A Practical Guide to Graphics Programming

Real-Time 3D Rendering with DirectX and HLSL

Paul Varcholik

Free Sample Chapter
Share with Others
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
This page intentionally left blank
Real-Time 3D Rendering with DirectX® and HLSL
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.

Visit informit.com/series/gamedesign for a complete list of available publications.
Real-Time 3D Rendering with DirectX® and HLSL

A Practical Guide to Graphics Programming

Paul Varcholik
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 or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: 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, MSDOS, 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.


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

First printing: April 2014
Contents-at-a-Glance

Introduction ................................................. 1

Part I  An Introduction to 3D Rendering ..................... 5
  1  Introducing DirectX ..................................... 7
  2  A 3D/Math Primer ....................................... 23
  3  Tools of the Trade ...................................... 43

Part II  Shader Authoring with HLSL ......................... 57
  4  Hello, Shaders! ......................................... 59
  5  Texture Mapping ......................................... 73
  6  Lighting Models .......................................... 91
  7  Additional Lighting Models .............................. 115
  8  Gleaming the Cube ....................................... 141
  9  Normal Mapping and Displacement Mapping .......... 169

Part III  Rendering with DirectX ............................. 183
  10 Project Setup and Window Initialization ............... 185
  11 Direct3D Initialization ................................ 205
  12 Supporting Systems .................................... 233
  13 Cameras .................................................. 269
  14 Hello, Rendering! ....................................... 283
  15 Models .................................................... 315
  16 Materials ................................................ 341
  17 Lights .................................................... 371
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Part IV</strong> Intermediate-Level Rendering Topics</td>
<td>389</td>
</tr>
<tr>
<td>18</td>
<td>Post-Processing</td>
<td>391</td>
</tr>
<tr>
<td>19</td>
<td>Shadow Mapping</td>
<td>435</td>
</tr>
<tr>
<td>20</td>
<td>Skeletal Animation</td>
<td>469</td>
</tr>
<tr>
<td>21</td>
<td>Geometry and Tessellation Shaders</td>
<td>497</td>
</tr>
<tr>
<td>22</td>
<td>Additional Topics in Modern Rendering</td>
<td>529</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>555</td>
</tr>
</tbody>
</table>
Contents

Introduction ......................................................... 1

Part I  An Introduction to 3D Rendering ......................... 5
1 Introducing DirectX .............................................. 7
   A Bit of History ............................................. 8
   The Direct3D 11 Graphics Pipeline ....................... 9
   Summary ..................................................... 21

2 A 3D/Math Primer ............................................... 23
   Vectors ..................................................... 24
   Matrices ................................................... 27
   Transformations ........................................... 31
   DirectXMath ............................................... 35
   Summary ..................................................... 41

3 Tools of the Trade .............................................. 43
   Microsoft Visual Studio .................................... 44
   NVIDIA FX Composer ..................................... 47
   Visual Studio Graphics Debugger ....................... 53
   Graphics Debugging Alternatives ....................... 55
   Summary ..................................................... 56
   Exercises .................................................... 56

Part II  Shader Authoring with HLSL ......................... 57
4 Hello, Shaders! .................................................. 59
   Your First Shader .......................................... 60
   Hello, Structs! ............................................. 68
   Summary ..................................................... 70
   Exercises .................................................... 71
5 Texture Mapping .............................................. 73
  An Introduction to Texture Mapping .................. 74
  A Texture Mapping Effect ............................... 75
  Texture Filtering ........................................ 81
  Texture Addressing Modes ............................... 86
  Summary .................................................. 89
  Exercises ................................................ 89

6 Lighting Models ........................................... 91
  Ambient Lighting ......................................... 92
  Diffuse Lighting ........................................ 97
  Specular Highlights .................................... 105
  Summary .................................................. 114
  Exercises ................................................ 114

7 Additional Lighting Models ............................ 115
  Point Lights ............................................. 116
  Spotlights ............................................... 124
  Multiple Lights ........................................ 130
  Summary .................................................. 139
  Exercises ................................................ 139

8 Gleaming the Cube ....................................... 141
  Texture Cubes ........................................... 142
  Skyboxes ................................................ 145
  Environment Mapping .................................. 149
  Fog ...................................................... 154
  Color Blending .......................................... 159
  Summary .................................................. 167
  Exercises ................................................ 168
## Normal Mapping and Displacement Mapping

- Normal Mapping
  - pg. 170
- Displacement Mapping
  - pg. 178
- Summary
  - pg. 181
- Exercises
  - pg. 181

## Part III Rendering with DirectX

- Project Setup and Window Initialization
  - pg. 185
  - A New Beginning
    - pg. 186
  - Project Setup
    - pg. 186
  - The Game Loop
    - pg. 195
  - Window Initialization
    - pg. 199
  - Summary
    - pg. 204
  - Exercise
    - pg. 204

- Direct3D Initialization
  - pg. 205
  - Initializing Direct3D
    - pg. 206
  - Putting It All Together
    - pg. 219
  - Summary
    - pg. 232
  - Exercise
    - pg. 232

- Supporting Systems
  - pg. 233
  - Game Components
    - pg. 234
  - Device Input
    - pg. 248
  - Software Services
    - pg. 265
  - Summary
    - pg. 268
  - Exercises
    - pg. 268

- Cameras
  - pg. 269
  - A Base Camera Component
    - pg. 270
  - A First-Person Camera
    - pg. 277
  - Summary
    - pg. 281
  - Exercise
    - pg. 281
<table>
<thead>
<tr>
<th>14</th>
<th>Hello, Rendering!</th>
<th>283</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Your First Full Rendering Application</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>An Indexed Cube</td>
<td>306</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>314</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>314</td>
</tr>
<tr>
<td>15</td>
<td>Models</td>
<td>315</td>
</tr>
<tr>
<td></td>
<td>Motivation</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>Model File Formats</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>The Content Pipeline</td>
<td>317</td>
</tr>
<tr>
<td></td>
<td>The Open Asset Import Library</td>
<td>317</td>
</tr>
<tr>
<td></td>
<td>What's in a Model?</td>
<td>318</td>
</tr>
<tr>
<td></td>
<td>Meshes</td>
<td>320</td>
</tr>
<tr>
<td></td>
<td>Model Materials</td>
<td>321</td>
</tr>
<tr>
<td></td>
<td>Asset Loading</td>
<td>323</td>
</tr>
<tr>
<td></td>
<td>A Model Rendering Demo</td>
<td>331</td>
</tr>
<tr>
<td></td>
<td>Texture Mapping</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>340</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>340</td>
</tr>
<tr>
<td>16</td>
<td>Materials</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>Motivation</td>
<td>342</td>
</tr>
<tr>
<td></td>
<td>The Effect Class</td>
<td>342</td>
</tr>
<tr>
<td></td>
<td>The Technique Class</td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>The Pass Class</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>The Variable Class</td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>The Material Class</td>
<td>352</td>
</tr>
<tr>
<td></td>
<td>A Basic Effect Material</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>A Skybox Material</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>370</td>
</tr>
</tbody>
</table>
## CONTENTS

### Part IV Intermediate-Level Rendering Topics

#### 18 Post-Processing
- Render Targets .................................................. 392
- A Full-Screen Quad Component ................................. 396
- Color Filtering .................................................. 401
- Gaussian Blurring .............................................. 410
- Bloom .......................................................... 419
- Distortion Mapping ............................................. 425
- Summary ......................................................... 433
- Exercises ......................................................... 433

#### 19 Shadow Mapping
- Motivation ......................................................... 436
- Projective Texture Mapping ..................................... 436
- Shadow Mapping ................................................ 456
- Summary ......................................................... 466
- Exercises ......................................................... 467

#### 20 Skeletal Animation
- Hierarchical Transformations .................................. 470
- Skinning .......................................................... 472
- Importing Animated Models .................................... 476
### CONTENTS

**Animation Rendering** ............................................. 489  
Summary ............................................................. 496  
Exercises ............................................................ 496

#### 21 Geometry and Tessellation Shaders ........................ 497
Motivation: Geometry Shaders ....................................... 498  
Processing Primitives ................................................ 498  
A Point Sprite Shader ............................................... 499  
Primitive IDs .......................................................... 507  
Motivation: Tessellation Shaders ................................... 508  
The Hull Shader Stage ............................................... 510  
The Tessellation Stage ............................................... 512  
The Domain Shader Stage .......................................... 514  
A Basic Tessellation Demo ......................................... 518  
Displacing Tessellated Vertices ................................... 520  
Dynamic Levels of Detail .......................................... 524  
Summary ...................................................................... 527  
Exercises ..................................................................... 528

#### 22 Additional Topics in Modern Rendering .................. 529
Rendering Optimization ............................................... 530  
Deferred Shading ....................................................... 543  
Global Illumination ..................................................... 544  
Compute Shaders ....................................................... 545  
Data-Driven Engine Architecture ................................... 550  
The End of the Beginning ............................................ 553  
Exercises ..................................................................... 553

Index ............................................................................ 555
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.
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
INTRODUCTION

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.
This page intentionally left blank
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).

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

```cpp
// 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);
}

technique1 main1
{
  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 `float4x4`. This is a C-style variable declaration in which the data type is a 4×4 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 `float4x4` 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

```cpp
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`. 
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

```hlsl
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.

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`

```cpp
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.”
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

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 \texttt{vs} \_\texttt{4} \_\texttt{0} and \texttt{ps} \_\texttt{4} \_\texttt{0} within the \texttt{SetVertexShader} and \texttt{SetPixelShader} statements. These values identify the shader profiles to use when compiling the shaders specified in the second arguments of the \texttt{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 \texttt{Build}, \texttt{Rebuild All} or \texttt{Build, Compile HelloShaders.fx} menu commands. Alternately, you can use the shortcut keys \texttt{F6} (Rebuild All) or \texttt{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 \texttt{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.

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

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 float4x4) 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.
This page intentionally left blank
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
  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
class
  checking 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::UpdateColorFilter
    Material() method, 404
column-major order, 30
  comments, HLSL, 77
  common color blending settings, 159
  Common.fxh, 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

devices

demonstrations

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
gamma-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
- environment mapping, 149-154
- dynamic environment mapping, 153-154
- environment 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
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
gamey shader stage (Direct3D Graphics Pipeline), 18

G
Game class
   header file, 200-201
   ServiceContainer member, adding, 267-268
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
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

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
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
输出
  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
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

Q-R
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
sepias 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
greyscale 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
updating WinMain() function for RenderingGame class, 230-232
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-merge 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