Computer Graphics Programming in OpenGL with C++

Second Edition

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and its companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, including the companion files, but does not give you the right of ownership to any of the textual content in the book / files or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion files, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or companion files, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state and might not apply to the purchaser of this product.

The companion files are available for downloading by writing to the publisher at info@merclearning.com.

Computer Graphics Programming in OpenGL with C++

Second Edition

V. Scott Gordon, Ph.D.
California State University, Sacramento

John Clevenger, Ph.D.
California State University, Sacramento

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

Computer Graphics Programming in OpenGL with C++, Second Edition.
V. Scott Gordon & John Clevenger.
ISBN: 978-1-68392-672-6

Library of Congress Control Number: 2020946880

202122321 Printed on acid-free paper in the United States of America.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223 (toll free). Digital versions of our titles are available at: www.academiccourseware.com and other e-vendors. *All companion files are available by writing to the publisher at info@merclearning.com.*

Contents

Preface

This book is designed primarily as a textbook for a typical computer science undergraduate course in OpenGL 3D graphics programming. However, we have also endeavored to create a text that could be used to teach oneself, without an accompanying course. With both of those aims in mind, we have tried to explain things as clearly and as simply as we can. All of the programming examples are stripped down and simplified as much as possible, but they are still complete so that the reader may run them all as presented.

One of the things that we hope is unique about this book is that we have strived to make it accessible to someone new to 3D graphics programming. While there is by no means a lack of information available on the topic—quite the contrary—many students are initially overwhelmed. This text is our attempt to write the book we wish we had had when we were starting out, with step-by-step explanations of the basics, progressing in an organized manner up through advanced topics. We considered titling the book "shader programming made easy"; however, we don't think that there really is any way of making shader programming "easy." We hope that we have come close.

This book teaches OpenGL programming in *C++*. There are several advantages to learning graphics programming in C++:

- OpenGL's native language is C, so a C++ program can make direct OpenGL function calls.
- OpenGL applications written in C++ typically exhibit very high performance.
- C++ offers modern programming constructs (classes, polymorphism, etc.) not available in C.
- C++ is a popular language choice for using OpenGL, and a large number of instructional resources for OpenGL are available in C++.

It is worth mentioning that there do exist other language bindings for OpenGL. Popular alternatives exist for Java, C#, Python, and many others. This textbook focuses only on C++.

Another thing that makes this book unique is that it has a "sister" textbook: *Computer Graphics Programming in OpenGL with Java 2/E*. The two books are organized in lockstep, with the same chapter and section numbers and topics, figures, exercises, and theoretical descriptions. Wherever possible, the code is organized similarly. Of course, the use of C++ versus Java leads to considerable programming differences (although all of the shader code is identical). Still, we believe that we have provided virtually identical learning paths, even allowing a student to choose either option within a single classroom.

An important point of clarification is that there exist both different *versions* of OpenGL (briefly discussed later) and different *variants* of OpenGL. For example, in addition to "standard OpenGL" (sometimes called "desktop OpenGL"), there exists a variant called "OpenGL ES," which is tailored for development of *embedded systems* (hence the "ES"). "Embedded systems" include devices such as mobile phones, game consoles, automobiles, and industrial control systems. OpenGL ES is mostly a subset of standard OpenGL, eliminating a large number of operations that are typically not needed for embedded systems. OpenGL ES also adds some additional functionality, typically application-specific operations for particular target environments. This book focuses on standard OpenGL.

Yet another variant of OpenGL is called "WebGL." Based on OpenGL ES, WebGL is designed to support the use of OpenGL in web browsers. WebGL allows an application to use JavaScript[1] to invoke OpenGL ES operations, which makes it easy to embed OpenGL graphics into standard HTML (web) documents. Most modern web browsers support WebGL, including Apple Safari, Google Chrome, Microsoft Edge, Microsoft Internet Explorer, Mozilla Firefox, and Opera. Since web programming is outside the scope of this book, we will not cover any WebGL specifics. Note however that because WebGL is based on OpenGL ES, which in turn is based on standard OpenGL, much of what *is* covered in this book can be transferred directly to learning about these OpenGL variants.

The very topic of 3D graphics lends itself to impressive, even beautiful images. Indeed, many popular textbooks on the topic are filled with breathtaking scenes, and it is enticing to leaf through their galleries. While we acknowledge the motivational utility of such examples, our aim is to teach, not to impress. The images in this book are simply

---

1 JavaScript is a scripting language that can be used to embed code in webpages. It has strong similarities to Java, but also many important differences.

the outputs of the example programs, and since this is an introductory text, the resulting scenes are unlikely to impress an expert. However, the techniques presented do constitute the foundational elements for producing today's stunning 3D effects.

We also haven't tried to create an OpenGL "reference." Our coverage of OpenGL represents only a tiny fraction of its capabilities. Rather, our aim is to use OpenGL as a vehicle for teaching the fundamentals of modern shader-based 3D graphics programming and provide the reader with a sufficiently deep understanding for further study.

What's New in this Edition

We have added three new chapters in this 2nd edition of *Computer Graphics Programming in OpenGL using C++*:

- Chapter 15 – *Simulating Water*
- Chapter 16 – *Ray Tracing*
- Chapter 17 – *Stereoscopy*

Ray tracing in particular has become "hot" recently, so we are especially excited that it is now included in our book. It is also a huge topic, so even though our coverage is just a basic introduction, Chapter 16 is now the longest chapter in the book. Chapter 16 also includes an introduction to *compute shaders*, which were introduced in OpenGL 4.3, and an introduction to additive and subtractive color blending, which expands on a topic that was introduced in Section 14.2.

For years, our own students have repeatedly expressed an interest in simulating water. However, water takes so many forms that writing an introductory section on the topic is challenging. Ultimately, we decided to present water in a way that would complement related topics in the book such as terrain, sky, etc., and so in Chapter 15 we focus on utilizing our noise maps from Chapter 14 to generate water surfaces such as are seen in lakes and oceans.

The new chapter on stereoscopy is motivated by the increased popularity of virtual reality. However, it is also applicable to the development of animation for "3D movies", and we have tried to provide introductory coverage of both uses equally.

As a result of these additions, this 2nd edition is larger than the previous edition.

Besides the new material, there are important revisions throughout the book. For example, we fixed bugs in our Torus class in Chapter 6 and made significant improvements to our noise map functions in Chapter 14. We expanded our Utils.cpp utility class to handle the loading of compute shaders. We also helped identify a bug in SOIL2 (now fixed) that affected Macintosh users attempting to load cubemaps.

There are dozens of small changes in every chapter that the reader might not even notice: fixing typos, cleaning up code inconsistencies, updating the installation instructions, making slight wording changes, sprucing up figures, updating references, etc. Completely eliminating typos is virtually impossible in a book that covers an ever-changing technology-rich topic, but we really have tried hard.

Intended Audience

This book is targeted at students of computer science. This could mean undergraduates pursuing a BS degree, but it could also mean anyone who studies computer science. As such, we are assuming that the reader has at least a solid background in object-oriented programming, at the level of someone who is, say, a computer science major at the junior or senior level.

There are also some specific things that we use in this book that we don't cover, because we assume the reader already has sufficient background. In particular, these are:

- C++ and its most commonly used libraries, such as the Standard Template Library;
- familiarity with using an *Integrated Development Environment* (IDE), such as Visual Studio;
- basic data structures and algorithms, such as linked lists, stacks, and queues, etc.
- recursion;
- event-driven programming concepts;
- basic matrix algebra and trigonometry; and
- awareness of color models, such as RGB, RGBA, etc.

It is hoped that the potential audience for this new book is further bolstered by the existence of its "sister" textbook, *Computer Graphics Programming in OpenGL with Java*. In particular, we envision a learning environment where students are free to utilize either C++ or Java *in the same classroom*, selecting one or the other book. The two texts cover the material sufficiently in lockstep that we have been able to conduct a graphics programming course successfully in this manner.

How to Use This Book

This book is designed to be read from front to back. That is, material in later chapters frequently relies on information learned in earlier chapters. So, it probably won't work to jump back and forth in the chapters; rather, work your way forward through the material.

This is also intended mostly as a practical, hands-on guide. While there is plenty of theoretical material included, the reader should treat this text as a sort of "workbook," in which you learn basic concepts by actually programming them yourself. We have provided code for all of the examples, but to really learn the concepts you will want to "play" with those examples—extend them to build your own 3D scenes.

At the end of each chapter are a few exercises to solve. Some are very simple, involving merely making simple modifications to the provided code. The problems that are marked "*(PROJECT)*," however, are expected to take some time to solve, and require writing a significant amount of code, or combining techniques from various examples. There are also a few marked "*(RESEARCH)*"—those are problems that encourage independent study because this textbook doesn't provide sufficient detail to solve them.

OpenGL calls often involve long lists of parameters. While writing this book, the authors debated whether or not to, in each case, describe all of the parameters. We decided that in the early chapters we would describe every detail. Further into the book, as the topics progress, we decided to avoid getting bogged down in every piece of minutiae in the OpenGL calls (and there are *many*), for fear of the reader losing sight of the big picture. For this reason, it is essential when working through the examples to have ready access to reference material for OpenGL and the various libraries being used.

For this, there are a number of excellent online resources that we recommend using in conjunction with this book. The documentation for OpenGL is absolutely essential; details on the various commands are available either by simply using Google to search for the command in question, or by visiting:

https://www.khronos.org/registry/OpenGL-Refpages/gl4/

Our examples utilize a mathematics library called GLM. After installing GLM (described in the appendices), the reader should locate the accompanying online documentation and bookmark it. At press time, the current link is:

https://glm.g-truc.net/0.9.9/index.html

Another library used throughout the book for which the reader may wish to periodically consult its documentation is SOIL2, which is used for loading and processing texture image files. SOIL2, and the image loading library stb on which it is based, doesn't have a central documentation resource, but several examples are available via Google, and on their respective repositories:

https://github.com/SpartanJ/soil2
https://github.com/nothings/stb

There are many other books on 3D graphics programming that we recommend reading in parallel with this book (such as for solving the "research" problems). Here are five that we often refer to:

- (Sellers et al.) *OpenGL SuperBible* [SW15]
- (Kessenich et al.) *OpenGL Programming Guide* [KS16] (the "red book")
- (Wolff) *OpenGL 4 Shading Language Cookbook* [WO18]
- (Angel and Shreiner) *Interactive Computer Graphics* [AS14]
- (Luna) *Introduction to 3D Game Programming with DirectX 12* [LU16]

Companion Files

This book is accompanied by a companion disc that contains the following items:

- All of the C++/OpenGL programs and related utility class files and GLSL shader code presented in the book
- The models and texture files used in the various programs and examples
- The cubemap and skydome image files used to make the skies and horizons
- Normal maps and height maps for lighting and surface detail effects
- All of the figures in the book, as image files

Readers who have purchased the electronic version of this book may obtain these files by contacting the publisher at *info@merclearning.com*.

Instructor Ancillaries

Instructors in a college or university setting are encouraged to obtain the *instructor ancillary package* that is available for this book, which contains the following additional items:

- A complete set of PowerPoint slides covering all topics in the book
- Solutions to most of the exercises at the ends of the chapters, including code where applicable
- Sample syllabus for a course based on the book
- Additional hints for presenting the material, chapter-by-chapter

This instructor ancillary package is available by contacting the publisher at *info@merclearning.com*.

Acknowledgments

A lot of the content in this book is built from our first book *Computer Graphics Programming in OpenGL with Java*, which we drafted in 2016 for the CSc-155 (Advanced Computer Graphics Programming) course at CSU Sacramento. Many CSc-155 students actively contributed suggestions and bug fixes to an early draft during that year, including Mitchell Brannan, Tiffany Chiapuzio-Wong, Samson Chua, Anthony Doan, Kian Faroughi, Cody Jackson, John Johnston, Zeeshan Khaliq, Raymond Rivera, Oscar Solorzano, Darren Takemoto, Jon Tinney, James Womack, and Victor Zepeda. The following year our colleague Dr. Pinar Muyan-Ozcelik used the first edition of the Java book while teaching CSc-155 for her first time, and kept a running log of questions and corrections for each chapter, which led to many improvements both for the second edition of the Java book, and the first edition of the C++ book.

In Spring 2020 we tested our idea of allowing students (in our CSc-155 course) to select either C++ or Java, using the respective edition of this textbook. It was a sort of acid test of our "sister" textbook idea, and we were pleased with how things went. Again, students caught typos – Paul McHugh in particular caught and fixed an important memory leak in our 3D texture code.

Much of the code in Chapters 15 and 16 was developed by two of our best students, Chris Swenson and Luis Gutierrez, respectively. Both did an excellent job of distilling these complex topics into nicely coherent solutions, and their contributions helped to make these two new chapters possible.

We continue to receive a steady stream of great feedback from instructors around the world who adopt our books for their courses, and from professionals and enthusiasts – Dr. Mauricio Papa (University of Tulsa), Dan Asimov (NASA Ames), Sean McCrory, Michael Hiatt, Scott Anderson, Reydalto Hernandez, and Bill Crupi, just to name a few.

Dr. Alan Mills, over the course of several months starting in early 2020, sent us over two hundred suggestions and corrections from his notes as he worked through our Java edition. About half of his notes were also applicable to the C++ edition. Among his many finds was a significant correction to the texture coordinates in the torus model. Alan's attention to detail is amazing and we greatly appreciate the positive impact of his efforts on both books.

Jay Turberville of Studio 522 Productions in Scottsdale (Arizona) built the dolphin model shown on the cover and used throughout all of our books. Our students love it. Studio 522 Productions does incredibly high-quality 3D animation and video production, as well as custom 3D modeling. We are thrilled that Mr. Turberville kindly offered to build such a wonderful model just for these books.

Martín Lucas Golini, who developed and maintains the SOIL2 texture image handling library, has been very supportive and enthusiastic about our book, and has been quickly responsive whenever any problems have arisen. His availability to us has been much appreciated.

We wish to thank a few other artists and researchers who were gracious enough to allow us to utilize their models and textures. James Hastings-Trew of Planet Pixel Emporium provided many of the planetary surface textures. Paul Bourke allowed us to use his wonderful star field. Dr. Marc Levoy of Stanford University granted us permission to use the famous "Stanford Dragon" model. Paul Baker's bump-mapping tutorial formed the basis of the "torus" model we used in many examples. We also thank Mercury Learning for allowing us to use some of the textures from [LU16].

The late Dr. Danny Kopec connected us with Mercury Learning and introduced us to its publisher, David Pallai. Being a chess enthusiast, Dr. Gordon (one of the authors) was originally familiar with Dr. Kopec as a well-known international chess master and prolific chess book author. He was also a computer science professor, and his textbook, *Artificial Intelligence in the 21st Century*, inspired us to consider Mercury Learning for our book project. We had several telephone conversations with Dr. Kopec which were extremely informative. We were deeply saddened by Dr. Kopec's untimely passing in 2016, and regret that he didn't have the chance to see our books, which he had helped jump start, come to fruition.

Finally, we wish to thank David Pallai and Jennifer Blaney of Mercury Learning for their continued enthusiasm and support for this project and for guiding us through the textbook publishing process.

Errata

If you find any errors in our book, please let us know! Despite our best efforts, this book certainly contains mistakes. We will do our best to post corrections as soon as errors are reported to us. We have established a webpage for collecting errata and posting corrections:

http://ecs.csus.edu/~gordonvs/textC2E.html

The publisher, Mercury Learning, also maintains a link to our errata page. So, if the URL for our errata page should ever change, check the Mercury Learning website for the latest link.

About the Authors

Dr. V. Scott Gordon has been a professor in the California State University system for over twenty-five years, and currently teaches advanced graphics and game engineering courses at CSU Sacramento. He has authored or coauthored over thirty publications in a variety of areas including artificial intelligence, neural networks, evolutionary computation, computer graphics, software engineering, video and strategy game programming, and computer science education. Dr. Gordon obtained his PhD at Colorado State University. He is also a jazz drummer and a competitive table tennis player.

Dr. John Clevenger has over forty years of experience teaching a wide variety of courses including advanced graphics, game architecture, operating systems, VLSI chip design, system simulation, and other topics. He is the developer of several software frameworks and tools for teaching graphics and game architecture, including the graphicslib3D library used in the first edition of our Java-based textbook. He is the technical director of the International Collegiate Programming Contest (ICPC), and oversees the ongoing development of PC^2, the most widely used programming contest support system in the world. Dr. Clevenger obtained his PhD at the University of California, Davis. He is also a performing jazz musician and spends summer vacations in his mountain cabin.

References

[AS14] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with WebGL*, 7th ed. (Pearson, 2014).

[KS16] J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9th ed. (Addison-Wesley, 2016).

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

[WO18] D. Wolff, *OpenGL 4 Shading Language Cookbook*, 3rd ed. (Packt Publishing, 2018).

CHAPTER 1

GETTING STARTED

Graphics programming has a reputation for being among the most challenging computer science topics to learn. These days, graphics programming is *shader based*—that is, some of the program is written in a standard language such as C++ or Java for running on the CPU and some is written in a special-purpose *shader* language for running directly on the graphics card (GPU). Shader programming has a steep learning curve, so that even drawing something simple requires a convoluted set of steps to pass graphics data down a "pipeline." Modern graphics cards are able to process this data in *parallel*, and so the graphics programmer must understand the parallel architecture of the GPU, even when drawing simple shapes.

The payoff, however, is extraordinary power. The blossoming of stunning virtual reality in videogames and increasingly realistic effects in Hollywood movies can be greatly attributed to advances in shader programming. If reading this book is your entrée into 3D graphics, you are taking on a personal challenge that will reward you not only with pretty pictures but with a level of control over your machine that you never imagined was possible. Welcome to the exciting world of computer graphics programming!

1.1 LANGUAGES AND LIBRARIES

Modern graphics programming is done using a *graphics library.* That is, the programmer writes code which invokes functions in a predefined library (or set of libraries) that provide support for lower-level graphical operations. There are many graphics libraries in use today, but the most common library for platform-independent graphics programming is called *OpenGL (Open Graphics Library).* This book describes how to use OpenGL for 3D graphics programming in C++.

Using OpenGL with C++ requires configuring several libraries. At this time there is a dizzying array of options, depending on one's individual needs. In this section, we describe which libraries are needed, some common options for each, and the option(s) that we will use throughout the book. Details on how to install and configure these libraries for use on your specific platform can be found in the Appendices.

In summary, you will need languages and libraries for the following functions:

- C++ development environment
- OpenGL / GLSL
- window management
- extension library
- math library
- texture management

It is likely that the reader will need to do several preparatory steps to ensure that each of these are installed and properly accessible on his/her system. In the following subsections we briefly describe each of them; see the Appendices for details on how to install and/or configure them for use.

1.1.1 C++

C++ is a general-purpose programming language that first appeared in the mid-1980s. Its design, and the fact that it is generally compiled to native machine code, make it an excellent choice for systems that require high performance, such as 3D graphics computing. Another advantage of C++ is that the OpenGL call library is C based.

Many C++ development environments are available. In this textbook we recommend using *Microsoft Visual Studio* [VS20] if using a PC, and *Xcode* [XC18] if using a Macintosh. Descriptions for installing and configuring each of them, depending on your platform, are given in the Appendices.

1.1.2 OpenGL / GLSL

Version 1.0 of OpenGL appeared in 1992 as an "open" alternative to vendor-specific Application Programming Interfaces (APIs) for computer graphics.

Its specification and development was managed and controlled by the *OpenGL Architecture Review Board (ARB)*, a then newly formed group of industry participants. In 2006 the ARB transferred control of the OpenGL specification to the *Khronos Group*, a nonprofit consortium which manages not only the OpenGL specification but a wide variety of other open industry standards.

Since its beginning OpenGL has been revised and extended regularly. In 2004, version 2.0 introduced the OpenGL Shading Language (GLSL), allowing "shader programs" to be installed and run directly in graphics pipeline stages.

In 2009, version 3.1 removed a large number of features that had been deprecated, to enforce the use of shader programming as opposed to earlier approaches (referred to as "immediate mode").[1] Among the more recent features, version 4.0 (in 2010) added a *tessellation* stage to the programmable pipeline.

This textbook assumes that the user is using a machine with a graphics card that supports at least version 4.3 of OpenGL. If you are not sure which version of OpenGL your GPU supports, there are free applications available on the web that can be used to find out. One such application is GLView, by a company named "realtechvr" [GV20].

1.1.3 Window Management

OpenGL doesn't actually draw to a computer screen. Rather, it renders to a *frame buffer*, and it is the job of the individual machine to then draw the contents of the frame buffer onto a window on the screen. There are various libraries that support doing this. One option is to use the windowing capabilities provided by the operating system, such as the ***Microsoft Windows API***. This is generally impractical and requires a lot of low-level coding. **GLUT** is a historically popular option; however, it is deprecated. A modernized extension is **freeglut**. Other related options are **CPW**, **GLOW**, and **GLUI**.

One of the most popular options, and the one used in this book, is **GLFW**, which has built-in support for Windows, Macintosh, Linux, and other systems [GF20]. It can be downloaded from www.glfw.org, and it must be compiled on the machine where it is to be used (we describe those steps in the Appendices).

---

[1] Despite this, many graphics card manufacturers (notably NVIDIA) continue to support deprecated functionality.

1.1.4 Extension Library

OpenGL is organized around a set of base functions and an *extension* mechanism used to support new functionality as technologies advance. Modern versions of OpenGL, such as those found in version 4+ as we use in this book, require identifying the extensions available on the GPU. There are commands built into core OpenGL for doing this, but they involve several rather convoluted lines of code that would need to be performed for each modern command used—and in this book we use such commands constantly. Therefore, it has become standard practice to use an *extension library* to take care of these details, and to make modern OpenGL commands available to the programmer directly. Examples are **Glee**, **GLLoader**, **GLEW**, and more recently **GL3W** and **GLAD**.

A commonly used library among those listed is **GLEW**, which stands for *OpenGL Extension Wrangler*. It is available for a variety of operating systems including Windows, Macintosh, and Linux [GE20]. **GLEW** is not a perfect choice; for example, it requires an additional DLL. Recently, many developers are choosing **GL3W** or **GLAD**. They have the advantage of being automatically updated, but they also require that Python be installed. For these reasons, in this book we have opted to use **GLEW**. It can be downloaded at glew.sourceforge.net. Complete instructions for installing and configuring **GLEW** are given in the appendices.

1.1.5 Math Library

3D graphics programming makes heavy use of vector and matrix algebra. For this reason, use of OpenGL is greatly facilitated by accompanying it with a function library or class package to support common mathematical tasks. Two such libraries that are frequently used with OpenGL are **Eigen** and **vmath**, the latter being used in the popular *OpenGL SuperBible* [SW15].

Arguably the most popular, and the one used in this book, is *OpenGL Mathematics*, usually called **GLM**. It is a header-only C++ library compatible with Windows, Macintosh, and Linux [GM20]. **GLM** commands conveniently use the same naming conventions as those in GLSL, making it easy to go back and forth when reading C++ and GLSL code used in a particular application. **GLM** is available for download at glm.g-truc.net.

GLM provides classes and basic math functions related to graphics concepts, such as *vector*, *matrix*, and *quaternion*. It also contains a variety of utility classes for creating and using common 3D graphics structures, such as perspective and

look-at matrices. It was first released in 2005, and it is maintained by Christophe Riccio [GM20]. Instructions for installing **GLM** are given in the appendices.

1.1.6 Texture Management

Starting with Chapter 5, we will use image files to add "texture" to the objects in our graphics scenes. This means that we will frequently need to load such image files into our C++/OpenGL code. It is possible to code a texture image loader from scratch; however, given the wide variety of image file formats, it is generally preferable to use a texture loading library. Some examples are **FreeImage**, **DevIL**, OpenGL Image (**GLI**), and **Glraw**. Probably the most commonly used OpenGL image loading library is Simple OpenGL Image Loader (**SOIL**), although it has become somewhat outdated.

The texture image loading library used in this book is **SOIL2**, an updated fork of **SOIL**. Like the previous libraries we have chosen, **SOIL2** is compatible with a wide variety of platforms [SO20], and detailed installation and configuration instructions are given in the appendices.

1.1.7 Optional Libraries

There are many other helpful libraries that the reader may wish to utilize. For example, in this book we show how to implement a simple "OBJ" model loader from scratch. However, as we will see, it doesn't handle many of the options available in the OBJ standard. Some examples of more sophisticated OBJ importers are **Assimp** and **tinyobjloader**. For our examples, we will just use our simple model loader described and implemented in this book.

1.2 INSTALLATION AND CONFIGURATION

While developing the C++ edition of this book, we wrestled with the best approach for including the platform-specific configuration information necessary to run the example programs. Configuring a system for using OpenGL with C++ is considerably more complicated than the equivalent configuration using Java, which can be described in just a few short paragraphs (as can be seen in the Java edition of the book [GC18]). Ultimately, we opted to separate installation and configuration information into individual platform-specific Appendices. We hope

that this will provide each reader with a single relevant place to look for information regarding his/her specific system, while at the same time avoiding bogging down the rest of the text with platform-specific details which may not be relevant to every reader. In this edition, we provide detailed configuration instructions for Microsoft Windows in Appendix A, and for the Apple Macintosh in Appendix B.

Continually updated library installation instructions will be maintained on this textbook's website, available at: *http://athena.ecs.csus.edu/~gordonvs/textC2E.html*

References

[GC18] V. Gordon and J. Clevenger, *Computer Graphics Programming in OpenGL with Java*, 2nd ed. (Mercury Learning, 2018).

[GE20] OpenGL Extension Wrangler (GLEW), accessed July 2020, http://glew.sourceforge.net/

[GF20] Graphics Library Framework (GLFW), accessed July 2020, http://www.glfw.org/

[GM20] OpenGL Mathematics (GLM), accessed July 2020, http://glm.g-truc.net/0.9.8/index.html

[GV20] GLView, realtech-vr, accessed July 2020, https://www.realtech-vr.com/home/glview

[SO20] Simple OpenGL Image Library 2 (SOIL2), *SpartanJ*, accessed July 2020, https://github.com/SpartanJ/SOIL2

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

[VS20] Microsoft Visual Studio downloads, accessed July 2020, https://www.visualstudio.com/downloads

[XC18] Apple Developer site for Xcode, accessed January 2018, https://developer.apple.com/xcode

CHAPTER 2

THE OPENGL GRAPHICS PIPELINE

OpenGL (Open Graphics Library) is a multi-platform 2D and 3D graphics API that incorporates both hardware and software. Using OpenGL requires a graphics card (GPU) that supports a sufficiently up-to-date version of OpenGL (as described in Chapter 1).

On the hardware side, OpenGL provides a multi-stage *graphics pipeline* that is partially programmable using a language called **GLSL** (OpenGL Shading Language).

On the software side, OpenGL's API is written in C, and thus the calls are directly compatible with C and C++. Stable language *bindings* (or "wrappers") are available for more than a dozen other popular languages (Java, Perl, Python, Visual Basic, Delphi, Haskell, Lisp, Ruby, etc.) with virtually equivalent performance. This textbook uses C++, probably the most popular language choice. When using C++, the programmer writes code that runs on the CPU (compiled, of course) and includes OpenGL calls. We will refer to a C++ program that contains OpenGL calls as a *C++/OpenGL application*. One important task of a C++/OpenGL application is to install the programmer's GLSL code onto the GPU.

An overview of a C++-based graphics application is shown in Figure 2.1, with the software components highlighted in pink.

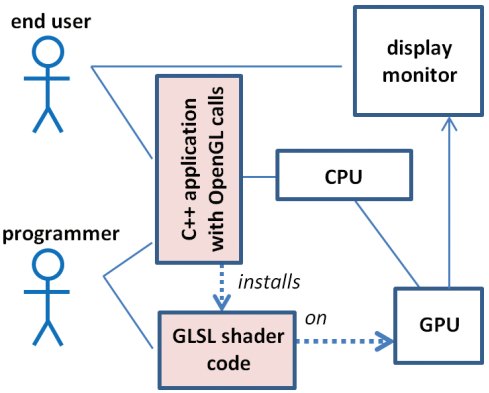

Figure 2.1
Overview of a C++-based graphics application.

Some of the code we will write will be in C++, with OpenGL calls, and some will be written in GLSL. Our C++/OpenGL application will work together with our GLSL modules, and the hardware, to create our 3D graphics output. Once our application is complete, the end user will interact with the C++ application.

GLSL is an example of a *shader language*. Shader languages are intended to run on a GPU, in the context of a graphics pipeline. There are other shader languages, such as HLSL, which works with Microsoft's 3D framework *DirectX*. GLSL is the specific shader language that is compatible with OpenGL, and thus we will write shader code in GLSL, in addition to our C++/OpenGL application code.

For the rest of this chapter, we will take a brief "tour" of the OpenGL pipeline. The reader is not expected to understand every detail thoroughly but should just get a feel for how the stages work together.

2.1 THE OPENGL PIPELINE

Modern 3D graphics programming utilizes a *pipeline*, in which the process of converting a 3D scene to a 2D image is broken down into a series of steps. OpenGL and DirectX both utilize similar pipelines.

A simplified overview of the OpenGL graphics pipeline is shown in Figure 2.2 (not every stage is shown, just the major ones we will study). The C++/OpenGL application sends graphics data into the vertex shader—processing proceeds through the pipeline, and pixels emerge for display on the monitor.

The stages shaded in blue (vertex, tessellation, geometry, and fragment) are *programmable* in GLSL. It is one of the responsibilities of the C++/OpenGL application to load GLSL programs into these shader stages, as follows:

1. It uses C++ to obtain the GLSL shader code, either from text files or hard-coded as strings.

2. It then creates OpenGL shader objects and loads the GLSL shader code into them.
3. Finally, it uses OpenGL commands to compile and link objects and install them on the GPU.

In practice, it is usually necessary to provide GLSL code for at least the *vertex* and *fragment* stages, whereas the tessellation and geometry stages are optional. Let's walk through the entire process and see what takes place at each step.

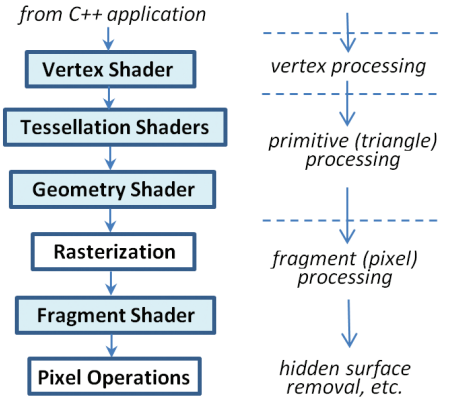

Figure 2.2
Overview of the OpenGL pipeline.

2.1.1 C++/OpenGL Application

The bulk of our graphics application is written in C++. Depending on the purpose of the program, it may interact with the end user using standard C++ libraries. For tasks related to 3D rendering, it uses OpenGL calls. As described in the previous chapter, we will be using several additional libraries: GLEW (OpenGL Extension Wrangler), GLM (OpenGL Mathematics), SOIL2 (Simple OpenGL Image Loader), and GLFW (Graphics Library Framework).

The GLFW library includes a class called GLFWwindow on which we can draw 3D scenes. As already mentioned, OpenGL also gives us commands for installing GLSL programs onto the programmable shader stages and compiling them. Finally, OpenGL uses *buffers* for sending 3D models and other related graphics data down the pipeline.

Before we try writing shaders, let's write a simple C++/OpenGL application that instantiates a GLFWwindow and sets its background color. Doing that won't require any shaders at all! The code is shown in Program 2.1. The main() function shown in Program 2.1 is the same one that we will use throughout this textbook. Among the significant operations in main() are: (a) initializes the GLFW library, (b) instantiates a GLFWwindow, (c) initializes the GLEW library, (d) calls the function "init()" once, and (e) calls the function "display()" repeatedly.

The "init()" function is where we will place application-specific initialization tasks. The display() method is where we place code that draws to the GLFWwindow.

In this example, the glClearColor() command specifies the color value to be applied when clearing the background—in this case (1,0,0,1), corresponding to the RGB values of the color red (plus a "1" for the opacity component). We then use the OpenGL call glClear(GL_COLOR_BUFFER_BIT) to actually fill the color buffer with that color.

Program 2.1 First C++/OpenGL Application

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

using namespace std;

void init(GLFWwindow* window) { }

void display(GLFWwindow* window, double currentTime) {
    glClearColor(1.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}

int main(void) {
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 - program1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {
        display(window, glfwGetTime());
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

The output of Program 2.1 is shown in Figure 2.3.

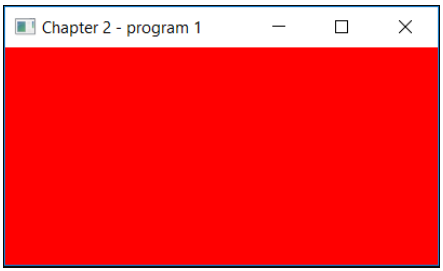

Figure 2.3
Output of Program 2.1.

The mechanism by which these functions are deployed is as follows: the GLFW and GLEW libraries are initialized using the commands glfwInit() and glewInit() respectively. The GLFW window and an associated OpenGL *context*[1] are created with the glfwCreateWindow() command, with options set by any preceding *window hints*. Our window hints specify that the machine must be compatible with OpenGL version 4.3 ("major"=4. and "minor"=3). The parameters on the glfwCreateWindow() command specify the width and height of the window (in pixels) and the title placed at the top of the window. (The additional two parameters which are set to NULL, and which we aren't using, allow for full screen mode and resource sharing.) Vertical synchronization (VSync) is enabled by using the glfwSwapInterval() and glfwSwapBuffers() commands—GLFW windows are by default double-buffered.[2] Note that creating the GLFW window doesn't automatically make the associated OpenGL context current—for that reason we also call glfwMakeContextCurrent().

Our main() includes a very simple rendering loop that calls our display() function repeatedly. It also calls glfwSwapBuffers(), which paints the screen, and glfwPollEvents(), which handles other window-related events (such as a key being pressed). The loop terminates when GLFW detects an event that should close the window (such as the user clicking the "X" in the upper right corner). Note that we have included a reference to the GLFW window object on the init() and display() calls; those functions may in certain circumstances need access to it. We have also included the current time on the call to display(), which will be useful for ensuring that our animations run at the same speed regardless of the computer being used. For this purpose, we use glfwGetTime(), which returns the elapsed time since GLFW was initialized.

---

1 The term "context" refers to an OpenGL instance and its state information, which includes items such as the color buffer.

2 "Double buffering" means that there are two color buffers—one that is displayed, and one that is being rendered to. After an entire frame is rendered, the buffers are swapped. Double buffering is used to reduce undesirable visual artifacts.

Now is an appropriate time to take a closer look at the OpenGL calls in Program 2.1. Consider this one:

```
glClear(GL_COLOR_BUFFER_BIT);
```

In this case, the OpenGL function being called, as described in the OpenGL reference documentation (available on the web at https://www.opengl.org/sdk/docs), is:

```
void glClear(GLbitfield mask);
```

The parameter references a "GLbitfield" called "GL_COLOR_BUFFER_BIT". OpenGL has many predefined constants (some of them are called *enums*); this one references the *color buffer* that contains the pixels as they are rendered. OpenGL has several color buffers, and this command clears all of them—that is, it fills them with a predefined color called the "clear color." Note that "clear" in this context doesn't mean "a color that is clear"; rather, it refers to the color that is applied when a color buffer is reset (cleared).

Immediately before the call to glClear() is a call to glClearColor(). This allows us to specify the value placed in the elements of a color buffer when it is cleared. Here we have specified (1,0,0,1), which corresponds to the RGBA color *red*.

Finally, our render loop exits when the user attempts to close the GLFW window. At that time, our main() asks GLFW to destroy the window and terminate, via calls to glfwDestroyWindow() and glfwTerminate() respectively.

2.1.2 Vertex and Fragment Shaders

Our first OpenGL program didn't actually draw anything—it simply filled the color buffer with a single color. To actually draw something, we need to include a *vertex shader* and a *fragment shader*.

You may be surprised to learn that OpenGL is capable of drawing only a few kinds of very simple things, such as *points*, *lines*, or *triangles*. These simple things are called *primitives*, and for this reason, most 3D models are made up of lots and lots of primitives, usually triangles.

Primitives are made up of *vertices*—for example, a triangle consists of three vertices. The vertices can come from a variety of sources—they can be read from files and then loaded into buffers by the C++/OpenGL application, or they can be hardcoded in the C++ code or even in the GLSL code.

Before any of this can happen, the C++/OpenGL application must compile and link appropriate GLSL vertex and fragment shader programs, and then load them into the pipeline. We will see the commands for doing this shortly.

The C++/OpenGL application also is responsible for telling OpenGL to construct triangles. We do this by using the following OpenGL function:

```
glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

The mode is the type of primitive—for triangles we use GL_TRIANGLES. The parameter "first" indicates which vertex to start with (generally vertex number 0, the first one), and count specifies the total number of vertices to be drawn.

When glDrawArrays() is called, the GLSL code in the pipeline starts executing. Let's now add some GLSL code to that pipeline.

Regardless of where they originate, all of the vertices pass through the vertex shader. They do so *one by one*; that is, the shader is executed *once per vertex*. For a large and complex model with a lot of vertices, the vertex shader may execute hundreds, thousands, or even millions of times, often in parallel.

Let's write a simple program with only one vertex, hardcoded in the vertex shader. That's not enough to draw a triangle, but it is enough to draw a point. For it to display, we also need to provide a *fragment shader*. For simplicity we will declare the two shader programs as arrays of strings.

Program 2.2 Shaders, Drawing a POINT

(…..#includes are the same as before)

```
#define numVAOs 1
GLuint renderingProgram;
GLuint vao[numVAOs];
```
} *new declarations*

```
GLuint createShaderProgram() {
    const char *vshaderSource =
        "#version 430    \n"
        "void main(void) \n"
        "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";

    const char *fshaderSource =
        "#version 430    \n"
        "out vec4 color; \n"
```

```
        "void main(void) \n"
        "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";

    GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
    GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);

    glShaderSource(vShader, 1, &vshaderSource, NULL);
    glShaderSource(fShader, 1, &fshaderSource, NULL);
    glCompileShader(vShader);
    glCompileShader(fShader);

    GLuint vfProgram = glCreateProgram();
    glAttachShader(vfProgram, vShader);
    glAttachShader(vfProgram, fShader);
    glLinkProgram(vfProgram);

    return vfProgram;
}

void init(GLFWwindow* window) {
    renderingProgram = createShaderProgram();
    glGenVertexArrays(numVAOs, vao);
    glBindVertexArray(vao[0]);
}

void display(GLFWwindow* window, double currentTime) {
    glUseProgram(renderingProgram);
    glDrawArrays(GL_POINTS, 0, 1);
}

. . . main() same as before
```

The program appears to have output a blank window (see Figure 2.4). But close examination reveals a tiny blue dot in the center of the window (assuming that this printed page is of sufficient resolution). The default size of a point in OpenGL is one pixel.

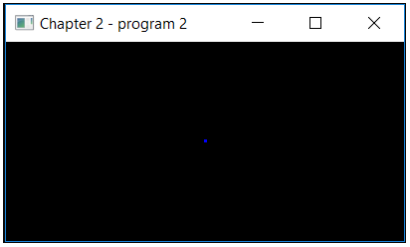

Figure 2.4
Output of Program 2.2.

There are many important details in Program 2.2 (color-coded in the program, for convenience) for us to discuss. First, note the frequent use of "GLuint"—this is a platform-independent shorthand for "unsigned int", provided by OpenGL (many OpenGL constructs have integer references). Next, note that init() is no longer empty—it now calls another function

named "createShaderProgram()" (that we wrote). This function starts by declaring two shaders as character strings called vshaderSource and fshaderSource. It then calls glCreateShader() twice, which generates the two shaders of types GL_VERTEX_SHADER and GL_FRAGMENT_SHADER. OpenGL creates each shader object (initially empty), and returns an integer ID for each that is an index for referencing it later—our code stores this ID in the variables vShader and fShader. It then calls glShaderSource(), which loads the GLSL code from the strings into the empty shader objects. The shaders are then each compiled using glCompileShader(). glShaderSource() has four parameters: (a) the shader object in which to store the shader, (b) the number of strings in the shader source code, (c) an array of pointers to strings containing the source code, and (d) an additional parameter we aren't using (it will be explained later, in the supplementary chapter notes). Note that the two calls specify the number of lines of code in each shader as being "1"—this too is explained in the supplementary notes.

The application then creates a *program* object named vfProgram, and saves the integer ID that points to it. An OpenGL "program" object contains a series of compiled shaders, and here we see the following commands: glCreateProgram() to create the program object, glAttachShader() to attach each of the shaders to it, and then glLinkProgram() to request that the GLSL compiler ensure that they are compatible.

As we saw earlier, after init() finishes, display() is called. One of the first things display() does is call glUseProgram(), which loads the program containing the two compiled shaders into the OpenGL pipeline stages (onto the GPU!). Note that glUseProgram() *doesn't run the shaders*, it just loads them onto the hardware.

As we will see later in Chapter 4, ordinarily at this point the C++/OpenGL program would prepare the vertices of the model being drawn for sending down the pipeline. But not in this case, because for our first shader program we simply hardcoded a single vertex in the vertex shader. Therefore in this example the display() function next proceeds to the glDrawArrays() call, which initiates pipeline processing. The primitive type is GL_POINTS, and there is just one point to display.

Now let's look at the shaders themselves, shown in green earlier (and duplicated in the explanations that follow). As we saw, they have been declared in the C++/OpenGL program as arrays of strings. This is a clumsy way to code, but it is sufficient in this very simple case. The vertex shader is:

```
#version 430
void main(void)
{   gl_Position = vec4(0.0,  0.0,  0.0,  1.0);  }
```

The first line indicates the OpenGL version, in this case 4.3. There follows a "main" function (as we will see, GLSL is somewhat C++-like in syntax). The primary purpose of any vertex shader is to send a vertex down the pipeline (which, as mentioned before, it does for every vertex). The built-in variable gl_Position is used to set a vertex's coordinate position in 3D space, and is sent to the next stage in the pipeline. The GLSL datatype vec4 is used to hold a 4-tuple, suitable for such coordinates, with the associated four values representing X, Y, Z, and a fourth value set here to 1.0 (we will learn the purpose of this fourth value in Chapter 3). In this case, the vertex is hardcoded to the origin location (0,0,0).

The vertices move through the pipeline to the *rasterizer*, where they are transformed into pixel locations (or more accurately *fragments*—this is described later). Eventually, these pixels (fragments) reach the fragment shader:

```
#version 430
out vec4 color;
void main(void)
{    color = vec4(0.0,  0.0,  1.0,  1.0); }
```

The purpose of any fragment shader is to set the RGB color of a pixel to be displayed. In this case the specified output color (0, 0, 1) is blue (the fourth value 1.0 specifies the level of opacity). Note the "out" tag indicating that the variable color is an output. (It wasn't necessary to specify an "out" tag for gl_Position in the vertex shader, because gl_Position is a predefined output variable.)

There is one detail in the code that we haven't discussed, in the last two lines in the init() function (shown in red). They probably appear a bit cryptic. As we will see in Chapter 4, when sets of data are prepared for sending down the pipeline, they are organized into *buffers*. Those buffers are in turn organized into *Vertex Array Objects* (VAOs). In our example, we hardcoded a single point in the vertex shader, so we didn't need any buffers. However, OpenGL still requires that at least one VAO be created whenever shaders are being used, even if the application isn't using any buffers. So the two lines create the required VAO.

Finally, there is the issue of how the *vertex* that came out of the vertex shader became a *pixel* in the fragment shader. Recall from Figure 2.2 that between vertex processing and pixel processing is the *rasterization* stage. It is there that primitives (such as points or triangles) are converted into sets of pixels. The default size of an OpenGL "point" is one pixel, so that is why our single point was rendered as a single pixel.

Let's add the following command in display(), right before the glDrawArrays() call:

```
glPointSize(30.0f);
```

Now, when the rasterizer receives the vertex from the vertex shader, it will set pixel color values that form a point having a size of 30 pixels. The resulting output is shown in Figure 2.5.

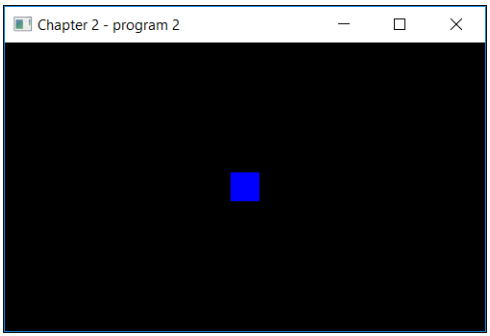

Figure 2.5
Changing glPointSize.

Let's now continue examining the remainder of the OpenGL pipeline.

2.1.3 Tessellation

We cover tessellation in Chapter 12. The programmable tessellation stage is one of the most recent additions to OpenGL (in version 4.0). It provides a *tessellator* that can generate a large number of triangles, typically as a grid, and also some tools to manipulate those triangles in a variety of ways. For example, the programmer might manipulate a tessellated grid of triangles as shown in Figure 2.6.

Tessellation is useful when a lot of vertices are needed on what is otherwise a simple shape, such as on a square area or curved surface. It is also very useful for generating complex terrain, as we will see later. In such instances, it is sometimes

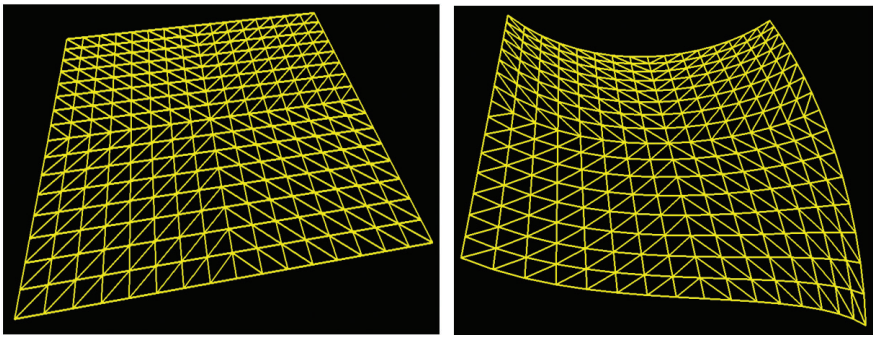

Figure 2.6
Grid produced by tessellator.

much more efficient to have the tessellator in the GPU generate the triangle mesh in hardware, rather than doing it in C++.

2.1.4 Geometry Shader

We cover the geometry shader stage in Chapter 13. Whereas the *vertex shader* gives the programmer the ability to manipulate one *vertex* at a time (i.e., "per-vertex" processing), and the *fragment shader* (as we will see) allows manipulating one *pixel* at a time ("per-fragment" processing), the *geometry shader* provides the capability to manipulate one *primitive* at a time—"per-primitive" processing.

Recalling that the most common primitive is the *triangle*, by the time we have reached the geometry stage, the pipeline must have completed grouping the vertices into triangles (a process called *primitive assembly*). The geometry shader then makes all three vertices in each triangle accessible to the programmer simultaneously.

There are a number of uses for per-primitive processing. The primitives could be altered, such as by stretching or shrinking them. Some of the primitives could be deleted, thus putting "holes" in the object being rendered—this is one way of turning a simple model into a more complex one.

The geometry shader also provides a mechanism for generating additional primitives. Here too, this opens the door to many possibilities for turning simple models into more complex ones.

An interesting use for the geometry shader is for adding surface texture such as bumps or scales—even "hair" or "fur"—to an object. Consider for example the simple torus shown in Figure 2.7 (we will see how to generate this later in the book). The surface of this torus is built out of many hundreds of triangles. If at each triangle we use a geometry shader to add additional triangles that face outward, we get the result shown in Figure 2.8. This "scaly torus" would be computationally expensive to try and model from scratch in the C++/OpenGL application side.

It might seem redundant to provide a per-primitive shader stage when the tessellation stage(s) give the programmer access to *all* of the vertices in an entire model simultaneously. The difference is that tessellation only offers this capability in very limited circumstances—specifically when the model is a grid of triangles generated by the tessellator. It does not provide such simultaneous access to all the vertices of, say, an arbitrary model being sent in from C++ through a buffer.

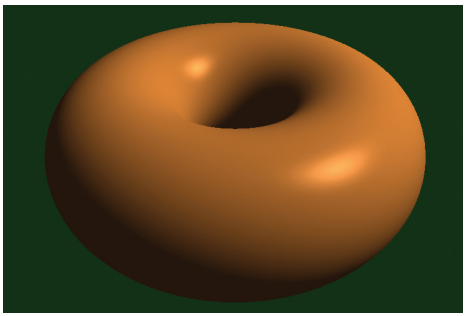

Figure 2.7
Torus model.

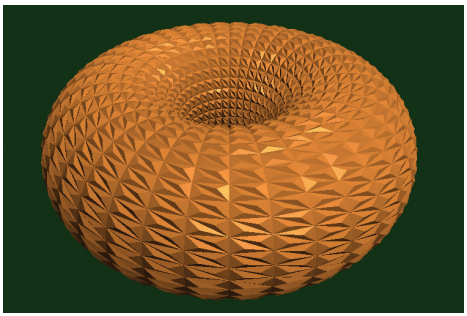

Figure 2.8
Torus modified in geometry shader.

2.1.5 Rasterization

Ultimately, our 3D world of vertices, triangles, colors, and so on needs to be displayed on a 2D monitor. That 2D monitor screen is made up of a *raster*—a rectangular array of pixels.

When a 3D object is *rasterized*, OpenGL converts the primitives in the object (usually triangles) into *fragments*. A fragment holds the information associated with a pixel. Rasterization determines the locations of pixels that need to be drawn in order to produce the triangle specified by its three vertices.

Rasterization starts by interpolating, pairwise, between the three vertices of the triangle. There are some options for doing this interpolation; for now it is sufficient to consider simple linear interpolation as shown in Figure 2.9. The original three vertices are shown in red.

If rasterization were to stop here, the resulting image would appear as wireframe. This is an option in OpenGL, by adding the following command in the display() function, before the call to glDrawArrays():

```
glPolygonMode(GL_FRONT_AND_BACK,
              GL_LINE);
```

If the torus shown previously in Section 2.1.4 is rendered with the addition of this line of code, it appears as shown in Figure 2.10.

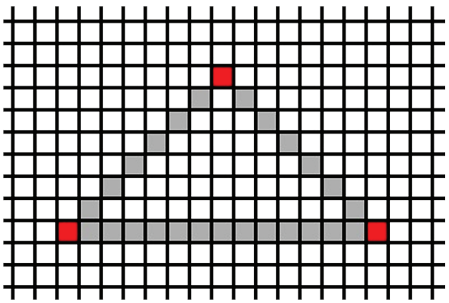

Figure 2.9
Rasterization (step 1).

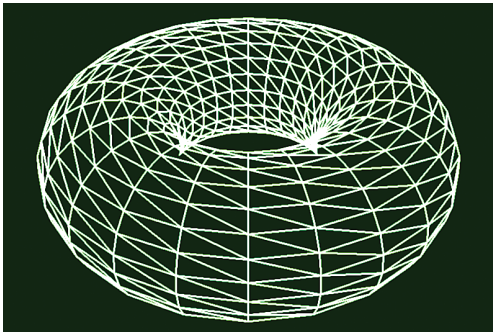

Figure 2.10
Torus with wireframe rendering.

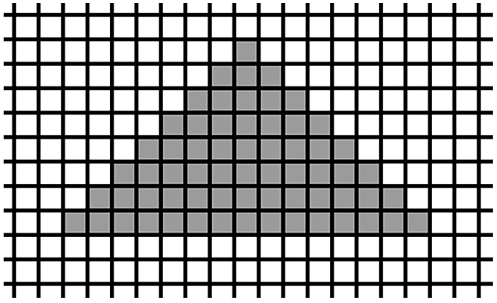

Figure 2.11
Fully rasterized triangle.

If we didn't insert the preceding line of code (or if GL_FILL had been specified instead of GL_LINE), interpolation would continue along raster lines and fill the interior of the triangle, as shown in Figure 2.11. When applied to the torus, this results in the fully rasterized or "solid" torus shown in Figure 2.12 (on the left). Note that in this case the overall shape and curvature of the torus is not evident—that is because we haven't included any texturing or lighting techniques, so it appears "flat." At the right, the same "flat" torus is shown with the wireframe rendering superimposed. The torus shown earlier in Figure 2.7 included lighting effects, and thus revealed the shape of the torus much more clearly. We will study lighting in Chapter 7.

As we will see in later chapters, the rasterizer can interpolate more than just pixels. *Any* variable that is output by the vertex shader and input by the fragment shader will be interpolated based on the corresponding pixel position. We will use this capability to generate smooth color gradations, achieve realistic lighting, and many more effects.

2.1.6 Fragment Shader

As mentioned earlier, the purpose of the fragment shader is to assign colors to the rasterized pixels. We have already seen an example of a fragment shader in Program 2.2. There, the fragment shader simply hardcoded its output to a specific value, so every generated pixel had the same color. However, GLSL affords us virtually limitless creativity to calculate colors in other ways.

One simple example would be to base the output color of a pixel on its location. Recall that in the vertex shader, the outgoing coordinates of a vertex are

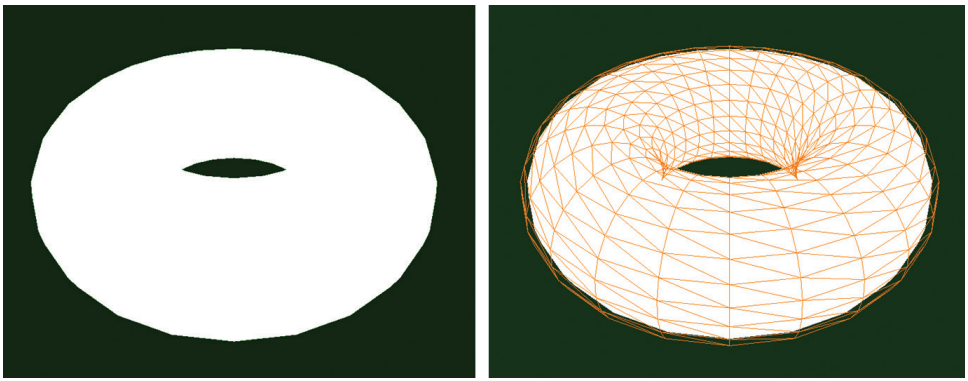

Figure 2.12
Torus with fully rasterized primitives (left), and with wireframe grid superimposed (right).

specified using the predefined variable gl_Position. In the fragment shader, there is a similar variable available to the programmer for accessing the coordinates of an incoming *fragment*, called gl_FragCoord. We can modify the fragment shader from Program 2.2 so that it uses gl_FragCoord (in this case referencing its x component using the GLSL *field selector* notation) to set each pixel's color based on its location, as shown here:

```
#version 430
out vec4 color;
void main(void)
{  if (gl_FragCoord.x < 295) color = vec4(1.0, 0.0, 0.0, 1.0); else color = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Assuming that we increase the GL_PointSize as we did at the end of Section 2.1.2, the pixel colors will now vary across the rendered point—red where the x coordinates are less than 200, and blue otherwise, as seen in Figure 2.13.

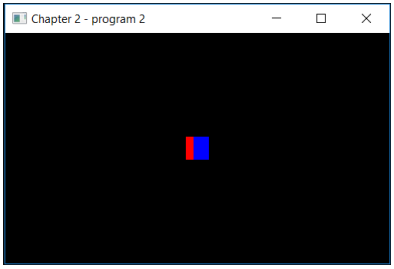

Figure 2.13
Fragment shader color variation.

2.1.7 Pixel Operations

As objects in our scene are drawn in the display() function using the glDrawArrays() command, we usually expect objects in front to block our view of objects behind them. This

also extends to the objects themselves, wherein we expect to see the front of an object, but generally not the back.

To achieve this, we need *hidden surface removal*, or *HSR*. OpenGL can perform a variety of HSR operations, depending on the effect we want in our scene. And even though this phase is not programmable, it is extremely important that we understand how it works. Not only will we need to configure it properly, we will later need to carefully manipulate it when we add shadows to our scene.

Hidden surface removal is accomplished by OpenGL through the cleverly coordinated use of two buffers: the *color buffer* (which we have discussed previously), and the *depth buffer* (sometimes called the *Z-buffer*). Both of these buffers are the same size as the raster—that is, there is an entry in each buffer for every pixel on the screen.

As various objects are drawn in a scene, pixel colors are generated by the fragment shader. The pixel colors are placed in the color buffer—it is the color buffer that is ultimately written to the screen. When multiple objects occupy some of the same pixels in the color buffer, a determination must be made as to which pixel color(s) are retained, based on which object is nearest the viewer.

Hidden surface removal is done as follows:

- Before a scene is rendered, the depth buffer is filled with values representing maximum depth.
- As a pixel color is output by the fragment shader, its distance from the viewer is calculated.
- If the computed distance is *less than* the distance stored in the depth buffer (for that pixel), then: (a) the pixel color replaces the color in the color buffer, and (b) the distance replaces the value in the depth buffer. Otherwise, the pixel is discarded.

This procedure is called the *Z-buffer algorithm*, as expressed in Figure 2.14.

2.2 DETECTING OPENGL AND GLSL ERRORS

The workflow for compiling and running GLSL code differs from standard coding, in that *GLSL compilation happens at C++ runtime*. Another complication is that GLSL code doesn't run on the CPU (it runs on the GPU), so *the operating*

```
Color [ ] [ ] colorBuf = new Color [pixelRows][pixelCols];
double [ ] [ ] depthBuf = new double [pixelRows][pixelCols];
for (each row and column)   // initialize color and depth buffers
{   colorBuf [row][col] = backgroundColor;
    depthBuf [row][col] = far away;
}
for (each shape)    // update buffers when new pixel is closer
{   for (each pixel in the shape)
    {   if (depth at pixel < depthBuf value)
        {   depthBuf [pixel.row][pixel.col] = depth at pixel;
            colorBuf [pixel.row][pixel.col] = color at pixel;
}   }   }
return colorBuf;
```

Figure 2.14
Z-buffer algorithm.

system cannot always catch OpenGL runtime errors. This makes debugging difficult, because it is often hard to detect if a shader failed, and why.

Program 2.3 (which follows) presents some modules for catching and displaying GLSL errors. They make use of the OpenGL functions glGetShaderiv() and glGetProgramiv(), which are used to provide information about compiled GLSL shaders and programs. Accompanying them is the createShaderProgram() function from the previous Program 2.2, but with the error-detecting calls added.

Program 2.3 contains the following three utilities:

- **checkOpenGLError** – checks the OpenGL error flag for the occurrence of an OpenGL error
- **printShaderLog** – displays the contents of OpenGL's log when GLSL compilation failed
- **printProgramLog** – displays the contents of OpenGL's log when GLSL linking failed

The first, checkOpenGLError(), is useful for detecting both GLSL compilation errors and OpenGL runtime errors, so it is highly recommended to use it throughout a C++/OpenGL application during development. For example, in the prior example (Program 2.2), the calls to glCompileShader() and glLinkProgram() could easily be augmented with the code shown in Program 2.3 to ensure that any

typos or other compile errors would be caught and their cause reported. Calls to checkOpenGLError() could be added after runtime OpenGL calls, such as immediately after the call to glDrawArrays().

Another reason that it is important to use these tools is that *a GLSL error does not cause the C++ program to stop.* So unless the programmer takes steps to catch errors at the point that they happen, debugging will be very difficult.

Program 2.3 Modules to Catch GLSL Errors

```
void printShaderLog(GLuint shader) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);
        glGetShaderInfoLog(shader, len, &chWrittn, log);
        cout << "Shader Info Log: " << log << endl;
        free(log);
}   }

void printProgramLog(int prog) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);
        glGetProgramInfoLog(prog, len, &chWrittn, log);
        cout << "Program Info Log: " << log << endl;
        free(log);
}   }

bool checkOpenGLError() {
    bool foundError = false;
    int glErr = glGetError();
    while (glErr != GL_NO_ERROR) {
        cout << "glError: " << glErr << endl;
        foundError = true;
        glErr = glGetError();
    }
    return foundError;
}
```

Example of checking for OpenGL errors:

```
GLuint createShaderProgram() {
    GLint vertCompiled;
    GLint fragCompiled;
    GLint linked;
    . . .
    // catch errors while compiling shaders

    glCompileShader(vShader);
    checkOpenGLError();
    glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);
    if (vertCompiled != 1) {
        cout << "vertex compilation failed" << endl;
        printShaderLog(vShader);
    }

    glCompileShader(fShader);
    checkOpenGLError();
    glGetShaderiv(fShader, GL_COMPILE_STATUS, &fragCompiled);
    if (fragCompiled != 1) {
        cout << "fragment compilation failed" << endl;
        printShaderLog(fShader);
    }

    // catch errors while linking shaders

    glAttachShader(vfProgram, vShader);
    glAttachShader(vfProgram, fShader);

    glLinkProgram(vfProgram);
    checkOpenGLError();
    glGetProgramiv(vfProgram, GL_LINK_STATUS, &linked);
    if (linked != 1) {
        cout << "linking failed" << endl;
        printProgramLog(vfProgram);
    }
    return vfProgram;
}
```

There are other tricks for deducing the causes of runtime errors in shader code. A common result of shader runtime errors is for the output screen to be completely blank, essentially with no output at all. This can happen even if the error is a very small typo in a shader, yet it can be difficult to tell at which stage of the pipeline the error occurred. With no output at all, it's like looking for a needle in a haystack.

One useful trick in such cases is to temporarily replace the fragment shader with the one shown in Program 2.2. Recall that in that example, the fragment shader simply output a particular color—solid blue, for example. If the subsequent output is of the correct geometric form (but solid blue), the vertex shader is probably correct, and there is an error in the original fragment shader. If the output is still a blank screen, the error is more likely earlier in the pipeline, such as in the vertex shader.

In Appendix C, we show how to use yet another useful debugging tool called *Nsight*, which is available for machines equipped with certain Nvidia graphics cards.

2.3 READING GLSL SOURCE CODE FROM FILES

So far, our GLSL shader code has been stored inline in strings. As our programs grow in complexity, this will become impractical. We should instead store our shader code in files and read them in.

Reading text files is a basic C++ skill, and won't be covered here. However, for practicality, code to read shaders is provided in readShaderSource(), shown in Program 2.4. It reads the shader text file and returns an array of strings, where each string is one line of text from the file. It then determines the size of that array based on how many lines were read in. Note that here, createShaderProgram() replaces the version from Program 2.2.

In this example, the vertex and fragment shader code is now placed in the text files "vertShader.glsl" and "fragShader.glsl" respectively.

Program 2.4 Reading GLSL Source from Files

(….#includes same as before, main(), display(), init() as before, plus the following…)

```
#include <string>
#include <iostream>
#include <fstream>
. . .
string readShaderSource(const char *filePath) {
    string content;
    ifstream fileStream(filePath, ios::in);
    string line = "";
```

```
    while (!fileStream.eof()) {
        getline(fileStream, line);
        content.append(line + "\n");
    }
    fileStream.close();
    return content;
}

GLuint createShaderProgram() {
    (…… as before plus…..)
    string vertShaderStr = readShaderSource("vertShader.glsl");
    string fragShaderStr = readShaderSource("fragShader.glsl");

    const char *vertShaderSrc = vertShaderStr.c_str();
    const char *fragShaderSrc = fragShaderStr.c_str();

    glShaderSource(vShader, 1, &vertShaderSrc, NULL);
    glShaderSource(fShader, 1, &fragShaderSrc, NULL);

    (….etc., building rendering program as before)
}
```

2.4 BUILDING OBJECTS FROM VERTICES

Ultimately we want to draw more than just a single point. We'd like to draw objects that are constructed of *many* vertices. Large sections of this book will be devoted to this topic. For now we just start with a simple example—we will define *three* vertices and use them to draw a *triangle*.

We can do this by making two small changes to Program 2.2 (actually, the version in Program 2.4 which reads the shaders from files): (a) modify the vertex shader so that *three different* vertices are output to the subsequent stages of the pipeline, and (b) modify the glDrawArrays() call to specify that we are using *three* vertices.

In the C++/OpenGL application (specifically in the glDrawArrays() call) we specify GL_TRIANGLES (rather than GL_POINTS), and also specify that there are *three* vertices sent through the pipeline. This causes the vertex shader to run *three times*, and at each iteration, the built-in variable gl_VertexID is automatically incremented (it is initially set to 0). By testing the value of gl_VertexID, the shader is designed to output a different point each of the three times it is executed. Recall that the three points then pass through the *rasterization* stage, producing a filled-in triangle. The modifications are shown in Program 2.5 (the remainder of the code is the same as previously shown in Program 2.4).

Program 2.5 Drawing a Triangle

Vertex Shader

```
#version 430
void main(void)
{   if (gl_VertexID == 0) gl_Position = vec4( 0.25, -0.25, 0.0, 1.0);
    else if (gl_VertexID == 1) gl_Position = vec4(-0.25, -0.25, 0.0, 1.0);
    else gl_Position = vec4( 0.25, 0.25, 0.0, 1.0);
}
```

C++/OpenGL application—in display()

```
. . .
glDrawArrays(GL_TRIANGLES, 0, 3);
```

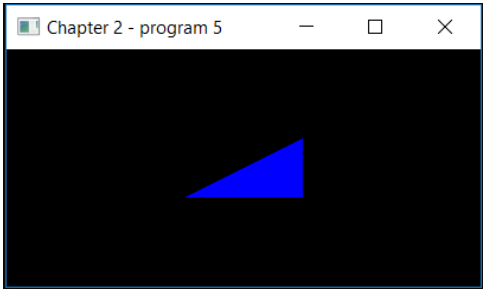

Figure 2.15
Drawing a simple triangle.

2.5 ANIMATING A SCENE

Many of the techniques in this book can be *animated*. This is when things in the scene are moving or changing, and the scene is rendered repeatedly to reflect these changes in real time.

Recall from Section 2.1.1 that we have constructed our main() to make a single call to init(), and then to call display() repeatedly. Thus, while each of the preceding examples may have appeared to be a single fixed rendered scene, in actuality the loop in the main was causing it to be drawn over and over again.

For this reason, our main() is already structured to support animation. We simply design our display() function to alter what it draws over time. Each rendering of our scene is then called a *frame*, and the frequency of the calls to display() is the *frame rate*. Handling the rate of movement within the application logic can be

controlled using the elapsed time since the previous frame (this is the reason for including "currentTime" as a parameter on the display() function).

An example is shown in Program 2.6. We have taken the triangle from Program 2.5 and animated it so that it moves to the right, then moves to the left, back and forth. In this example, we don't consider the elapsed time, so the triangle may move more or less quickly depending on the speed of the computer. In future examples, we will use the elapsed time to ensure that our animations run at the same speed regardless of the computer on which they are run.

In Program 2.6, the application's display() method maintains a variable "x" used to offset the triangle's X coordinate position. Its value changes each time display() is called (and thus is different for each frame), and it reverses direction each time it reaches 1.0 or -1.0. The value in x is copied to a corresponding variable called "offset" in the vertex shader. The mechanism that performs this copy uses something called a *uniform variable*, which we will study later in Chapter 4. It isn't necessary to understand the details of uniform variables yet. For now, just note that the C++/OpenGL application first calls glGetUniformLocation() to get a pointer to the "offset" variable, and then calls glProgramUniform1f() to copy the value of x into offset. The vertex shader then adds the offset to the X coordinate of the triangle being drawn. Note also that the background is cleared at each call to display(), to avoid the triangle leaving a trail as it moves. Figure 2.16 illustrates the display at three time instances (of course, the movement can't be shown in a still figure).

Program 2.6 Simple Animation Example

C++/OpenGL application:

```
// same #includes and declarations as before, plus the following:

float x = 0.0f;              // location of triangle on x axis
float inc = 0.01f;           // offset for moving the triangle

void display(GLFWwindow* window, double currentTime) {
    glClear(GL_DEPTH_BUFFER_BIT);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);      // clear the background to black, each time

    glUseProgram(renderingProgram);

    x += inc;                          // move the triangle along x axis
    if (x > 1.0f) inc = -0.01f;        // switch to moving the triangle to the left
```

```
        if (x < -1.0f) inc = 0.01f;                    // switch to moving the triangle to the right
        GLuint offsetLoc = glGetUniformLocation(renderingProgram, "offset");    // get ptr to "offset"
        glProgramUniform1f(renderingProgram, offsetLoc, x);          // send value in "x" to "offset"

        glDrawArrays(GL_TRIANGLES,0,3);
    }
    . . . // remaining functions, same as before
}
```

Vertex shader:

```
#version 430
uniform float offset;
void main(void)
{   if (gl_VertexID == 0) gl_Position = vec4( 0.25 + offset, -0.25, 0.0, 1.0);
    else if (gl_VertexID == 1) gl_Position = vec4(-0.25 + offset, -0.25, 0.0, 1.0);
    else gl_Position = vec4( 0.25 + offset, 0.25, 0.0, 1.0);
}
```

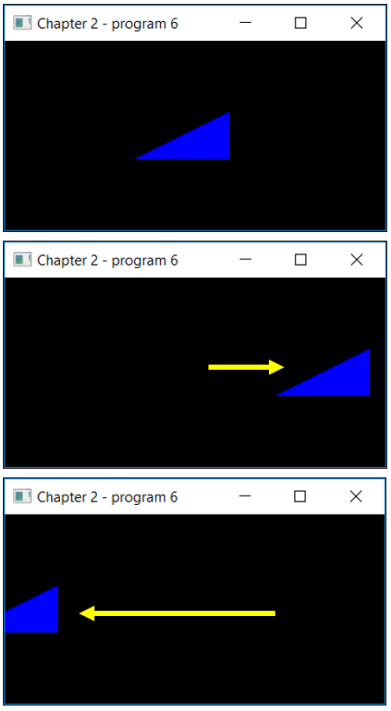

Figure 2.16
An animated, moving triangle.

Note that in addition to adding code to animate the triangle, we have also added the following line at the beginning of the display() function:

```
glClear(GL_DEPTH_BUFFER_BIT);
```

While not strictly necessary in this particular example, we have added it here and it will continue to appear in most of our applications. Recall from the discussion in Section 2.1.7 that *hidden surface removal* requires both a color buffer and a depth buffer. As we proceed to drawing progressively more complex 3D scenes, it will be necessary to initialize (clear) the depth buffer each frame, especially for scenes that are animated, to ensure that depth comparisons aren't affected by old depth data. It should be apparent from the previous example that

the command for clearing the depth buffer is essentially the same as for clearing the color buffer.

2.6 ORGANIZING THE C++ CODE FILES

So far, we have been placing all of the C++/OpenGL application code in a single file called "main.cpp", and the GLSL shaders into files called "vertShader.glsl" and "fragShader.glsl". While we admit that stuffing a lot of application code into main.cpp isn't a best practice, we have adopted this convention in this book so that it is absolutely clear in every example which file contains the main block of C++/OpenGL code relevant to the example being discussed. Throughout this textbook, it will always be called "main.cpp". In practice, applications should of course be modularized to appropriately reflect the tasks performed by the application.

However, as we proceed, there will be circumstances in which we create modules that will be useful in many different applications. Wherever appropriate, we will move those modules into separate files to facilitate reuse. For example, later we will define a Sphere class that will be useful in many different examples, and so it will be separated into its own files (Sphere.cpp and Sphere.h).

Similarly, as we encounter *functions* that we wish to reuse, we will place them in a file called "Utils.cpp" (and an associated "Utils.h"). We have already seen several functions that are appropriate to move into "Utils.cpp": the error-detecting modules described in Section 2.2, and the functions for reading in GLSL shader programs described in Section 2.3. The latter is particularly well-suited to overloading, such that a "createShaderProgram()" function can be defined for each possible combination of pipeline shaders assembled in a given application:

- GLuint Utils::createShaderProgram(const char *vp, const char *fp)
- GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp)
- GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES, const char *fp)
- GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES, const char *gp, const char *fp)

The first case in the previous list supports shader programs which utilize only a vertex and fragment shader. The second supports those utilizing vertex, geometry, and fragment shaders. The third supports those using vertex, tessellation,

and fragment shaders. The fourth supports those using vertex, tessellation, geometry, and fragment shaders. The parameters accepted in each case are pathnames for the GLSL files containing the shader code. For example, the following call uses one of the overloaded functions to compile and link a shader pipeline program that includes a vertex and fragment shader. The completed program is placed in the variable "renderingProgram":

```
renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");
```

These createShaderProgram() implementations can all be found on the accompanying CD (in the "Utils.cpp" file), and all of them incorporate the error-detecting modules from Section 2.2 as well. There is nothing new about them; they are simply organized in this way for convenience. As we move forward in the book, other similar functions will be added to Utils.cpp as we go along. The reader is strongly encouraged to examine the Utils.cpp file on the accompanying CD, and even add to it as desired. The programs found there are built from the methods as we learn them in the book, and studying their organization should serve to strengthen one's own understanding.

Regarding the functions in the "Utils.cpp" file, we have implemented them as static methods so that it isn't necessary to instantiate the Utils class. Readers may prefer to implement them as instance methods rather than static methods, or even as freestanding functions, depending on the architecture of the particular system being developed.

All of our shaders will be named with a ".glsl" extension.

SUPPLEMENTAL NOTES

There are many details of the OpenGL pipeline that we have not discussed in this introductory chapter. We have skipped a number of internal stages and have completely omitted how *textures* are processed. Our goal was to map out, as simply as possible, the framework in which we will be writing our code. As we proceed we will continue to learn additional details.

We have also deferred presenting code examples for tessellation and geometry. In later chapters, we will build complete systems that show how to write practical shaders for each of the stages.

There are more sophisticated ways to organize the code for animating a scene, especially with respect to managing threads. Some language bindings such as JOGL and LW**JGL** (for Java) offer classes to support animation. Readers interested in designing a render loop (or "game loop") appropriate for a particular application are encouraged to consult some of the more specialized books on game engine design (e.g., [NY14]), and to peruse the related discussions on gamedev.net [GD20].

We glossed over one detail on the glShaderSource() command. The fourth parameter is used to specify a "lengths array" that contains the integer string lengths of each line of code in the given shader program. If this parameter is set to null, as we have done, OpenGL will build this array automatically if the strings are null-terminated. While we have been careful to ensure that our strings sent to glShaderSource() are null-terminated (by calling the c_str() function in createShaderProgram()), it is not uncommon to encounter applications that build these arrays manually rather than sending null.

Throughout this book, the reader may at times wish to know one or more of OpenGL's upper limits. For example, the programmer might need to know the maximum number of outputs that can be produced by the geometry shader, or the maximum size that can be specified for rendering a point. Many such values are implementation-dependent, meaning that they can vary between different machines. OpenGL provides a mechanism for retrieving such limits using the glGet() command, which takes various forms depending on the type of the parameter being queried. For example, to find the maximum allowable point size, the following call will place the minimum and maximum values (for your machine's OpenGL implementation) into the first two elements of the float array named "size":

```
glGetFloatv(GL_POINT_SIZE_RANGE, size)
```

Many such queries are possible. Consult the OpenGL reference [OP16] documentation for examples.

In this chapter, we have tried to describe each parameter on each OpenGL call. However, as the book proceeds, this will become unwieldy and we will sometimes not bother describing a parameter when we believe that doing so would complicate matters unnecessarily. This is because many OpenGL functions have a large number of parameters that are irrelevant to our examples. The reader should get used to using the OpenGL documentation to fill in such details when necessary.

Exercises

2.1 Modify Program 2.2 to add animation that causes the drawn point to grow and shrink, in a cycle. Hint: use the glPointSize() function, with a variable as the parameter.

2.2 Modify Program 2.5 so that it draws an isosceles triangle (rather than the right triangle shown in Figure 2.15).

2.3 *(PROJECT)* Modify Program 2.5 to include the error-checking modules shown in Program 2.3. After you have that working, try inserting various errors into the shaders and observing both the resulting behavior and the error messages generated.

2.4 Modify Program 2.6 so that it calculates the amount of movement for the triangle using the "currentTime" variable passed into the display() function. Hint: the "currentTime" variable contains the total time that has elapsed since the program began. Your solution will need to determine the time that has elapsed since the last frame, and compute the increment amount based on that. Computing animations in this way will ensure that they move at the same speed regardless of the speed of the computer.

References

[GD20] Game Development Network, accessed July 2020, https://www.gamedev.net/

[NY14] R. Nystrom, "Game Loop," in *Game Programming Patterns* (Genever Benning, 2014), and accessed July 2020, http://gameprogrammingpatterns.com/game-loop.html

[OP16] OpenGL 4.5 Reference Pages, accessed July 2016, https://www.khronos.org/registry/OpenGL-Refpages/gl4/

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).