
    ReleaseObject(texture);
    throw GameException("IDXGIDevice::CreateUnorderedAccessView()
failed.", hr);
}

D3D11_SHADER_RESOURCE_VIEW_DESC resourceViewDesc;
ZeroMemory(&resourceViewDesc, sizeof(resourceViewDesc));
resourceViewDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
resourceViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
resourceViewDesc.Texture2D.MipLevels = 1;

if (FAILED(hr = mGame->Direct3DDevice()->
CreateShaderResourceView(texture, &resourceViewDesc, &mColorTexture)))
{
    ReleaseObject(texture);
    throw GameException("IDXGIDevice::CreateShaderResourceView()
failed.", hr);
}

ReleaseObject(texture);
```

---

```
void ComputeShaderDemo::Draw(const GameTime& gameTime)
{
    ID3D11DeviceContext* direct3DDeviceContext =
mGame->Direct3DDeviceContext();

    // Update the compute shader's material
    mMaterial->TextureSize() << XMLoadFloat2(&mTextureSize);
    mMaterial->BlueColor() << mBlueColor;
    mMaterial->OutputTexture() << mOutputTexture;
    mComputePass->Apply(0, direct3DDeviceContext);

    // Dispatch the compute shader
    direct3DDeviceContext->Dispatch(mThreadGroupCount.x,
mThreadGroupCount.y, 1);

    // Unbind the UAV from the compute shader, so we can bind the same
underlying resource as an SRV
    static ID3D11UnorderedAccessView* emptyUAV = nullptr;
    direct3DDeviceContext->CSSetUnorderedAccessViews(0, 1, &emptyUAV,
nullptr);

    // Draw the texture written by the compute shader
    mFullScreenQuad->Draw(gameTime);
    mGame->UnbindPixelShaderResources(0, 1);
}

void ComputeShaderDemo::UpdateRenderingMaterial()
{
    mMaterial->ColorTexture() << mColorTexture;
}
```

```
void ComputeShaderDemo::Update(const GameTime& gameTime)
{
    mBlueColor = (0.5f) * static_cast<float>(sin(gameTime.
TotalGameTime())) + 0.5f;
}
```

```
<!-- Game.xml, the root asset store of the configuration hierarchy -->
<AssetStore>
    <Items>
        <Item Class="AssetStore" File="Content\Levels\Level1.xml" />
        <Item Class="AssetStore" File="Content\Levels\Level2.xml" />
        <Item Class="AssetStore" File="Content\Levels\Level3.xml" />
    </Items>
</AssetStore>

<!-- Level1.xml -->
<AssetStore>
    <Items>
        <Item Class="Camera" FOV="0.785398" AspectRatio="1.33333"
NearPlaneDistance="1.0" FarPlaneDistance="1000.0">
            <Position X="0" Y="0" Z="0" />
            <Direction X="0" Y="0" Z="-1" />
            <Up X="0" Y="1" Z="0" />
        </Item>

        <Item Class="Skybox" CubemapFile="Content\Textures\Maskonaiive2_1024.
dds" Scale="500" />
    </Items>
</AssetStore>
```

```
template <class T>
class Factory
{
public:
    virtual ~Factory();

    virtual const std::string ClassName() const = 0;
    virtual T* Create() const = 0;

    static Factory<T>* Find(const std::string& className);
    static T* Create(const std::string& className);

    static typename std::map<std::string, Factory<T>*>::iterator
Begin();
    static typename std::map<std::string, Factory<T>*>::iterator End();

protected:
    static void Add(Factory<T>* const factory);
    static void Remove(Factory<T>* const factory);

private:
    static std::map<std::string, Factory<T>*> sFactories;
};
```

---

```
#define ConcreteFactory(ConcreteProductT, AbstractProductT) \
class ConcreteProductT ## Factory : public Factory<AbstractProductT> \
{ \
public: \
    ConcreteProductT ## Factory() \
    { \
        Add(this); \
    } \
 \
    ~ConcreteProductT ## Factory() \
    { \
        Remove(this); \
    } \
 \
    virtual const std::string ClassName() const \
    { \
        return std::string( #ConcreteProductT ); \
    } \
 \
    virtual AbstractProductT* Create() const \
    { \
        AbstractProductT* product = new ConcreteProductT(); \
        return product; \
    } \
};
```

ConcreteFactory(Skybox, RTTI)

```
RTTI* skybox = Factory<RTTI>::Create("Skybox");
```