A PRACTICAL GUIDE TO GRAPHICS PROGRAMMING

REAL-TIME 3D RENDERING with DIRECTX® and HLSL

Paul **VARCHOLIK**

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.

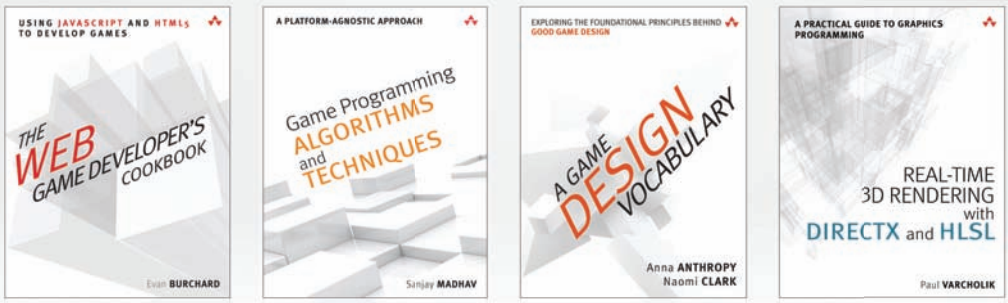
"A great tour of the modern DirectX landscape, with a heavy emphasis on authoring HLSL shaders for common game rendering techniques for C++ developers."

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

This page intentionally left blank

Real-Time 3D Rendering with DirectX® and HLSL

The Addison-Wesley Game Design and Development Series



Addison-Wesley

Visit **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.

Real-Time 3D Rendering with DirectX® and HLSL

A Practical Guide to Graphics Programming

Paul Varcholik

Addison-Wesley

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2014933263

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

Contents-at-a-Glance

Contents

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.

This page intentionally left blank

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).

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

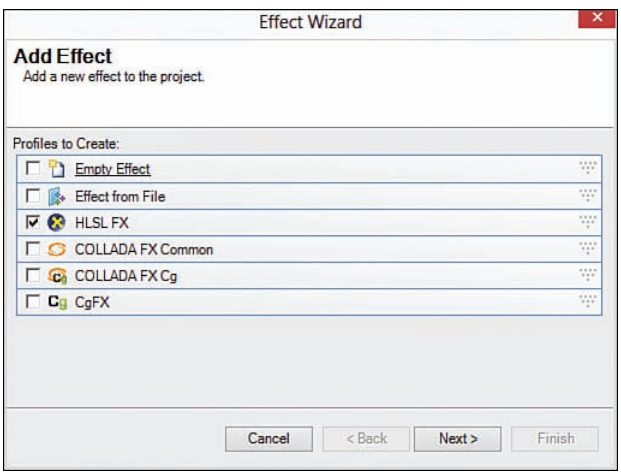


Figure 4.1 NVIDIA FX Composer Add Effect dialog box.

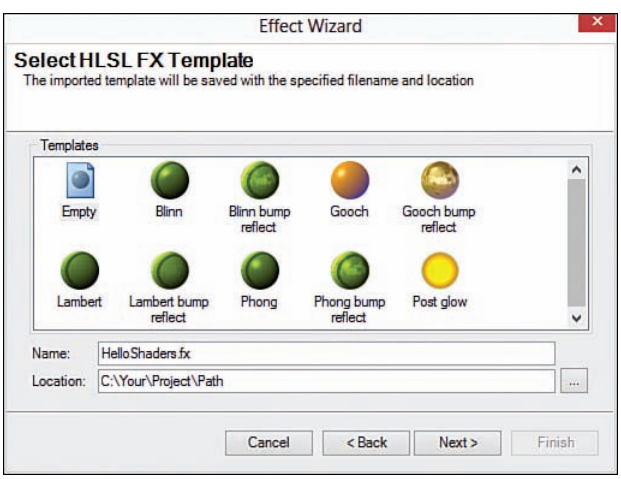


Figure 4.2 NVIDIA FX Composer Select HLSL FX Template dialog box.

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`

```
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, `cbuffer`s 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 `cbuffer`s 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`

```
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`

```
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`

```
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`

```
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.

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.

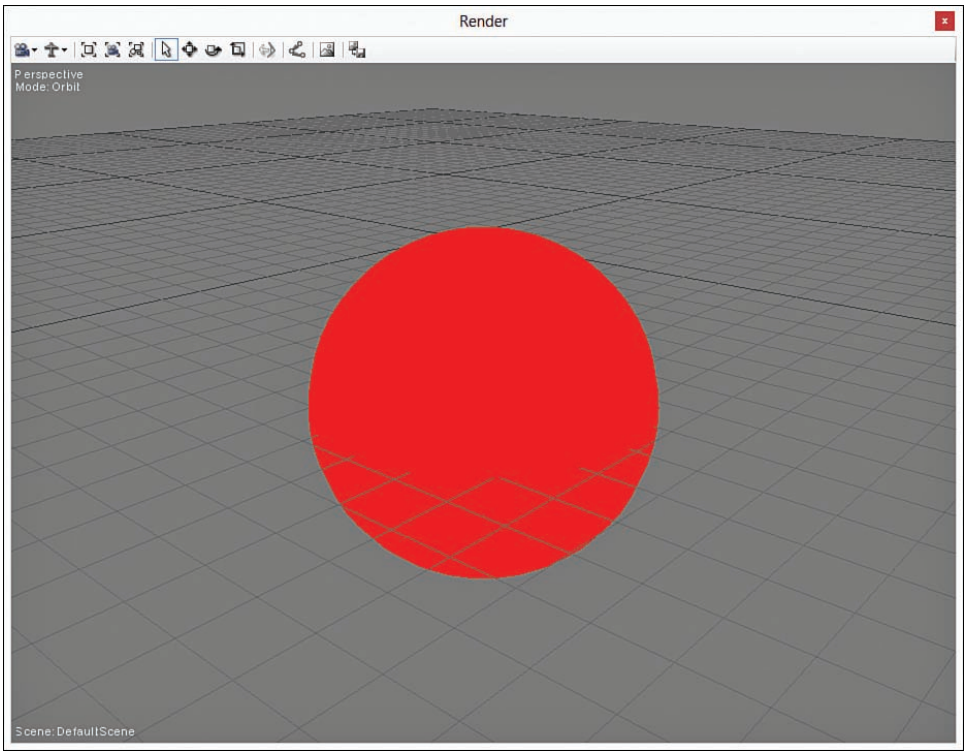


Figure 4.3 `HellShaders.fx` applied to a sphere in the NVIDIA FX Composer Render panel.

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

This page intentionally left blank

INDEX

Numerics

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V

W

X-Y-Z