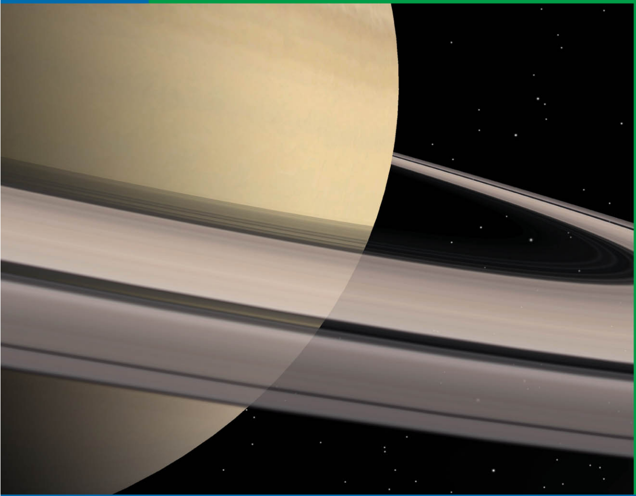
OpenGL® SuperBible

Seventh Edition
Comprehensive Tutorial and Reference



Graham Sellers ▪ Richard S. Wright, Jr. ▪ Nicholas Haemel

OpenGL®

SuperBible

Seventh Edition

OpenGL®

SuperBible

Seventh Edition

Comprehensive Tutorial and Reference

Graham Sellers
Richard S. Wright, Jr.
Nicholas Haemel

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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 United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Wright, Richard S., Jr., 1965- author.
OpenGL superBible : comprehensive tutorial and reference.—Seventh edition / Graham Sellers, Richard S. Wright, Jr., Nicholas Haemel.
pages cm
Includes bibliographical references and index.
ISBN 978-0-672-33747-5 (pbk. : alk. paper)—ISBN 0-672-33747-9 (pbk. : alk. paper)
1. Computer graphics. 2. OpenGL. I. Sellers, Graham, author. II. Haemel, Nicholas, author. III. Title.
T385.W728 2016
006.6'8—dc23
2015014278

ISBN-13: 978-0-672-33747-5
ISBN-10: 0-672-33747-9
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2015

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Jill Hobbs

Indexer
Larry D. Sweazy

Proofreader
Anna Popick

Technical Reviewer
Matías Goldberg

Editorial Assistant
Olivia Basegio

Compositor
diacriTech

For you, the reader.
—Graham Sellers

This page intentionally left blank

Contents

II In Depth 239

Figures

Tables

Listings

Foreword

When OpenGL was young, the highest-end SGI systems like the Reality Engine 2 cost $80,000 and could render 200,000 textured triangles per second, or 3,333 triangles per frame at 60 Hz. The CPUs of that era were slower than today, to be sure, but at around 100 MHz, that's still 500 CPU cycles for each triangle. It was pretty easy to be graphics limited back then, and the API reflected that—the only way to specify geometry was immediate mode! Well, there were also display lists for static geometry, which made being graphics-limited even easier.

OpenGL is not young anymore, the highest-end GPUs that it can run on cost around $1000, and they don't even list triangles per second in their basic product description anymore, but the number is north of 6 billion. Today these GPUs are in the middle of the single digit teraflops and several hundred gigabytes per second of bandwidth. CPUs have gotten faster, too: With 4 cores and around 3 GHz, they are shy of 200 gigaflops and have around 20 gigabytes per second of memory bandwidth. So where we had 500 CPU cycles for a triangle in the early days, we now have 0.5 cycles. Even if we could perfectly exploit all 4 cores, that would give us a paltry 2 CPU cycles for each triangle!

All that is to say that the growth in hardware graphics performance has outstripped conventional CPU performance growth by several orders of magnitude, and the consequences are pretty obvious today. Not only is the CPU frequently the limiting factor in graphics performance, we have an API that was designed against a different set of assumptions.

The good news with OpenGL is that it has evolved too. First it added vertex arrays so that a single draw command with fairly low CPU overhead gets amplified into a lot of GPU work. This helped for a while, but it wasn't enough. We added instancing to further increase the amount of work, but this was a somewhat limited form of work amplification, as we don't always want many instances of the same object in an organic, believable rendering.

Recognizing that these emerging limitations in the API had to be circumvented somehow, OpenGL designers began extending the interface to remove as much CPU-side overhead from the interface as possible. The "bindless" family of extensions allows the GPU to reference buffers and textures directly rather than going through expensive binding calls in the driver. Persistent maps allow the application to scribble on memory at the same time the GPU is referencing it. This sounds dangerous—and it can be!—but allowing the application to manage memory hazards relieves a tremendous burden from the driver and allows for far simpler, less general mechanisms to be employed. Sparse texture arrays allow applications to manage texture memory as well with similar, very low-overhead benefits. And finally multi-draw and multi-draw indirect added means the GPU can generate the very buffers that it sources for drawing, leaving the CPU a lot more available for other work.

All of these advances in OpenGL have been loosely lumped under the *AZDO* (Approaching Zero Driver Overhead) umbrella, and most of them have been incorporated into the core API. There are still significant areas for improvement as we try to get to an API that allows developers to render as much as they want, the way they want, without worrying that the CPU or driver overhead will get in the way. These features require a bit more work to make use of, but the results can be truly amazing! This edition of the *OpenGL® SuperBible* includes many new examples that make use of AZDO features and provide good guidance on how to get the CPU out of the way. In particular, you'll learn good ways to make use of zero copy, proper fencing, and bindless.

Cass Everitt
Oculus

Preface

This book is designed both for people who are learning computer graphics through OpenGL and for people who may already know about graphics but want to learn about OpenGL. The intended audience is students of computer science, computer graphics, or game design; professional software engineers; or simply just hobbyists and people who are interested in learning something new. We begin by assuming that the reader knows nothing about either computer graphics or OpenGL. The reader should be familiar with computer programming in C++, however.

One of our goals with this book was to ensure that there were as few forward references as possible and to require little or no assumed knowledge. The book is accessible and readable, and if you start from the beginning and read all the way through, you should come away with a good comprehension of how OpenGL works and how to use it effectively in your applications. After reading and understanding the content of this book, you will be well positioned to read and learn from more advanced computer graphics research articles and confident that you can take the principles that they cover and implement them in OpenGL.

It is *not* a goal of this book to cover every last feature of OpenGL—that is, to mention every function in the specification or every value that can be passed to a command. Rather, we intend to provide a solid understanding of OpenGL, introduce the fundamentals, and explore some of its more advanced features. After reading this book, readers should be comfortable looking up finer details in the OpenGL specification, experimenting with

OpenGL on their own machines, and using extensions (bonus features that add capabilities to OpenGL not required by the main specification).

The Architecture of the Book

This book is subdivided into three parts. In Part I, “Foundations,” we explain what OpenGL is and how it connects to the graphics pipeline, and we give minimal working examples that are sufficient to demonstrate each section of it without requiring much, if any, knowledge of any other part of the whole system. We lay a foundation for the math behind three-dimensional computer graphics, and describe how OpenGL manages the large amounts of data that are required to provide a compelling experience to the users of such applications. We also describe the programming model for *shaders*, which form a core part of any OpenGL application.

In Part II, “In Depth,” we introduce features of OpenGL that require some knowledge of multiple parts of the graphics pipeline and may refer to concepts mentioned in Part I. This allows us to cover more complex topics without glossing over details or telling you to skip forward in the book to find out how something really works. By taking a second pass over the OpenGL system, we are able to delve into where data goes as it leaves each part of OpenGL, as you’ll already have been (at least briefly) introduced to its destination.

Finally, in Part III, “In Practice,” we dive deeper into the graphics pipeline, cover some more advanced topics, and give a number of examples that use multiple features of OpenGL. We provide a number of worked examples that implement various rendering techniques, give a series of suggestions and advice on OpenGL best practices and performance considerations, and end up with a practical overview of OpenGL on several popular platforms, including mobile devices.

In Part I, we start gently and then blast through OpenGL to give you a taste of what’s to come. Then, we lay the groundwork of knowledge that will be essential to you as you progress through the rest of the book. In this part, you will find the following chapters:

- Chapter 1, “Introduction,” provides a brief introduction to OpenGL, including its origins, history, and current state.
- Chapter 2, “Our First OpenGL Program,” jumps right into OpenGL and shows you how to create a simple OpenGL application using the source code provided with this book.

- Chapter 3, “Following the Pipeline,” takes a more careful look at OpenGL and its various components, introducing each in a little more detail and adding to the simple example presented in the previous chapter.
- Chapter 4, “Math for 3D Graphics,” introduces the foundations of math that is essential for effective use of OpenGL and the creation of interesting 3D graphics applications.
- Chapter 5, “Data,” provides you with the tools necessary to manage data that will be consumed and produced by OpenGL.
- Chapter 6, “Shaders and Programs,” takes a deeper look at *shaders*, which are fundamental to the operation of modern graphics applications.

In Part II, we take a more detailed look at several of the topics introduced in the first chapters. We dig deeper into each of the major parts of OpenGL and our example applications start to become a little more complex and interesting. In this part, you will find these six chapters:

- Chapter 7, “Vertex Processing and Drawing Commands,” covers the inputs to OpenGL and the mechanisms by which semantics are applied to the raw data you provide.
- Chapter 8, “Primitive Processing,” covers some higher-level concepts in OpenGL, including connectivity information, higher-order surfaces, and tessellation.
- Chapter 9, “Fragment Processing and the Framebuffer,” looks at how high-level 3D graphics information is transformed by OpenGL into 2D images, and how your applications can determine the appearance of objects on the screen.
- Chapter 10, “Compute Shaders,” illustrates how your applications can harness OpenGL for more than just graphics and make use of the incredible computing power locked up in a modern graphics card.
- Chapter 11, “Advanced Data Management,” discusses topics related to managing large data sets, loading data efficiently, and arbitrating access to that data once loaded.
- Chapter 12, “Controlling and Monitoring the Pipeline,” shows you how to get a glimpse into how OpenGL executes the commands you give it—including how long they take to execute, and how much data they produce.

In Part III, we build on the knowledge that you will have gained in reading the first two parts of the book and use it to construct example applications that touch on multiple aspects of OpenGL. We also get into the practicalities of building larger OpenGL applications and deploying them across multiple platforms. In this part, you will find three chapters:

- Chapter 13, "Rendering Techniques," covers several applications of OpenGL for graphics rendering, including simulation of light, artistic methods and even some nontraditional techniques.
- Chapter 14, "High-Performance OpenGL," digs into some topics related to getting the highest possible performance from OpenGL.
- Chapter 15, "Debugging and Stability," provides advice and tips on how to get your applications running without errors and how to debug problems with your programs.

Finally, several appendices are provided that describe the tools and file formats used in this book, discuss which versions of OpenGL support which features and list which extensions introduced those features, and give pointers to more useful OpenGL resources.

What's New in This Edition

In this book, we have expanded on the sixth edition to cover new features and topics introduced in OpenGL in versions 4.4 and 4.5 of the API. In the previous edition, we did not cover extensions—features that are entirely optional and not a mandatory part of the OpenGL core—and so left out a number of interesting topics. Since the release of the sixth edition of this book, some of these extensions have become fairly ubiquitous; in turn, we have decided to cover the ARB and KHR extensions. Thus extensions that have been ratified by Khronos (the OpenGL governing body) are part of this book.

We have built on the previous edition by expanding the book's application framework and adding new chapters and appendices that provide further insight and cover new topics. One important set of features enabled by the extensions that are now part of the book are the AZDO (Approaching Zero Driver Overhead) features, which are a way of using OpenGL that produces very low software overhead and correspondingly high performance. These features include *persistent maps* and *bindless textures*.

To make room for the new content, we decided to remove the chapter on platform specifics, which covered per-platform window system bindings. Also gone is official support for the Apple Mac platform. Almost all of the new content in this edition requires features introduced with OpenGL 4.4 or 4.5, or recent OpenGL extensions—none of which were supported by OS X at the time of writing. There is no expectation that Apple will further invest in its OpenGL implementation, so we encourage our readers to move away from the platform. To support multiple platforms, we recommend the use of cross-platform toolkits such as the excellent `SDL` (`https://www.libsdl.org/`) or `glfw` (`http://www.glfw.org/`) libraries. In fact, this book's framework is built on `glfw`, and it works well for us.

This book includes several new example applications, including demonstrations of new features, a texture compressor, text drawing, font rendering using distance fields, high-quality texture filtering, and multi-threaded programs using OpenMP. We also tried to address all of the errata and feedback we've received from our readers since the publication of the previous edition. We believe this to be the best update yet to the *OpenGL® SuperBible* yet.

We hope you enjoy it.

How to Build the Examples

Retrieve the sample code from the book's companion Web site, `http://www.openglsuperbible.com`, unpack the archive to a directory on your computer, and follow the instructions in the included `HOWTOBUILD.TXT` file for your platform of choice. The book's source code has been built and tested on Microsoft Windows (Windows 7 or later is required) and Linux (several major distributions). It is recommended that you install any available operating system updates and obtain the most recent graphics drivers from your graphics card manufacturer.

You may notice some minor discrepancies between the source code printed in this book and that in the source files. There are a number of reasons for this:

- This book is about OpenGL 4.5—the most recent version at the time of writing. The examples printed in the book are written assuming that OpenGL 4.5 is available on the target platform. However, we understand that in practice, operating systems, graphics drivers, and platforms may not have the *latest and greatest* available.

Consequently, where possible, we've made minor modifications to the example applications to allow them to run on earlier versions of OpenGL.

- Several months passed between when this book's text was finalized for printing and when the sample applications were packaged and posted to the Web. In that time, we discovered opportunities for improvement, whether that was uncovering new bugs, platform dependencies, or optimizations. The latest version of the source code on the Web will have those fixes and tweaks applied and will therefore deviate from the necessarily static copy printed in the book.
- There is not necessarily a one-to-one mapping of listings in the book's text and example applications in the Web package. Some example applications demonstrate more than one concept, some aren't mentioned in the book at all, and some listings in the book don't have an equivalent example application. Where possible, we've mentioned which of the example applications correspond to the listings in the book. We recommend that the reader take a close look at the example application package, as it includes some nuggets that may not be mentioned in the book.

Errata

We made a bunch of mistakes—we're certain of it. It's incredibly frustrating as an author to spot an error that you made and know that it has been printed, in books that your readers paid for, thousands and thousands of times. We have to accept that this will happen, though, and do our best to correct issues as we are able. If you think you see something that doesn't quite gel, check the book's Web site for errata:

`http://www.openglsuperbible.com`

Note from the Publisher

Some of the figures in the print edition of the book are dark due to the nature of the images themselves. To assist readers, color PDFs of figures are freely available at http://www.openglsuperbible.com and http://informit.com/title/9780672337475. In addition, PowerPoint slides of the figures for professors' classroom use are available at www.pearsonhighered.com/educator/product/OpenGL-Superbible-Comprehensive-Tutorial-and-Reference/9780672337475.page.

Acknowledgments

First, thanks to you—the reader. The best part of what I do is knowing that someone I've never met might benefit from all this. It's the biggest thrill, and the reason why people like me do this. I appreciate that you're reading this now and hope you get as much out of this book as I put into it.

I'd like to thank my wonderful wife, Chris, who's put up with me disappearing into my office for three editions of this book now. She's worked around my deadlines and cheered me on as I made (sometimes slow and painful) progress. I couldn't have done this without her. Thanks, too, to my kids, Jeremy and Emily. The answer to "What are you doing, dad?" is almost always "Working"—and you've always taken it in stride.

Thanks to my coauthors, Richard and Nick. You've let me run alone on this edition, but your names are on the cover because of your contributions—your fingerprints are etched into this book. Many thanks to Matías Goldberg, who performed a thorough technical review of the book on short notice.

Thanks again to Laura Lewin and Olivia Basegio and the Pearson team for letting me be me and just dropping random files and documents off whenever I felt like it. I don't work well with a plan, but seem to relish pressure and am really excellent at procrastination. I'm glad you guys put up with me.

Graham Sellers

This page intentionally left blank

About the Author

Graham Sellers is a classic geek. His family got their first computer (a BBC Model B) right before his sixth birthday. After his mum and dad stayed up all night programming it to play "Happy Birthday," he was hooked and determined to figure out how it worked. Next came basic programming and then assembly language. His first real exposure to graphics was via "demos" in the early 1990s, and then through Glide, and finally OpenGL in the late 1990s. Graham holds a master's in engineering from the University of Southampton, England.

Currently, Graham is a software architect at AMD. He represents AMD at the OpenGL ARB and has contributed to many extensions and to the core OpenGL Specification. Prior to that, he was a team lead at Epson, implementing OpenGL-ES and OpenVG drivers for embedded products. Graham holds several patents in the fields of computer graphics and image processing. When he's not working on OpenGL, he likes to disassemble and reverse-engineer old video game consoles (just to see how they work and what he can make them do). Originally from England, Graham now lives in Orlando, Florida, with his wife and two children.

This page intentionally left blank

Chapter 3

Following the Pipeline

WHAT YOU'LL LEARN IN THIS CHAPTER

- What each of the stages in the OpenGL pipeline does.
- How to connect your shaders to the fixed-function pipeline stages.
- How to create a program that uses every stage of the graphics pipeline simultaneously.

In this chapter, we will walk all the way along the OpenGL pipeline from start to finish, providing insight into each of the stages, which include fixed-function blocks and programmable shader blocks. You have already read a whirlwind introduction to the vertex and fragment shader stages. However, the application that you constructed simply drew a single triangle at a fixed position. If we want to render anything interesting with OpenGL, we're going to have to learn a lot more about the pipeline and all of the things you can do with it. This chapter introduces every part of the pipeline, hooks them up to one another, and provides an example shader for each stage.

Passing Data to the Vertex Shader

The vertex shader is the first *programmable* stage in the OpenGL pipeline and has the distinction of being the only mandatory stage in the graphics pipeline. However, before the vertex shader runs, a fixed-function stage known as *vertex fetching*, or sometimes *vertex pulling*, is run. This automatically provides inputs to the vertex shader.

Vertex Attributes

In GLSL, the mechanism for getting data in and out of shaders is to declare global variables with the `in` and `out` storage qualifiers. You were briefly introduced to the `out` qualifier in Chapter 2, "Our First OpenGL Program," when Listing 2.4 used it to output a color from the fragment shader. At the start of the OpenGL pipeline, we use the `in` keyword to bring inputs into the vertex shader. Between stages, `in` and `out` can be used to form conduits from shader to shader and pass data between them. We'll get to that shortly. For now, consider the input to the vertex shader and what happens if you declare a variable with an `in` storage qualifier. This marks the variable as an input to the vertex shader, which means that it is essentially an input to the OpenGL graphics pipeline. It is automatically filled in by the fixed-function vertex fetch stage. The variable becomes known as a *vertex attribute*.

Vertex attributes are how vertex data is introduced into the OpenGL pipeline. To declare a vertex attribute, you declare a variable in the vertex shader using the `in` storage qualifier. An example of this is shown in Listing 3.1, where we declare the variable `offset` as an input attribute.

```
#version 450 core

// 'offset' is an input vertex attribute
layout (location = 0) in vec4 offset;

void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;
}
```

Listing 3.1: Declaration of a vertex attribute

In Listing 3.1, we have added the variable **offset** as an input to the vertex shader. As it is an input to the first shader in the pipeline, it will be filled automatically by the vertex fetch stage. We can tell this stage what to fill the variable with by using one of the many variants of the vertex attribute functions, **glVertexAttrib*()**. The prototype for **glVertexAttrib4fv()**, which we use in this example, is

```
void glVertexAttrib4fv(GLuint index,
                       const GLfloat * v);
```

Here, the parameter `index` is used to reference the attribute and `v` is a pointer to the new data to put into the attribute. You may have noticed the `layout (location = 0)` code in the declaration of the **offset** attribute. This is a *layout qualifier*, which we have used to set the *location* of the vertex attribute to zero. This location is the value we'll pass in `index` to refer to the attribute.

Each time we call one of the **glVertexAttrib*()** functions (of which there are many), it will update the value of the vertex attribute that is passed to the vertex shader. We can use this approach to animate our one triangle. Listing 3.2 shows an updated version of our rendering function that updates the value of **offset** in each frame.

```
// Our rendering function
virtual void render(double currentTime)
{
    const GLfloat color[] = { (float)sin(currentTime) * 0.5f + 0.5f,
                              (float)cos(currentTime) * 0.5f + 0.5f,
                              0.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, color);

    // Use the program object we created earlier for rendering
    glUseProgram(rendering_program);

    GLfloat attrib[] = { (float)sin(currentTime) * 0.5f,
                         (float)cos(currentTime) * 0.6f,
                         0.0f, 0.0f };

    // Update the value of input attribute 0
    glVertexAttrib4fv(0, attrib);

    // Draw one triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Listing 3.2: Updating a vertex attribute

When we run the program with the rendering function of Listing 3.2, the triangle will move in a smooth oval shape around the window.

Passing Data from Stage to Stage

So far, you have seen how to pass data into a vertex shader by creating a vertex attribute using the `in` keyword, how to communicate with fixed-function blocks by reading and writing built-in variables such as `gl_VertexID` and `gl_Position`, and how to output data from the fragment shader using the `out` keyword. However, it's also possible to send your own data from shader stage to shader stage using the same `in` and `out` keywords. Just as you used the `out` keyword in the fragment shader to create the output variable to which it writes its color values, so you can also create an output variable in the vertex shader by using the `out` keyword. Anything you write to an output variable in one shader is sent to a similarly named variable declared with the `in` keyword in the subsequent stage. For example, if your vertex shader declares a variable called `vs_color` using the `out` keyword, it would match up with a variable named `vs_color` declared with the `in` keyword in the fragment shader stage (assuming no other stages were active in between).

If we modify our simple vertex shader as shown in Listing 3.3 to include `vs_color` as an output variable, and correspondingly modify our simple fragment shader to include `vs_color` as an input variable as shown in Listing 3.4, we can pass a value from the vertex shader to the fragment shader. Then, rather than outputting a hard-coded value, the fragment can simply output the color passed to it from the vertex shader.

```
#version 450 core

// 'offset' and 'color' are input vertex attributes
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// 'vs_color' is an output that will be sent to the next shader stage
out vec4 vs_color;

void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;

    // Output a fixed value for vs_color
    vs_color = color;
}
```

Listing 3.3: Vertex shader with an output

As you can see in Listing 3.3, we declare a second input to our vertex shader, `color` (this time at location 1), and write its value to the `vs_output` output. This is picked up by the fragment shader of Listing 3.4 and written to the framebuffer. This allows us to pass a color all the way from a vertex attribute that we can set with **`glVertexAttrib*()`** through the vertex shader, into the fragment shader, and out to the framebuffer. As a consequence, we can draw different-colored triangles!

```
#version 450 core

// Input from the vertex shader
in vec4 vs_color;

// Output to the framebuffer
out vec4 color;

void main(void)
{
   // Simply assign the color we were given by the vertex shader to our output
   color = vs_color;
}
```

Listing 3.4: Fragment shader with an input

Interface Blocks

Declaring interface variables one at a time is possibly the simplest way to communicate data between shader stages. However, in most nontrivial applications, you will likely want to communicate a number of different pieces of data between stages; these may include arrays, structures, and other complex arrangements of variables. To achieve this, we can group together a number of variables into an *interface block*. The declaration of an interface block looks a lot like a structure declaration, except that it is declared using the `in` or `out` keyword depending on whether it is an input to or output from the shader. An example interface block definition is shown in Listing 3.5.

```
#version 450 core

// 'offset' is an input vertex attribute
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// Declare VS_OUT as an output interface block
out VS_OUT
{
    vec4 color;     // Send color to the next stage
} vs_out;
```

```
void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;

    // Output a fixed value for vs_color
    vs_out.color = color;
}
```

Listing 3.5: Vertex shader with an output interface block

Note that the interface block in Listing 3.5 has both a block name (`VS_OUT`, uppercase) and an instance name (`vs_out`, lowercase). Interface blocks are matched between stages using the block name (`VS_OUT` in this case), but are referenced in shaders using the instance name. Thus, modifying our fragment shader to use an interface block gives the code shown in Listing 3.6.

```
#version 450 core

// Declare VS_OUT as an input interface block
in VS_OUT
{
    vec4 color;     // Send color to the next stage
} fs_in;

// Output to the framebuffer
out vec4 color;

void main(void)
{
    // Simply assign the color we were given by the vertex shader to our output
    color = fs_in.color;
}
```

Listing 3.6: Fragment shader with an input interface block

Matching interface blocks by block name but allowing block instances to have different names in each shader stage serves two important purposes. First, it allows the name by which you refer to the block to be different in each stage, thereby avoiding confusing things such as having to use `vs_out` in a fragment shader. Second, it allows interfaces to go from being single items to arrays when crossing between certain shader stages, such as the vertex and tessellation or geometry shader stages, as we will see in a short while. Note that interface blocks are only for moving data from shader stage to shader stage—you can't use them to group together inputs to the vertex shader or outputs from the fragment shader.

Tessellation

Tessellation is the process of breaking a high-order primitive (which is known as a *patch* in OpenGL) into many smaller, simpler primitives such as triangles for rendering. OpenGL includes a fixed-function, configurable tessellation engine that is able to break up quadrilaterals, triangles, and lines into a potentially large number of smaller points, lines, or triangles that can be directly consumed by the normal rasterization hardware further down the pipeline. Logically, the tessellation phase sits directly after the vertex shading stage in the OpenGL pipeline and is made up of three parts: the tessellation control shader, the fixed-function tessellation engine, and the tessellation evaluation shader.

Tessellation Control Shaders

The first of the three tessellation phases is the tessellation control shader (TCS; sometimes known as simply the control shader). This shader takes its input from the vertex shader and is primarily responsible for two things: the determination of the level of tessellation that will be sent to the tessellation engine, and the generation of data that will be sent to the tessellation evaluation shader that is run after tessellation has occurred.

Tessellation in OpenGL works by breaking down high-order surfaces known as *patches* into points, lines, or triangles. Each patch is formed from a number of *control points*. The number of control points per patch is configurable and set by calling **`glPatchParameteri()`** with `pname` set to `GL_PATCH_VERTICES` and `value` set to the number of control points that will be used to construct each patch. The prototype of **`glPatchParameteri()`** is

```
void glPatchParameteri(GLenum pname,
                       GLint value);
```

By default, the number of control points per patch is three. Thus, if this is what you want (as in our example application), you don't need to call it at all. The maximum number of control points that can be used to form a single patch is implementation defined, but is guaranteed to be at least 32.

When tessellation is active, the vertex shader runs once per control point, while the tessellation control shader runs in batches on groups of control points where the size of each batch is the same as the number of vertices per patch. That is, vertices are used as control points and the result of the vertex shader is passed in batches to the tessellation control shader as its input. The number of control points per patch can be changed such that the number of control points that is output by the tessellation control shader can differ from the number of control points that it consumes. The number of control points produced by the control shader is set using an output layout qualifier in the control shader's source code. Such a layout qualifier looks like this:

```
layout (vertices = N) out;
```

Here, `N` is the number of control points per patch. The control shader is responsible for calculating the values of the output control points and for setting the tessellation factors for the resulting patch that will be sent to the fixed-function tessellation engine. The output tessellation factors are written to the `gl_TessLevelInner` and `gl_TessLevelOuter` built-in output variables, whereas any other data that is passed down the pipeline is written to user-defined output variables (those declared using the `out` keyword, or the special built-in `gl_out` array) as normal.

Listing 3.7 shows a simple tessellation control shader. It sets the number of output control points to three (the same as the default number of input control points) using the `layout (vertices = 3) out;` layout qualifier, copies its input to its output (using the built-in variables `gl_in` and `gl_out`), and sets the inner and outer tessellation level to 5. Higher numbers would produce a more densely tessellated output, and lower numbers would yield a more coarsely tessellated output. Setting the tessellation factor to 0 will cause the whole patch to be thrown away.

The built-in input variable `gl_InvocationID` is used as an index into the `gl_in` and `gl_out` arrays. This variable contains the zero-based index of the control point within the patch being processed by the current invocation of the tessellation control shader.

```
#version 450 core

layout (vertices = 3) out;

void main(void)
{
    // Only if I am invocation 0 ...
    if (gl_InvocationID == 0)
```

```
    {
        gl_TessLevelInner[0] = 5.0;
        gl_TessLevelOuter[0] = 5.0;
        gl_TessLevelOuter[1] = 5.0;
        gl_TessLevelOuter[2] = 5.0;
    }
    // Everybody copies their input to their output
    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 3.7: Our first tessellation control shader

The Tessellation Engine

The tessellation engine is a fixed-function part of the OpenGL pipeline that takes high-order surfaces represented as patches and breaks them down into simpler primitives such as points, lines, or triangles. Before the tessellation engine receives a patch, the tessellation control shader processes the incoming control points and sets tessellation factors that are used to break down the patch. After the tessellation engine produces the output primitives, the vertices representing them are picked up by the tessellation evaluation shader. The tessellation engine is responsible for producing the parameters that are fed to the invocations of the tessellation evaluation shader, which it then uses to transform the resulting primitives and get them ready for rasterization.

Tessellation Evaluation Shaders

Once the fixed-function tessellation engine has run, it produces a number of output vertices representing the primitives it has generated. These are passed to the tessellation evaluation shader. The tessellation evaluation shader (TES; also called simply the evaluation shader) runs an invocation for each vertex produced by the tessellator. When the tessellation levels are high, the tessellation evaluation shader could run an extremely large number of times. For this reason, you should be careful with complex evaluation shaders and high tessellation levels.

Listing 3.8 shows a tessellation evaluation shader that accepts input vertices produced by the tessellator as a result of running the control shader shown in Listing 3.7. At the beginning of the shader is a layout qualifier that sets the tessellation mode. In this case, we selected the mode to be triangles. Other qualifiers, equal_spacing and cw, indicate that new vertices should be generated equally spaced along the tessellated polygon edges and that a clockwise vertex winding order should be used for the generated triangles. We will cover the other possible choices in the "Tessellation" section in Chapter 8.

The remainder of the shader assigns a value to gl_Position just like a vertex shader does. It calculates this using the contents of two more built-in variables. The first is gl_TessCoord, which is the *barycentric coordinate* of the vertex generated by the tessellator. The second is the gl_Position member of the gl_in[] array of structures. This matches the gl_out structure written to in the tessellation control shader given in Listing 3.7. This shader essentially implements pass-through tessellation. That is, the tessellated output patch is exactly the same shape as the original, incoming triangular patch.

```
#version 450 core

layout (triangles, equal_spacing, cw) in;

void main(void)
{
    gl_Position = (gl_TessCoord.x * gl_in[0].gl_Position +
                   gl_TessCoord.y * gl_in[1].gl_Position +
                   gl_TessCoord.z * gl_in[2].gl_Position);
}
```

Listing 3.8: Our first tessellation evaluation shader

To see the results of the tessellator, we need to tell OpenGL to draw only the outlines of the resulting triangles. To do this, we call **glPolygonMode()**, whose prototype is

```
void glPolygonMode(GLenum face,
                   GLenum mode);
```

The face parameter specifies which type of polygons we want to affect. Because we want to affect everything, we set it to GL_FRONT_AND_BACK. The other modes will be explained shortly. mode says how we want our polygons to be rendered. As we want to render in wireframe mode (i.e., lines), we set this to GL_LINE. The result of rendering our one triangle

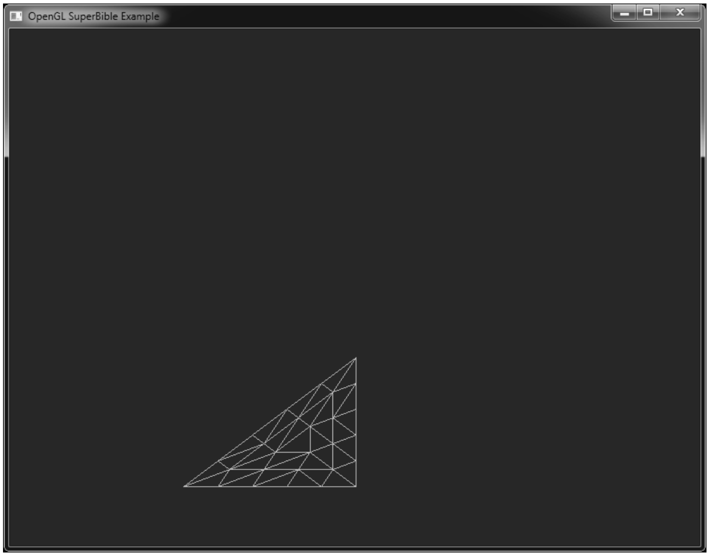


Figure 3.1: Our first tessellated triangle

example with tessellation enabled and the two shaders of Listing 3.7 and Listing 3.8 is shown in Figure 3.1.

Geometry Shaders

The geometry shader is logically the last shader stage in the front end, sitting after the vertex and tessellation stages and before the rasterizer. The geometry shader runs once per primitive and has access to all of the input vertex data for all of the vertices that make up the primitive being processed. The geometry shader is also unique among the shader stages in that it is able to increase or reduce the amount of data flowing through the pipeline in a programmatic way. Tessellation shaders can also increase or decrease the amount of work in the pipeline, but only implicitly by setting the tessellation level for the patch. Geometry shaders, in contrast, include two functions—`EmitVertex()` and `EndPrimitive()`—that explicitly produce vertices that are sent to primitive assembly and rasterization.

Another unique feature of geometry shaders is that they can change the primitive mode mid-pipeline. For example, they can take triangles as input and produce a bunch of points or lines as output, or even create triangles from independent points. An example geometry shader is shown in Listing 3.9.

```
#version 450 core

layout (triangles) in;
layout (points, max_vertices = 3) out;

void main(void)
{
    int i;

    for (i = 0; i < gl_in.length(); i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
}
```

Listing 3.9: Our first geometry shader

The shader shown in Listing 3.9 acts as another simple pass-through shader that converts triangles into points so that we can see their vertices. The first layout qualifier indicates that the geometry shader is expecting to see triangles as its input. The second layout qualifier tells OpenGL that the geometry shader will produce points and that the maximum number of points that each shader will produce will be three. In the `main` function, a loop runs through all of the members of the `gl_in` array, which is determined by calling its `.length()` function.

We actually know that the length of the array will be three because we are processing triangles and every triangle has three vertices. The outputs of the geometry shader are again similar to those of a vertex shader. In particular, we write to `gl_Position` to set the position of the resulting vertex. Next, we call `EmitVertex()`, which produces a vertex at the output of the geometry shader. Geometry shaders automatically call `EndPrimitive()` at the end of your shader, so calling this function explicitly is not necessary in this example. As a result of running this shader, three vertices will be produced and rendered as points.

By inserting this geometry shader into our simple one tessellated triangle example, we obtain the output shown in Figure 3.2. To create this image, we set the point size to 5.0 by calling **`glPointSize()`**. This makes the points large and highly visible.

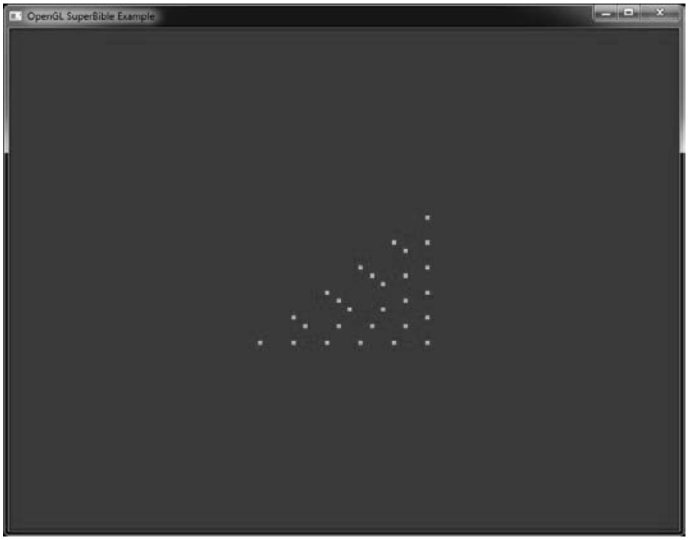


Figure 3.2: Tessellated triangle after adding a geometry shader

Primitive Assembly, Clipping, and Rasterization

After the front end of the pipeline has run (which includes vertex shading, tessellation, and geometry shading), a fixed-function part of the pipeline performs a series of tasks that take the vertex representation of our scene and convert it into a series of pixels, which in turn need to be colored and written to the screen. The first step in this process is primitive assembly, which is the grouping of vertices into lines and triangles. Primitive assembly still occurs for points, but it is trivial in that case.

Once primitives have been constructed from their individual vertices, they are *clipped* against the displayable region, which usually means the window or screen, but can also be a smaller area known as the *viewport*. Finally, the parts of the primitive that are determined to be potentially visible are sent to a fixed-function subsystem called the rasterizer. This block determines which pixels are covered by the primitive (point, line, or triangle) and sends the list of pixels on to the next stage—that is, fragment shading.

Clipping

As vertices exit the front end of the pipeline, their position is said to be in *clip space*. This is one of the many coordinate systems that can be used to represent positions. You may have noticed that the `gl_Position` variable that we have written to in our vertex, tessellation, and geometry shaders has a `vec4` type, and that the positions we have produced by writing to it are all four-component vectors. This is what is known as a *homogeneous* coordinate. The homogeneous coordinate system is used in projective geometry because much of the math ends up being simpler in homogeneous coordinate space than it does in regular Cartesian space. Homogeneous coordinates have one more component than their equivalent Cartesian coordinate, which is why our three-dimensional position vector is represented as a four-component variable.

Although the output of the front end is a four-component homogeneous coordinate, clipping occurs in Cartesian space. Thus, to convert from homogeneous coordinates to Cartesian coordinates, OpenGL performs a *perspective division*, which involves dividing all four components of the position by the last, w component. This has the effect of projecting the vertex from the homogeneous space to the Cartesian space, leaving w as 1.0. In all of the examples so far, we have set the w component of `gl_Position` as 1.0, so this division has not had any effect. When we explore projective geometry in a short while, we will discuss the effect of setting w to values other than 1.0.

After the projective division, the resulting position is in *normalized device space*. In OpenGL, the visible region of normalized device space is the volume that extends from -1.0 to 1.0 in the x and y dimensions and from 0.0 to 1.0 in the z dimension. Any geometry that is contained in this region may become visible to the user and anything outside of it should be discarded. The six sides of this volume are formed by planes in three-dimensional space. As a plane divides a coordinate space in two, the volumes on each side of the plane are called *half-spaces*.

Before passing primitives on to the next stage, OpenGL performs clipping by determining which side of each of these planes the vertices of each primitive lie on. Each plane effectively has an "outside" and an "inside." If a primitive's vertices all lie on the "outside" of any one plane, then the whole thing is thrown away. If all of primitive's vertices are on the "inside" of all the planes (and therefore inside the view volume), then it is passed through unaltered. Primitives that are partially visible (which means that they cross one of the planes) must be handled specially. More details about how this works is given in the "Clipping" section in Chapter 7.

Viewport Transformation

After clipping, all of the vertices of the geometry have coordinates that lie between -1.0 and 1.0 in the x and y dimensions. Along with a z coordinate that lies between 0.0 and 1.0, these are known as normalized device coordinates. However, the window that you're drawing to has coordinates that usually[1] start from $(0, 0)$ at the bottom left and range to $(w-1, h-1)$, where w and h are the width and height of the window in pixels, respectively. To place your geometry into the window, OpenGL applies the *viewport transform*, which applies a scale and offset to the vertices' normalized device coordinates to move them into *window coordinates*. The scale and bias to apply are determined by the viewport bounds, which you can set by calling **glViewport()** and **glDepthRange()**. Their prototypes are

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

and

```
void glDepthRange(GLdouble nearVal, GLdouble farVal);
```

This transform takes the following form:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ \frac{f-n}{2} z_d + \frac{n+f}{2} \end{pmatrix}$$

Here, x_w, y_w, and z_w are the resulting coordinates of the vertex in window space, and x_d, y_d, and z_d are the incoming coordinates of the vertex in normalized device space. p_x and p_y are the width and height of the viewport in pixels, and n and f are the near and far plane distances in the z coordinate, respectively. Finally, o_x, o_y, and o_z are the origins of the viewport.

Culling

Before a triangle is processed further, it may be optionally passed through a stage called *culling*, which determines whether the triangle faces toward or away from the viewer and can decide whether to actually go ahead and draw it based on the result of this computation. If the triangle faces toward the viewer, then it is considered to be *front-facing*; otherwise, it is said to be *back-facing*. It is very common to discard triangles that are back-facing because when an object is closed, any back-facing triangle will be hidden by another front-facing triangle.

1. It's possible to change the coordinate convention such that the $(0, 0)$ origin is at the upper-left corner of the window, which matches the convention used in some other graphics systems.

To determine whether a triangle is front- or back-facing, OpenGL will determine its *signed* area in window space. One way to determine the area of a triangle is to take the cross product of two of its edges. The equation for this is

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i$$

Here, x_w^i and y_w^i are the coordinates of the ith vertex of the triangle in window space and $i \oplus 1$ is $(i+1)$ mod 3. If the area is positive, then the triangle is considered to be front-facing; if it is negative, then it is considered to be back-facing. The sense of this computation can be reversed by calling **`glFrontFace()`** with `dir` set to either `GL_CW` or `GL_CCW` (where `CW` and `CCW` stand for clockwise and counterclockwise, respectively). This is known as the *winding order* of the triangle, and the clockwise or counterclockwise terms refer to the order in which the vertices appear in window space. By default, this state is set to `GL_CCW`, indicating that triangles whose vertices are in counterclockwise order are considered to be front-facing and those whose vertices are in clockwise order are considered to be back-facing. If the state is `GL_CW`, then a is simply negated before being used in the culling process. Figure 3.3 shows this pictorially for the purpose of illustration.

Once the direction that the triangle is facing has been determined, OpenGL is capable of discarding either front-facing, back-facing, or even both types of triangles. By default, OpenGL will render all triangles, regardless of which way they face. To turn on culling, call **`glEnable()`** with `cap` set to `GL_CULL_FACE`. When you enable culling, OpenGL will cull back-facing triangles by default. To change which types of triangles are

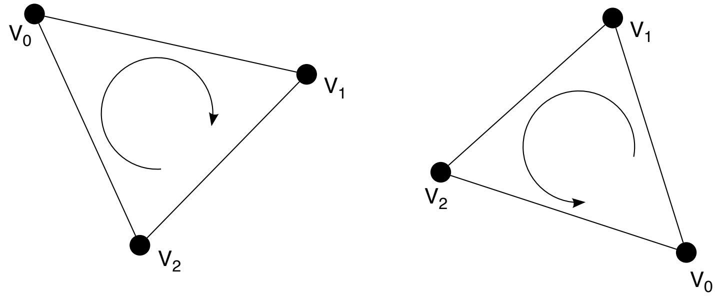


Figure 3.3: Clockwise (left) and counterclockwise (right) winding order

culled, call **`glCullFace()`** with `face` set to `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

As points and lines don't have any geometric area,[2] this facing calculation doesn't apply to them and they can't be culled at this stage.

Rasterization

Rasterization is the process of determining which fragments might be covered by a primitive such as a line or a triangle. There are myriad algorithms for doing this, but most OpenGL systems will settle on a half-space–based method for triangles, as it lends itself well to parallel implementation. Essentially, OpenGL will determine a bounding box for the triangle in window coordinates and test every fragment inside it to determine whether it is inside or outside the triangle. To do this, it treats each of the triangle's three edges as a half-space that divides the window in two.

Fragments that lie on the interior of all three edges are considered to be inside the triangle and fragments that lie on the exterior of any of the three edges are considered to be outside the triangle. Because the algorithm to determine which side of a line a point lies on is relatively simple and is independent of anything besides the position of the line's endpoints and of the point being tested, many tests can be performed concurrently, providing the opportunity for massive parallelism.

Fragment Shaders

The fragment[3] shader is the last programmable stage in OpenGL's graphics pipeline. This stage is responsible for determining the color of each fragment before it is sent to the framebuffer for possible composition into the window. After the rasterizer processes a primitive, it produces a list of fragments that need to be colored and passes this list to the fragment

2. Obviously, once they are rendered to the screen, points and lines have area; otherwise, we wouldn't be able to see them. However, this area is artificial and can't be calculated directly from their vertices.

3. The term *fragment* is used to describe an element that may ultimately contribute to the final color of a pixel. The pixel may not end up being the color produced by any particular invocation of the fragment shader due to a number of other effects such as depth or stencil tests, blending, and multi-sampling, all of which will be covered later in the book.

shader. Here, an explosion in the amount of work in the pipeline occurs, as each triangle could produce hundreds, thousands, or even millions of fragments.

Listing 2.4 in Chapter 2 contains the source code of our first fragment shader. It's an extremely simple shader that declares a single output and then assigns a fixed value to it. In a real-world application, the fragment shader would normally be substantially more complex and be responsible for performing calculations related to lighting, applying materials, and even determining the depth of the fragment. Available as input to the fragment shader are several built-in variables such as `gl_FragCoord`, which contains the position of the fragment within the window. It is possible to use these variables to produce a unique color for each fragment.

Listing 3.10 provides a shader that derives its output color from `gl_FragCoord`. Figure 3.4 shows the output of running our original single-triangle program with this shader installed.

```
#version 450 core

out vec4 color;

void main(void)
{
    color = vec4(sin(gl_FragCoord.x * 0.25) * 0.5 + 0.5,
                 cos(gl_FragCoord.y * 0.25) * 0.5 + 0.5,
                 sin(gl_FragCoord.x * 0.15) * cos(gl_FragCoord.y * 0.15),
                 1.0);
}
```

Listing 3.10: Deriving a fragment's color from its position

As you can see, the color of each pixel in Figure 3.4 is now a function of its position and a simple screen-aligned pattern has been produced. The shader of Listing 3.10 created the checkered patterns in the output.

The `gl_FragCoord` variable is one of the built-in variables available to the fragment shader. However, just as with other shader stages, we can define our own inputs to the fragment shader, which will be filled in based on the outputs of whichever stage is last before rasterization. For example, if we have a simple program with only a vertex shader and fragment shader in it, we can pass data from the fragment shader to the vertex shader.

The inputs to the fragment shader are somewhat unlike inputs to other shader stages, in that OpenGL *interpolates* their values across the primitive

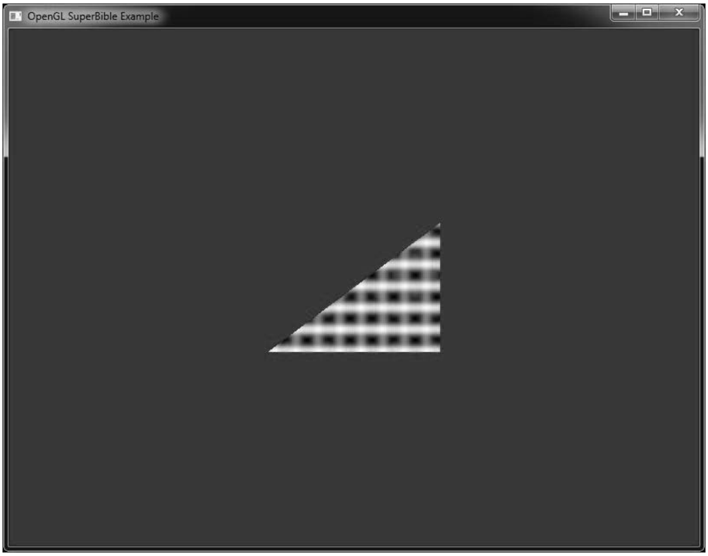


Figure 3.4: Result of Listing 3.10

that's being rendered. To demonstrate, we take the vertex shader of Listing 3.3 and modify it to assign a different, fixed color for each vertex, as shown in Listing 3.11.

```
#version 450 core

// 'vs_color' is an output that will be sent to the next shader stage
out vec4 vs_color;

void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));
    const vec4 colors[] = vec4[3](vec4( 1.0, 0.0, 0.0, 1.0),
                                  vec4( 0.0, 1.0, 0.0, 1.0),
                                  vec4( 0.0, 0.0, 1.0, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;

    // Output a fixed value for vs_color
    vs_color = color[gl_VertexID];
}
```

Listing 3.11: Vertex shader with an output

As you can see, in Listing 3.11 we added a second constant array that contains colors and index into it using `gl_VertexID`, writing its content to the `vs_color` output. In Listing 3.12 we modify our simple fragment shader to include the corresponding input and write its value to the output.

```
#version 450 core

// 'vs_color' is the color produced by the vertex shader
in vec4 vs_color;

out vec4 color;

void main(void)
{
    color = vs_color;
}
```

Listing 3.12: Deriving a fragment's color from its position

The result of using this new pair of shaders is shown in Figure 3.5. As you can see, the color changes smoothly across the triangle.

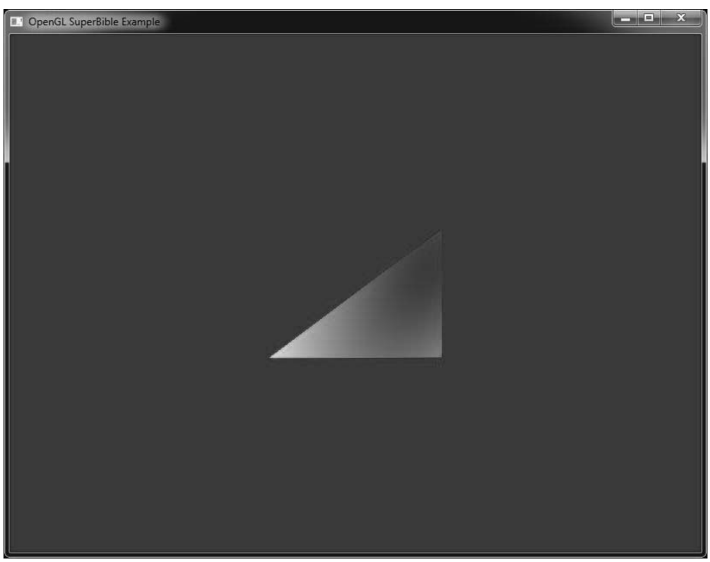


Figure 3.5: Result of Listing 3.12

Framebuffer Operations

The framebuffer is the last stage of the OpenGL graphics pipeline. It can represent the visible content of the screen and a number of additional regions of memory that are used to store per-pixel values other than color. On most platforms, this means the window you see on your desktop (or possibly the whole screen if your application covers it), which is owned by the operating system (or windowing system to be more precise). The framebuffer provided by the windowing system is known as the default framebuffer, but it is possible to provide your own if you wish to do things like render into off-screen areas. The state held by the framebuffer includes information such as where the data produced by your fragment shader should be written, what the format of that data should be, and so on. This state is stored in a *framebuffer object*. Also considered part of the framebuffer, but not stored per framebuffer object, is the pixel operation state.

Pixel Operations

After the fragment shader has produced an output, several things may happen to the fragment before it is written to the window, such as a determination of whether it even belongs in the window. Each of these things may be turned on or off by your application. The first thing that could happen is the *scissor test*, which tests your fragment against a rectangle that you can define. If it's inside the rectangle, then it will be processed further; if it's outside, it will be thrown away.

Next comes the *stencil test*. This compares a reference value provided by your application with the contents of the stencil buffer, which stores a single[4] value per pixel. The content of the stencil buffer has no particular semantic meaning and can be used for any purpose.

After the stencil test has been performed, the *depth test* is performed. The depth test is an operation that compares the fragment's z coordinate against the contents of the *depth buffer*. The depth buffer is a region of memory that, like the stencil buffer, is part of the framebuffer with enough space for a single value for each pixel; it contains the depth (which is related to distance from the viewer) of each pixel.

4. It's possible for a framebuffer to store multiple depth, stencil, or color values per pixel when a technique called *multi-sampling* is employed. We'll dig into this later in the book.

Normally, the values in the depth buffer range from 0 to 1, with 0 being the closest possible point in the depth buffer and 1 being the furthest possible point in the depth buffer. To determine whether a fragment is closer than other fragments that have already been rendered in the same place, OpenGL can compare the z component of the fragment's window-space coordinate against the value already in the depth buffer. If this value is less than what's already there, then the fragment is visible. The sense of this test can also be changed. For example, you can ask OpenGL to let fragments through that have a z coordinate that is greater than, equal to, or not equal to the content of the depth buffer. The result of the depth test also affects what OpenGL does to the stencil buffer.

Next, the fragment's color is sent to either the blending or logical operation stage, depending on whether the framebuffer is considered to store floating-point, normalized, or integer values. If the content of the framebuffer is either floating-point or normalized integer values, then blending is applied. Blending is a highly configurable stage in OpenGL and will be covered in detail in its own section.

In short, OpenGL is capable of using a wide range of functions that take components of the output of your fragment shader and of the current content of the framebuffer and calculate new values that are written back to the framebuffer. If the framebuffer contains unnormalized integer values, then logical operations such as logical AND, OR, and XOR can be applied to the output of your shader and the value currently in the framebuffer to produce a new value that will be written back into the framebuffer.

Compute Shaders

The first sections of this chapter describe the *graphics pipeline* in OpenGL. However, OpenGL also includes the *compute shader* stage, which can almost be thought of as a separate pipeline that runs indepdendently of the other graphics-oriented stages.

Compute shaders are a way of getting at the computational power possessed by the graphics processor in the system. Unlike the graphics-centric vertex, tessellation, geometry, and fragment shaders, compute shaders could be considered as a special, single-stage pipeline all on their own. Each compute shader operates on a single unit of work known as a *work item*; these items are, in turn, collected together into small groups called *local workgroups*. Collections of these workgroups can be sent into OpenGL's compute pipeline to be processed. The compute shader doesn't have any fixed inputs or outputs besides a handful of built-in variables to tell the shader which item it is working on. All processing performed by a compute shader is explicitly written to memory by the shader itself, rather than being consumed by a subsequent pipeline stage. A very basic compute shader is shown in Listing 3.13.

```
#version 450 core

layout (local_size_x = 32, local_size_y = 32) in;

void main(void)
{
    // Do nothing
}
```

Listing 3.13: Simple do-nothing compute shader

Compute shaders are otherwise just like any other shader stage in OpenGL. To compile one, you create a shader object with the type `GL_COMPUTE_SHADER`, attach your GLSL source code to it with **`glShaderSource()`**, compile it with **`glCompileShader()`**, and then link it into a program with **`glAttachShader()`** and **`glLinkProgram()`**. The result is a program object with a compiled compute shader in it that can be launched to do work for you.

The shader in Listing 3.13 tells OpenGL that the size of the local workgroup will be 32 by 32 work items, but then proceeds to do nothing. To create a compute shader that actually does something useful, you need to know a bit more about OpenGL—so we'll revisit this topic later in the book.

Using Extensions in OpenGL

All of the examples shown in this book so far have relied on the core functionality of OpenGL. However, one of OpenGL's greatest strengths is that it can be extended and enhanced by hardware manufacturers, operating system vendors, and even publishers of tools and debuggers. Extensions can have many different effects on OpenGL functionality.

An extension is any addition to a core version of OpenGL. Extensions are listed in the OpenGL extension registry[5] on the OpenGL Web site. These

5. Find the OpenGL extension registry at `http://www.opengl.org/registry/`.

extensions are written as a list of differences from a particular version of the OpenGL specification, and note what that version of OpenGL is. That means the text of the extensions describes how the core OpenGL specification must be changed if the extension is supported. However, popular and generally useful extensions are normally "promoted" into the core versions of OpenGL; thus, if you are running the latest and greatest version of OpenGL, there might not be that many extensions that are interesting but not part of the core profile. A complete list of the extensions that were promoted to each version of OpenGL and a brief synopsis of what they do is included in Appendix C, "OpenGL Features and Versions."

There are three major classifications of extensions: vendor, EXT, and ARB. Vendor extensions are written and implemented on one vendor's hardware. Initials representing the specific vendor are usually part of the extension name—"AMD" for Advanced Micro Devices or "NV" for NVIDIA, for example. It is possible that more than one vendor might support a specific vendor extension, especially if it becomes widely accepted. EXT extensions are written together by two or more vendors. They often start their lives as vendor-specific extensions, but if another vendor is interested in implementing the extension, perhaps with minor changes, it may collaborate with the original authors to produce an EXT version. ARB extensions are an official part of OpenGL because they are approved by the OpenGL governing body, the Architecture Review Board (ARB). These extensions are often supported by most or all major hardware vendors and may also have started out as vendor or EXT extensions.

This extension process may sound confusing at first. Hundreds of extensions currently are available! But new versions of OpenGL are often constructed from extensions programmers have found useful. In this way each extension gets its time in the sun. The ones that shine can be promoted to core; the ones that are less useful are not considered. This "natural selection" process helps to ensure only the most useful and important new features make it into a core version of OpenGL.

A useful tool to determine which extensions are supported in your computer's OpenGL implementation is Realtech VR's OpenGL Extensions Viewer. It is freely available from the Realtech VR Web site (see Figure 3.6).

Enhancing OpenGL with Extensions

Before using any extensions, you *must* make sure that they're supported by the OpenGL implementation that your application is running on. To find

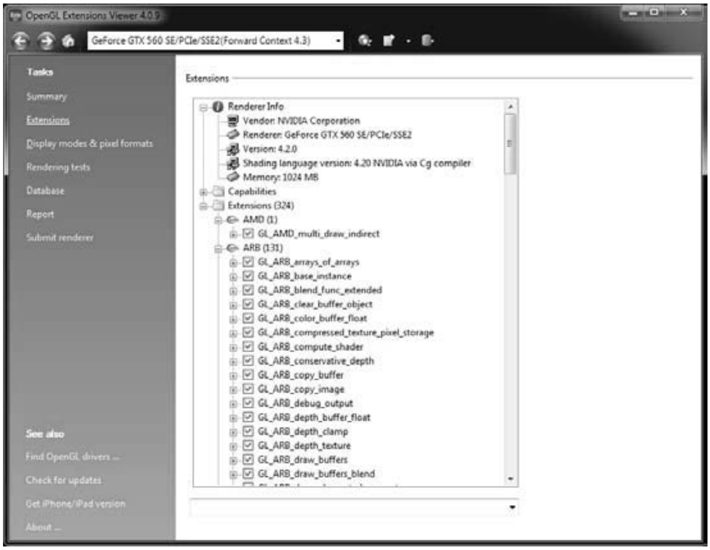


Figure 3.6: Realtech VR's OpenGL Extensions Viewer

out which extensions OpenGL supports, there are two functions that you can use. First, to determine the *number* of supported extensions, you can call **glGetIntegerv()** with the GL_NUM_EXTENSIONS parameter. Next, you can find out the name of each of the supported extensions by calling

```
const GLubyte* glGetStringi(GLenum name,
                            GLuint index);
```

You should pass GL_EXTENSIONS as the name parameter, and a value between 0 and 1 less than the number of supported extensions in index. The function returns the name of the extension as a string. To see if a specific extension is supported, you can simply query the number of extensions, and then loop through each supported extension and compare its name to the one you're looking for. The book's source code comes with a simple function that does this for you. **sb7IsExtensionSupported()** has the prototype

```
int sb7IsExtensionSupported(const char * extname);
```

This function is declared in the <sb7ext.h> header, takes the name of an extension, and returns non-zero if it is supported by the current OpenGL context and zero if it is not. Your application should always check for support for extensions you wish to use before using them.

Extensions generally add to OpenGL in some combination of four different ways:

- They can make things legal that weren't before, by simply removing restrictions from the OpenGL specification.
- They can add tokens or extend the range of values that can be passed as parameters to existing functions.
- They can extend GLSL to add functionality, built-in functions, variables, or data types.
- They can add entirely new functions to OpenGL itself.

In the first case, where things that once were considered errors no longer are, your application doesn't need to do anything besides start using the newly allowed behavior (once you have determined that the extension is supported, of course). Likewise, for the second case, you can just start using the new token values in the relevant functions, presuming that you have their values. The values of the tokens are in the extension specifications, so you can look them up there if they are not included in your system's header files.

To enable use of extensions in GLSL, you must first include a line at the beginning of shaders that use them to tell the compiler that you're going to need their features. For example, to enable the hypothetical `GL_ABC_foobar_feature` extension in GLSL, include the following in the beginning of your shader:

```
#extension GL_ABC_foobar_feature : enable
```

This tells the compiler that you intend to use the extension in your shader. If the compiler knows about the extension, it will let you compile the shader, even if the underlying hardware doesn't support the feature. If this is the case, the compiler should issue a warning if it sees that the extension is actually being used. Typically, extensions to GLSL will add preprocessor tokens to indicate their presence. For example, `GL_ABC_foobar_feature` will implicitly include

```
#define GL_ABC_foobar_feature 1
```

This means that you could write code such as

```
#if GL_ABC_foobar_feature
    // Use functions from the foobar extension
#else
```

```
    // Emulate or otherwise work around the missing functionality
#endif
```

This allows you to conditionally compile or execute functionality that is part of an extension that may or may not be supported by the underlying OpenGL implementation. If your shader absolutely requires support for an extension and will not work at all without it, you can instead include this more assertive code:

```
#extension GL_ABC_foobar_feature : require
```

If the OpenGL implementation does not support the `GL_ABC_foobar_feature` extension, then it will fail to compile the shader and report an error on the line including the **`#extension`** directive. In effect, GLSL extensions are opt-in features, and applications must[6] tell compilers up front which extensions they intend to use.

Next we come to extensions that introduce new functions to OpenGL. On most platforms, you don't have direct access to the OpenGL driver and extension functions don't just magically appear as available to your applications to call. Rather, you must ask the OpenGL driver for a *function pointer* that represents the function you want to call. Function pointers are generally declared in two parts; the first is the definition of the function pointer type, and the second is the function pointer variable itself. Consider this code as an example:

```
typedef void
(APIENTRYP PFNGLDRAWTRANSFORMFEEDBACKPROC) (GLenum mode,
                                            GLuint id);
PFNGLDRAWTRANSFORMFEEDBACKPROC glDrawTransformFeedback = NULL;
```

This declares the `PFNGLDRAWTRANSFORMFEEDBACKPROC` type as a pointer to a function taking `GLenum` and `GLuint` parameters. Next, it declares the `glDrawTransformFeedback` variable as an instance of this type. In fact, on many platforms, the declaration of the **`glDrawTransformFeedback()`** function is actually just like this. This seems pretty complicated, but fortunately the following header files include declarations of all of the

6. In practice, many implementations enable functionality included in some extensions by default and don't require that your shaders include these directives. However, if you rely on this behavior, your application is likely to not work on other OpenGL drivers. Because of this risk, you should always explicitly enable the extensions that you plan to use.

function prototypes, function pointer types, and token values introduced by all registered OpenGL extensions:

```
#include <glext.h>
#include <glxext.h>
#include <wglext.h>
```

These files can be found at the OpenGL extension registry Web site. The `glext.h` header contains both standard OpenGL extensions and many vendor-specific OpenGL extensions, the `wglext.h` header contains a number of extensions that are Windows specific, and the `glxext.h` header contains definitions that are X specific (X is the windowing system used on Linux and many other UNIX derivatives and implementations).

The method for querying the address of extension functions is actually platform specific. The book's application framework wraps up these intricacies into a handy function that is declared in the `<sb7ext.h>` header file. The function **`sb7GetProcAddress()`** has this prototype:

```
void * sb7GetProcAddress(const char * funcname);
```

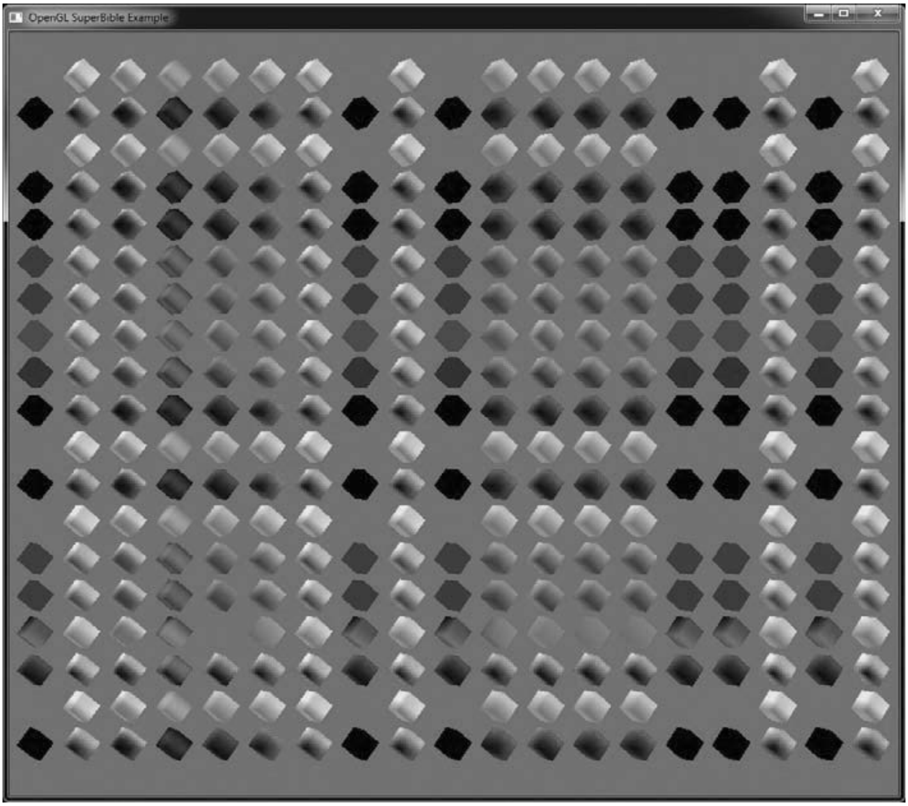
Here, `funcname` is the name of the extension function that you wish to use. The return value is the address of the function, if it's supported, and `NULL` otherwise. Even if OpenGL returns a valid function pointer for a function that's part of the extension you want to use, that doesn't mean the extension is present. Sometimes the same function is part of more than one extension, and sometimes vendors ship drivers with partial implementations of extensions present. Always check for support for extensions using the official mechanisms or the **`sb7IsExtensionSupported()`** function.

Summary

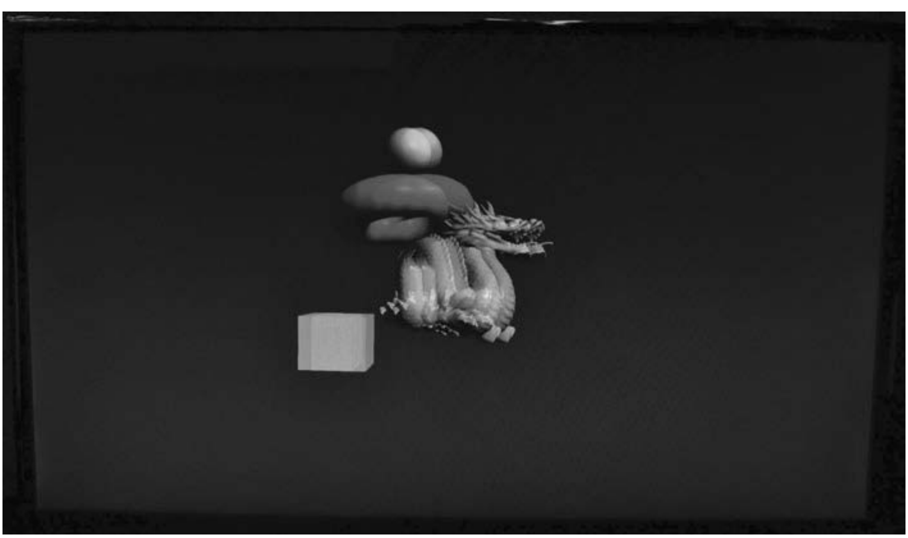
In this chapter, you have taken a whirlwind trip down OpenGL's graphics pipeline. You have been (very) briefly introduced to each major stage and have created a program that uses each one of them, if only to do nothing impressive. We've glossed over or even neglected to mention several useful features of OpenGL with the intention of getting you from zero to rendering in as few pages as possible. You've also seen how OpenGL's pipeline and functionality can be enhanced by using extensions, which some of the examples later in the book will rely on. Over the next few chapters, you'll learn more fundamentals of computer graphics and of OpenGL, and then we'll take a second trip down the pipeline, dig deeper into the topics from this chapter, and get into some of the things we skipped in this preview of what OpenGL can do.

Index

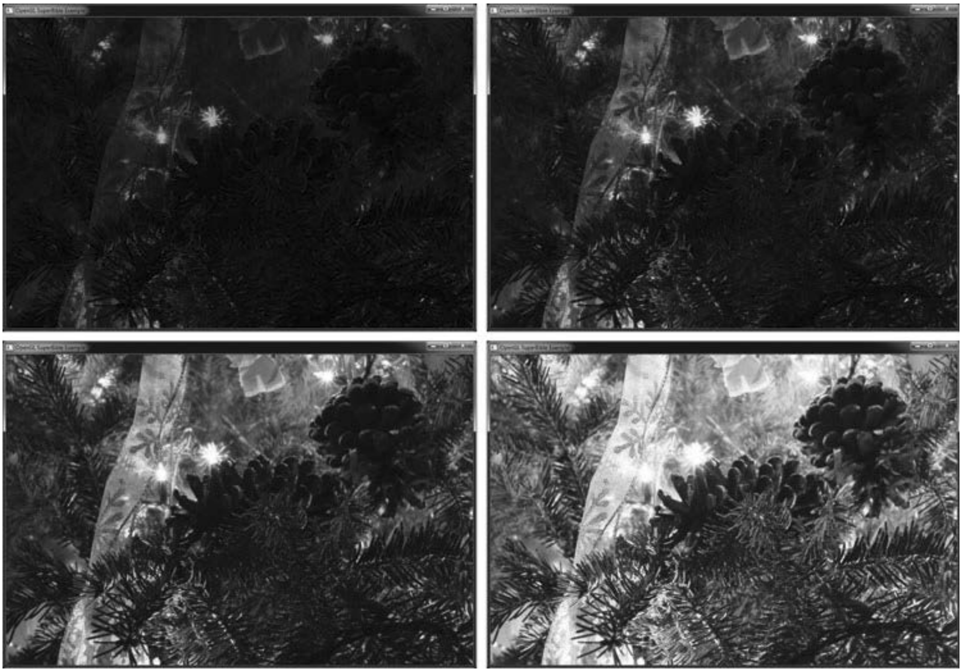
This page intentionally left blank



Color Plate 1: All possible combinations of blend function

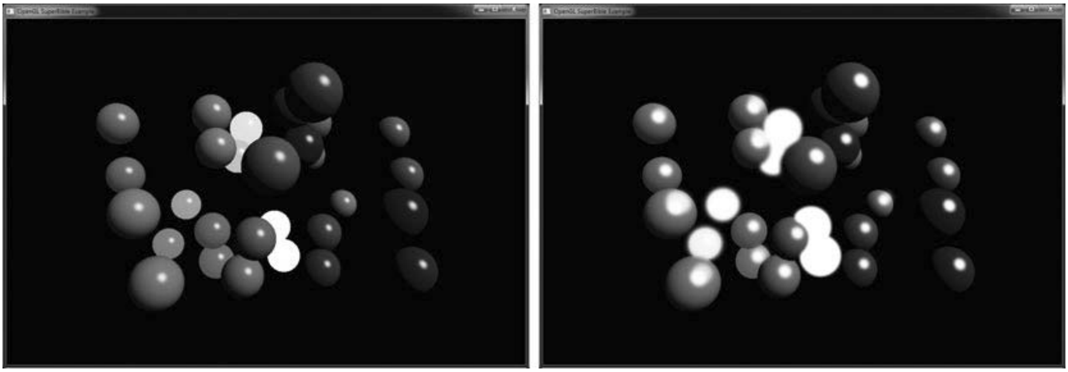


Color Plate 2: Rendering to a stereo display

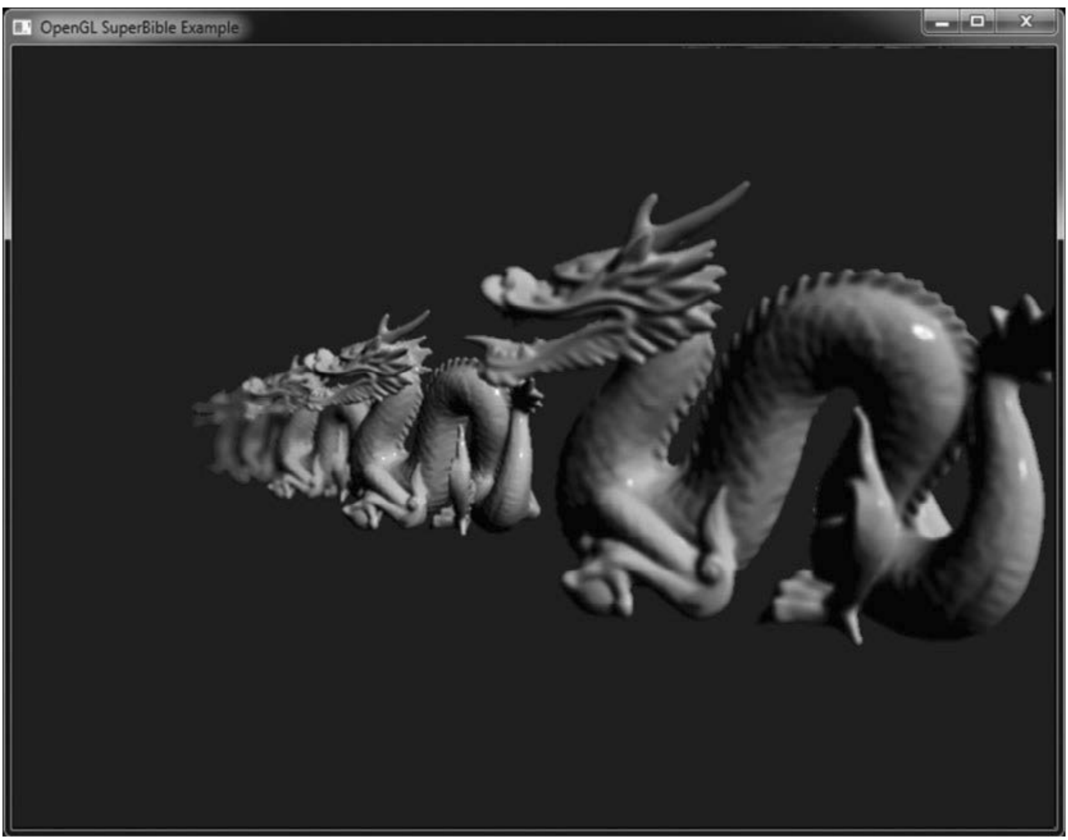


Color Plate 3: Different views of an HDR image

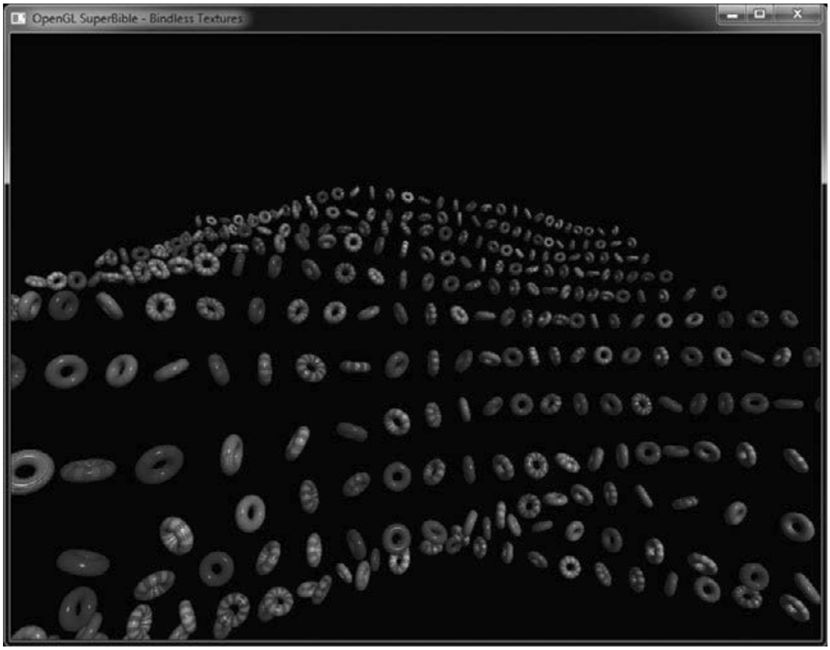
Color Plate 4: Adaptive tone mapping



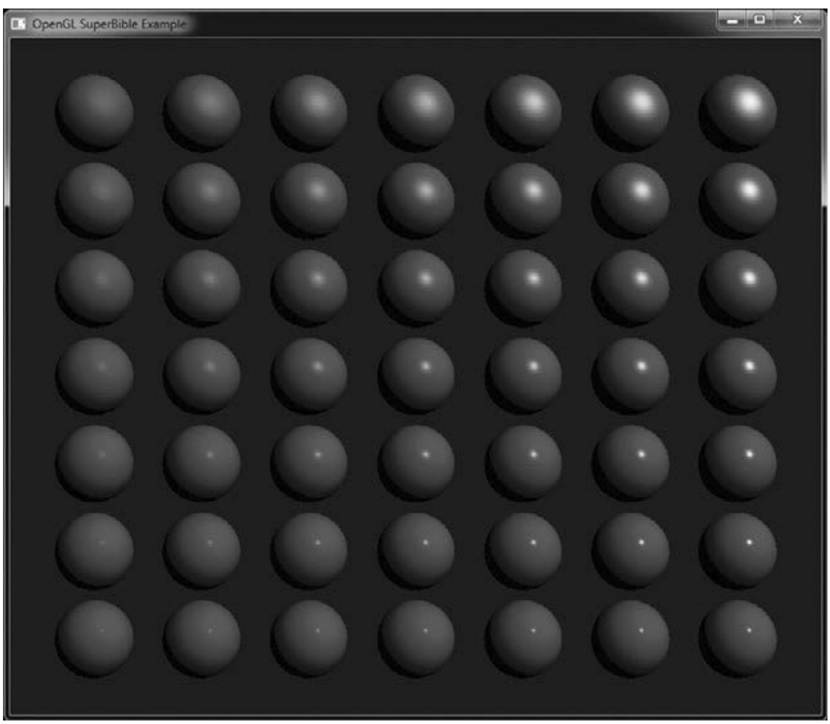
Color Plate 5: Bloom filtering: no bloom (left) and bloom (right)



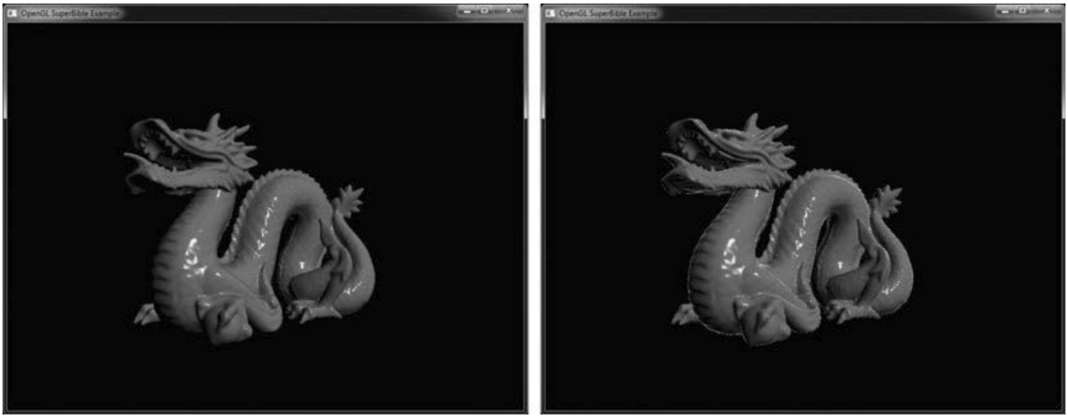
Color Plate 6: Depth of field applied to an image



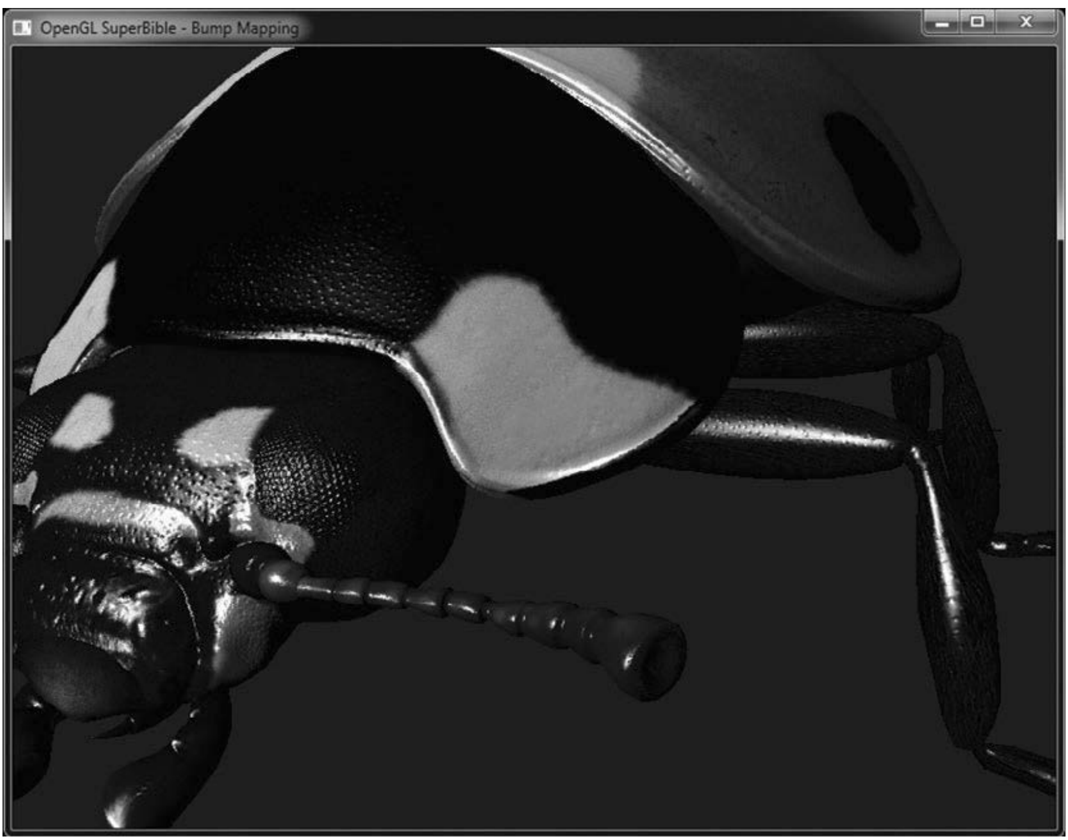
Color Plate 7: Output of bindless texture example



Color Plate 8: Varying specular parameters of a material



Color Plate 9: Result of rim lighting example

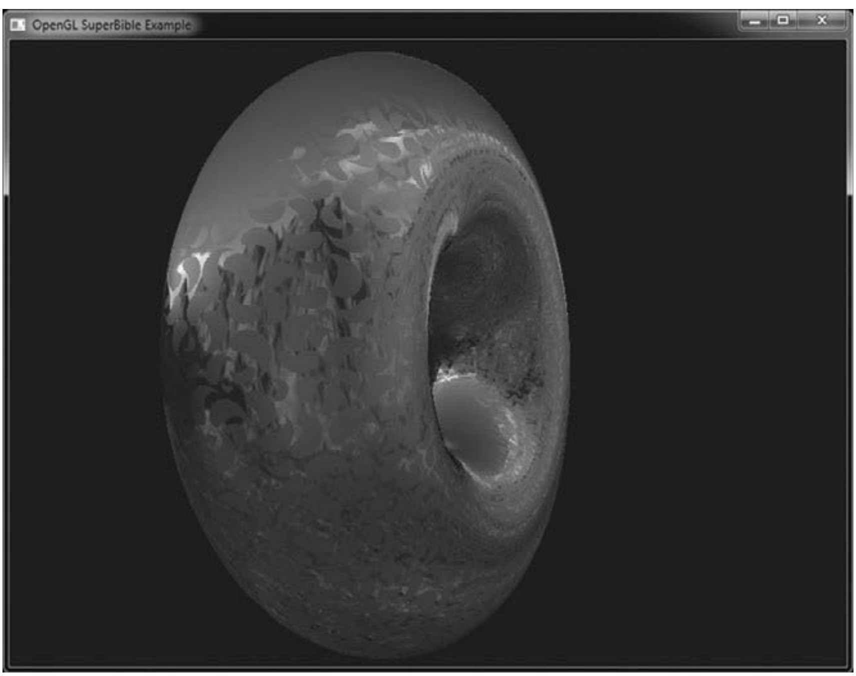


Color Plate 10: Normal mapping in action

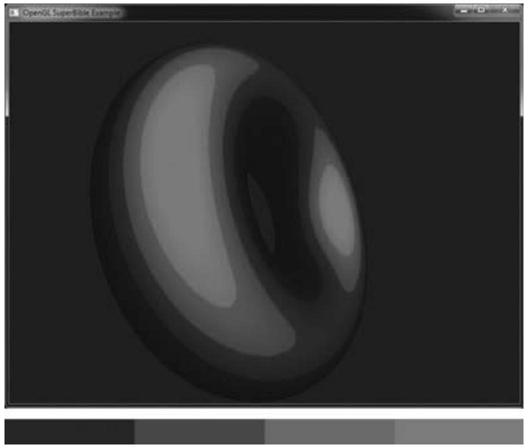
Color Plate 11: Depth of field applied in a photograph

Color Plate 12: A selection of spherical environment maps

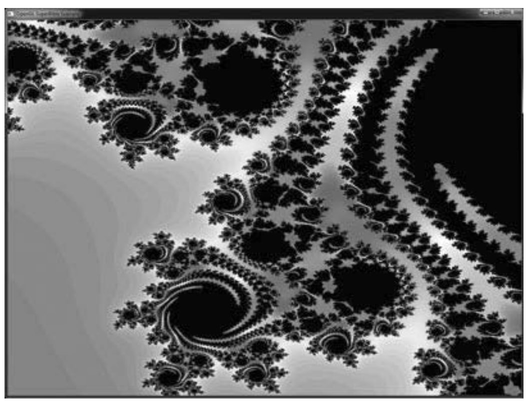
Color Plate 13: A golden environment-mapped dragon



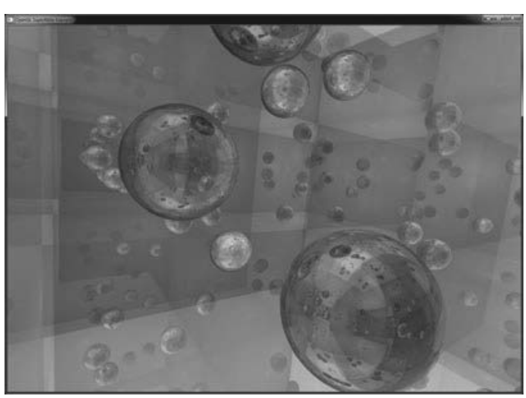
Color Plate 14: Result of per-pixel gloss example



Color Plate 15: Toon shading output with color ramp



Color Plate 16: Real-time rendering of the Julia set



Color Plate 17: Ray tracing with four bounces