

COMPUTER GRAPHICS PROGRAMMING

IN OPENGL WITH JAVA



INCLUDES DISC



V. SCOTT GORDON AND JOHN CLEVENGER

COMPUTER GRAPHICS

PROGRAMMING IN OPENGL

WITH JAVA

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the “Work”), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book 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 disc, 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 authors, 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 authors, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, 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.

Companion disc files are available for download from the publisher by writing to info@merclearning.com.

COMPUTER GRAPHICS

PROGRAMMING IN OPENGL

WITH JAVA

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

Copyright ©2017 by MERCURY LEARNING AND INFORMATION LLC. All rights reserved.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai

MERCURY LEARNING AND INFORMATION
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

V. Scott Gordon & John Clevenger
Computer Graphics Programming in OpenGL with Java
ISBN: 978-1-683920-27-4

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2016962393

171819321 Printed in the United States of America on acid-free paper

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.authorcloudware.com and other e-vendors. *All companion files are available by writing to the publisher at info@merclearning.com.*

The sole obligation of MERCURY LEARNING AND INFORMATION to the purchaser is to replace the book and/or disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

Contents

<i>Preface</i>	<i>xi</i>
<i>Intended Audience</i>	<i>xiii</i>
<i>How to Use This Book</i>	<i>xiii</i>
<i>Acknowledgments</i>	<i>xv</i>
<i>About the Authors</i>	<i>xv</i>
Chapter 1 Getting Started	1
1.1 Languages and Libraries	1
1.1.1 Java	2
1.1.2 OpenGL / GLSL	2
1.1.3 JOGL	3
1.1.4 graphicslib3D	3
1.2 Installation and Configuration	3
1.2.1 Installing Java	3
1.2.2 Installing OpenGL / GLSL	4
1.2.3 Installing JOGL	4
1.2.4 Installing graphicslib3D	4
Chapter 2 JOGL and the OpenGL Graphics Pipeline	7
2.1 The OpenGL Pipeline	8
2.1.1 Java/JOGL Application	9
2.1.2 Vertex and Fragment Shaders	12
2.1.3 Tessellation	17

2.1.4	Geometry Shader	18
2.1.5	Rasterization	19
2.1.6	Fragment Shader	20
2.1.7	Pixel Operations	21
2.2	Detecting OpenGL and GLSL Errors	23
2.3	Reading GLSL Source Code from Files	27
2.4	Building Objects from Vertices	29
2.5	Animating a Scene	30
Chapter 3 Mathematical Foundations		35
3.1	3D Coordinate Systems	36
3.2	Points	36
3.3	Matrices	37
3.4	Transformation Matrices	40
3.4.1	Translation	40
3.4.2	Scaling	41
3.4.3	Rotation	42
3.5	Vectors	44
3.5.1	Uses for Dot Product	45
3.5.2	Uses for Cross Product	47
3.6	Local and World Space	48
3.7	Eye Space and the Synthetic Camera	49
3.8	Projection Matrices	52
3.8.1	The Perspective Projection Matrix	52
3.8.2	The Orthographic Projection Matrix	54
3.9	Look-At Matrix	56
3.10	GLSL Functions for Building Matrix Transforms	57
Chapter 4 Managing 3D Graphics Data		63
4.1	Buffers & Vertex Attributes	64
4.2	Uniform Variables	67
4.3	Interpolation of Vertex Attributes	68
4.4	Model-View and Perspective Matrices	69
4.5	Our First 3D Program – a 3D Cube	71
4.6	Rendering Multiple Copies of an Object	80
4.6.1	Instancing	81
4.7	Rendering Multiple Different Models in a Scene	85
4.8	Matrix Stacks	87

4.9	Combating “Z-Fighting” Artifacts	94
4.10	Other Options for Primitives	95
4.11	Back-Face Culling	97
Chapter 5 Texture Mapping		103
5.1	Loading Texture Image Files	104
5.2	Texture Coordinates	106
5.3	Creating a Texture Object	108
5.4	Constructing Texture Coordinates	108
5.5	Loading Texture Coordinates into Buffers	110
5.6	Using the Texture in a Shader: Sampler Variables and Texture Units	110
5.7	Texture Mapping: Example Program	111
5.8	Mipmapping	114
5.9	Anisotropic Filtering	120
5.10	Wrapping and Tiling	121
5.11	Perspective Distortion	123
5.12	Loading Texture Image Files Using Java AWT Classes	124
Chapter 6 3D Models		129
6.1	Procedural Models – Building a Sphere	130
6.2	OpenGL Indexing – Building a Torus	137
6.2.1	The Torus	137
6.2.2	Indexing in OpenGL	139
6.3	Loading Externally Produced Models	144
Chapter 7 Lighting		157
7.1	Lighting Models	157
7.2	Lights	159
7.3	Materials	163
7.4	ADS Lighting Computations	165
7.5	Implementing ADS Lighting	168
7.5.1	Gouraud Shading	169
7.5.2	Phong Shading	177
7.6	Combining Lighting and Textures	182
Chapter 8 Shadows		189
8.1	The Importance of Shadows	189
8.2	Projective Shadows	190

8.3	Shadow Volumes	191
8.4	Shadow Mapping	192
8.4.1	Shadow Mapping (PASS ONE) – “Draw” Objects from Light Position	193
8.4.2	Shadow Mapping (Intermediate Step) – Copying the Z-Buffer to a Texture	194
8.4.3	Shadow Mapping (PASS TWO) – Rendering the Scene with Shadows	195
8.5	A Shadow Mapping Example	199
8.6	Shadow Mapping Artifacts	205
Chapter 9	Sky and Backgrounds	211
9.1	Skyboxes	211
9.2	Skydomes	214
9.3	Implementing a Skybox	215
9.3.1	Building a Skybox from Scratch	216
9.3.2	Using OpenGL Cube Maps	219
9.4	Environment Mapping	223
Chapter 10	Enhancing Surface Detail	231
10.1	Bump Mapping	232
10.2	Normal Mapping	233
10.3	Height Mapping	242
Chapter 11	Parametric Surfaces	249
11.1	Quadratic Bézier Curves	249
11.2	Cubic Bézier Curves	251
11.3	Quadratic Bézier Surfaces	254
11.4	Cubic Bézier Surfaces	256
Chapter 12	Tessellation	261
12.1	Tessellation in OpenGL	261
12.2	Tessellation for Bézier Surfaces	267
12.3	Tessellation for Terrain / Height Maps	273
12.4	Controlling Level of Detail (LOD)	281

Chapter 13 Geometry Shaders	287
13.1 Per-Primitive Processing in OpenGL	287
13.2 Altering Primitives	288
13.3 Deleting Primitives	293
13.4 Adding Primitives	294
Chapter 14 Other Techniques	299
14.1 Fog	299
14.2 Compositing / Blending / Transparency	302
14.3 User-Defined Clipping Planes	308
14.4 3D Textures	310
14.5 Noise	316
14.6 Noise Application – Marble	321
14.7 Noise Application – Wood	323
14.8 Noise Application – Clouds	327
14.9 Noise Application – Special Effects	332
Index	337

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 wanted 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 simply as we can. Every programming example is stripped-down and simplified as much as possible, but 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 a beginner—that is, 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.

Another thing that makes this book unique is that it teaches OpenGL programming in *Java*, using *Jogl*—a Java “wrapper” for OpenGL’s native C calls [JO16]. There are several advantages to learning graphics programming in Java rather than in C:

- It is more convenient for students at schools that conduct most of their curriculum in Java.
- Java’s I/O, window, and event handling are arguably cleaner than in C.
- Java’s excellent support for object-oriented design patterns can foster good design.

It is worth mentioning that there do exist other Java bindings for OpenGL. One that is becoming increasingly popular is Lightweight Java Game Library, or LWJGL [LW16]. Like JOGL, LWJGL also offers bindings for OpenAL and OpenCL. This textbook focuses only on JOGL.

Another 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. The JOGL suite of Java bindings includes interfaces for different versions of OpenGL ES, although we do not use them in this book.

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 JavaScript¹ 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 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 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 or JOGL “reference.” Our coverage of OpenGL and JOGL represents only a tiny fraction of their capabilities. Rather, our aim is to

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.

use OpenGL and JOGL as vehicles for teaching the fundamentals of modern shader-based 3D graphics programming, and provide the reader with a sufficiently deep understanding for further study. If along the way this text helps to expand awareness of JOGL and other JogAmp technologies, that would be nice too.

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, but that we don't cover because we assume the reader already has sufficient background. In particular:

- Java and its Abstract Window Toolkit (AWT) or Swing library, especially for GUI-building
- Java configuration details, such as manipulating the CLASSPATH
- event-driven programming
- basic matrix algebra and trigonometry
- awareness of color models, such as RGB, RGBA, etc.

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 forwards 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 problems 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, whether made in C or in Java through JOGL, 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 at the very beginning, we would describe every detail. But 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 Java, OpenGL, and JOGL.

For this, there are a number of excellent reference sources that we recommend using in conjunction with this book. The *javadocs* for Java and JOGL are *absolutely essential*, and can be accessed online or downloaded (we explain in Chapter 1 how to download the JOGL javadoc). The reader should bookmark them for easy access in a browser, and *expect to access them continuously* for looking up items such as parameter and constructor details. The URLs for the Java and JOGL javadocs are:

<https://docs.oracle.com/javase/8/docs/api/>
<https://jogamp.org/deployment/webstart/javadoc/jogl/javadoc>

Many of the entries in the JOGL javadoc are simply pointers to the corresponding entry in the OpenGL documentation, available here:

<https://www.opengl.org/sdk/docs/man/>

Our examples utilize a mathematics library called `graphicslib3D`. This is a Java library that also has its own set of javadocs included. After installing `graphicslib3D` (described in Chapter 1), the reader should locate the accompanying javadoc folder and bookmark its root file (`index.html`).

Finally, 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* [WO13]
- (Angel and Shreiner) *Interactive Computer Graphics* [AS14]
- (Luna) *Introduction to 3D Game Programming with DirectX 12* [LU16]

Acknowledgments

Early drafts of this book were used in the CSc-155 (Advanced Computer Graphics Programming) course at CSU Sacramento, and benefited from many student corrections and comments (and in some cases, code). The authors would particularly like to thank 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 for their suggestions. We apologize if we have missed anyone.

We are extremely grateful for the invaluable advice provided to us by Julien Gouesse, engine support maintainer at Jogamp. Mr. Gouesse provided technical information on JOGL textures, cube maps, buffer handling, and proper loading of shader source files that led to improvements in our text.

Jay Turberville of Studio 522 Productions in Scottsdale (Arizona) built the dolphin model shown on the cover and used throughout this book. Studio 522 Productions does incredibly high-quality 3D animation and video production, as well as custom 3D modeling. We were thrilled that Mr. Turberville kindly offered to build such a wonderful new model just for this book project.

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].

Dr. Danny Kopec connected us with Mercury Learning and introduced us to its publisher, David Pallai. Dr. Kopec’s *Artificial Intelligence* textbook inspired us to consider Mercury, and our telephone conversations with him were extremely informative. We were deeply saddened to hear of Dr. Kopec’s untimely passing, and regret that he didn’t have the chance to see our book come to fruition.

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

About the Authors

Dr. V. Scott Gordon has been a professor in the California State University system for over twenty 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 video and strategy game programming, artificial intelligence, neural networks, software engineering, 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 this textbook. He is the technical director of the ACM International Collegiate Programming Contest, 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).
- [JO16] Jogamp, accessed July 2016, <http://jogamp.org/>.
- [LW16] Lightweight Java Game Library (**LWJGL**), accessed July 2016, <https://www.lwjgl.org/>.
- [LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).
- [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).
- [SW15] G. Sellers, R. Wright Jr., and N. Haemel., *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).
- [WO13] D. Wolff, *OpenGL Shading Language Cookbook*, 2nd ed. (Packt Publishing, 2013).

GETTING STARTED

1.1	<i>Languages and Libraries</i>	<i>1</i>
1.2	<i>Installation and Configuration</i>	<i>3</i>



Graphics programming has a reputation of 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 Java or C++ 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!

1.1 LANGUAGES AND LIBRARIES

Running the programs in this book requires the use of the following four languages and libraries:

- Java
- OpenGL / GLSL
- JOGL
- graphicslib3D

It is likely that the reader will need to do a few preparatory steps to ensure that each of these are installed and properly accessible on his/her system. In this section we briefly describe each of them. Then in the following section we show how to install and/or configure them for use.

1.1.1 Java

Java was developed at Sun Microsystems in the early 1990s, and the first stable release of a development kit (JDK) was in 1995. In 2010, Oracle Corp. acquired Sun, and has maintained Java since that time [OR16]. This book assumes Java version 8, which was released in 2014.

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” [GV16].

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

1.13 JOGL

JOGL first appeared in 2003, published on the website Java.net. Since 2010 it has been an independent open source project, and part of a suite of Java bindings maintained by *JogAmp* [JO15], an online community of developers. JogAmp also maintains JOAL and JOCL, bindings for OpenAL and OpenCL, respectively. As new versions of OpenGL and/or Java are released, new versions of JOGL are developed to support continued compatibility. JogAmp also maintains a short online user’s guide that includes valuable guidelines for installing and using JOGL efficiently and effectively [JU16]. This book assumes at least version 2.3 of JOGL.

1.14 graphicslib3D

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 functions. For example, the popular OpenGL Superbible [SW15] utilizes a C library called “*vecmath*”. In this book, we use a Java library called *graphicslib3D*.

graphicslib3D provides classes for basic math functions related to graphics concepts, such as *vector*, *matrix*, *point*, *vertex*, and *quaternion*. It also contains a few utility classes for storing frequently used 3D graphics structures, such as a stack for building hierarchical structures, lighting and material information, and a few basic shapes such as a sphere and a torus.

graphicslib3D was first developed in 2005 by John Clevenger at California State University Sacramento, and is maintained by the authors.

1.2 INSTALLATION AND CONFIGURATION

1.2.1 Installing Java

To use Java for the examples in this book, you will need both the JRE (Java Runtime Environment) and the JDK (Java Development Kit). To install them, use Oracle’s download site, <http://www.oracle.com/technetwork/java>, and click the “Java SE” (Standard Edition) link under “Software Downloads.” From there you can find instructions for downloading the latest JDK, which includes both the Java compiler and the JRE. It is assumed that the reader is experienced with programming in Java. At the time of this writing, the current version of Java is version 8.

1.2.2 Installing OpenGL / GLSL

It is not necessary to “install” OpenGL or GLSL, but it is necessary to ensure that your graphics card supports at least version 4.3 of OpenGL. If you do not know what version of OpenGL your machine supports, you can use one of the various free applications (such as GLView [GV16]) to find out.

1.2.3 Installing JOGL

To install JOGL, visit <http://jogamp.org>. As of this writing, the current version of JOGL is in the “Builds/Downloads” section—look under “Current” and click on [zip]. This displays the latest stable JOGL files in a folder named “/deployment/jogamp-current/archive”. Download the following:

```
jogamp-all-platforms.7z  
jogl-javadoc.7z
```

Unzip these files into the folder on your machine where you would like to store the JOGL system. A typical location in Windows could be, for example, in a folder at the root of the C: drive.

The unzipped “jogamp-all-platforms” file contains a folder named “jar”, which contains two important files that will be used by your applications:

```
jogl-all.jar  
gluegen-rt.jar
```

Add the full path name of each of these two files to your CLASSPATH environment variable.

In the jogl-javadoc folder, double-click the file named index.html. This opens the JOGL javadocs in a browser, which you should then bookmark.

1.2.4 Installing graphicslib3D



To install graphicslib3D, download the graphicslib3D.zip file from the textbook support website or the accompanying disc. Unzipping this file produces a folder containing the following two items:

- A file named `graphicslib3D.jar`
- A folder named `javadoc`

Move these files to wherever you would like to store `graphicslib3D`—a typical location in Windows could be, for example, in a folder at the root of the C: drive.

Add the full path name of the `.jar` file to your `CLASSPATH` environment variable.

Open the `index.html` file in the `javadoc` folder, and bookmark it as you did for the JOGL Javadoc.

References

- [GV16] GLView, realtech-vr, accessed July 2016, <http://www.realtech-vr.com/glview/>.
- [JO16] Jogamp, accessed July 2016, <http://jogamp.org/>.
- [JU16] JOGL Users Guide, accessed July 2016, <https://jogamp.org/jogl/doc/userguide/>.
- [OR16] Java Software, Oracle Corp., accessed July 2016, <https://www.oracle.com/java/index.html>.
- [SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

JOGL AND THE OPENGL GRAPHICS PIPELINE

2.1	<i>The OpenGL Pipeline</i>	8
2.2	<i>Detecting OpenGL and GLSL Errors</i>	23
2.3	<i>Reading GLSL Source Code from Files</i>	27
2.4	<i>Building Objects from Vertices</i>	29
2.5	<i>Animating a Scene</i>	30
	<i>Supplemental Notes</i>	32



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++. However, 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 the popular Java wrapper **JOGL** (Java OpenGL). When using JOGL, the programmer writes a Java program that runs on the CPU (more specifically, on the Java Virtual Machine, or JVM) and includes JOGL (and thus, OpenGL) calls. We will refer to a Java program that contains JOGL calls as a *Java/JOGL application*. One important task of a Java/JOGL application is to install the programmer’s GLSL code onto the GPU.

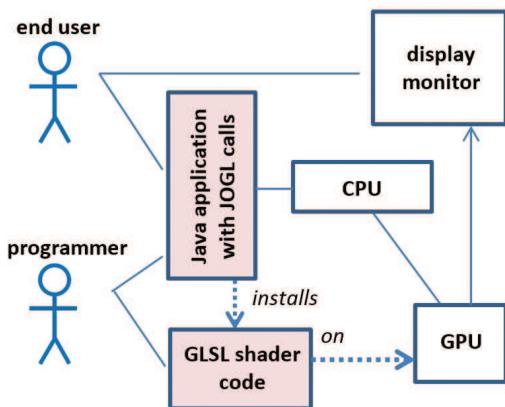


Figure 2.1
Overview of a JOGL-based graphics application.

An overview of a JOGL-based graphics application is shown in Figure 2.1, with the software components highlighted in red:

Some of the code we will write will be in Java, with JOGL calls, and some will be written in GLSL. Our Java/JOGL 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 Java 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 Java/JOGL 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 just to 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 Java/JOGL 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 Java/JOGL application to load GLSL programs into these shader stages, as follows:

1. It uses Java to obtain the GLSL code, either from text files or hardcoded 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.

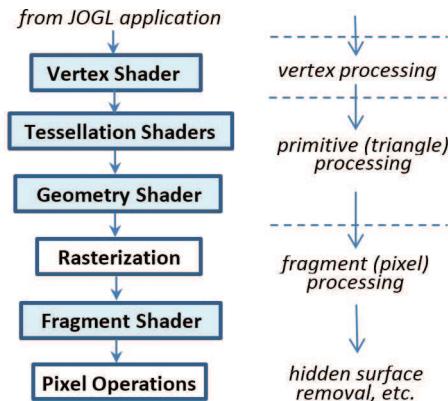


Figure 2.2
Overview of the OpenGL pipeline.

2.1.1 Java/JOGL Application

The bulk of our graphics application is written in Java. Depending on the purpose of the program, it may interact with the end user using standard Java libraries such as AWT or Swing. For tasks related to 3D rendering, it uses the JOGL library. Other windowing libraries exist that interface with JOGL, such as SWT and NEWT, that have some performance advantages; in this book, however, we use AWT and Swing because of the likelihood the reader already has familiarity with them.

JOGL includes a class called `GLCanvas` that is compatible with the standard Java `JFrame`, and on which we can draw 3D scenes. As already mentioned, JOGL also gives us commands for installing GLSL programs onto the programmable shader stages and compiling them. Finally, JOGL uses *buffers* for sending 3D models and other related graphics data down the pipeline.

Before we try writing shaders, let's write a simple Java/JOGL application that instantiates a `GLCanvas` and sets its background color. Doing that won't require any shaders at all! The code is shown in Program 2.1 (translated to Java from the C++ version in [SW15]). It extends `JFrame`, and instantiates a `GLCanvas`, adding it to the `JFrame`. It also implements `GLEventListener`, required to utilize OpenGL—this necessitates implementing some methods, specifically `display()`, `init()`, `reshape()`, and `dispose()`. The `display()` method is where we place code that draws to the `GLCanvas`. In this example, we load a `FloatBuffer` with values $(1,0,0,1)$, corresponding to the RGB values of the color red (plus a " 1 " for the opacity component), and use the OpenGL call `glClearBuffer()` to fill the display buffer with that color.

Program 2.1 First Java/JOGL Application

```

import java.nio.*;
import javax.swing.*;
import static com.jogamp.opengl.GL4.*;
import com.jogamp.opengl.*;
import com.jogamp.opengl.awt.GLCanvas;
import com.jogamp.common.nio.Buffers;

public class Code extends JFrame implements GLEventListener
{ private GLCanvas myCanvas;

    public Code()
    { setTitle("Chapter2 - program1");
        setSize(600, 400);
        setLocation(200, 200);
        myCanvas = new GLCanvas();
        myCanvas.addGLEventListener(this);
        this.add(myCanvas);
        setVisible(true);
    }

    public void display(GLAutoDrawable drawable)
    { GL4 gl = (GL4) GLContext.getCurrentGL();
        float bkg[] = { 1.0f, 0.0f, 0.0f, 1.0f };
        FloatBuffer bkgBuffer = Buffers.newDirectFloatBuffer(bkg);
        gl.glClearBufferfv(GL_COLOR, 0, bkgBuffer);
    }

    public static void main(String[ ] args)
    { new Code();
    }

    public void init(GLAutoDrawable drawable) { }
    public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) { }
    public void dispose(GLAutoDrawable drawable) { }
}

```

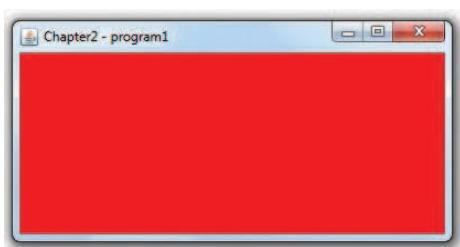


Figure 2.3
Output of Program 2-1.

When running a Java/JOGL application (such as the above Program 2.1) on a Microsoft Windows machine, it is advisable to add the command-line option to disable the use of Direct3D acceleration, such as:

```
java -Dsun.java2d.d3d=false Code
```

The mechanism by which these functions are deployed is as follows: When a GLCanvas

is made “visible” (by our calling “`setVisible(true)`” on the `JFrame` that contains it), it initializes OpenGL, which in turn creates a “`GL4`” object that our application can use for making OpenGL function calls. OpenGL then does a “callback,” calling `init()`, and passes it a “drawable” object (in this case the drawable object is the `GLCanvas`, although that isn’t obvious from the code). In this particular example, `init()` doesn’t do anything—in most applications it is where we would read in GLSL code, load 3D models, and so on. OpenGL next calls `display()`, also sending it the drawable object. It is typical to immediately obtain the `GL4` object and put it in a variable called “`gl`”. (Actually, `GL4` is an interface—in practice we don’t need to know the actual GL object class).

Later we will see that if we want our scene to be animated, our Java/JOGL application will need to tell OpenGL to make additional calls to `display()`.

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

```
gl.glClearBufferfv(GL_COLOR, 0, bkgBuffer);
```

Since JOGL is a Java binding for OpenGL, that means that calls to JOGL in turn generate calls to OpenGL’s library of C functions. In this case, the C function being called, as described in the OpenGL reference documentation (available on the web at <https://www.opengl.org/sdk/docs>) is:

```
glClearBufferfv(GLenum buffer, GLint drawbuffer, const GLfloat * value);
```

The first thing to notice is that the name of the JOGL function is the same as that of the original OpenGL C function, except it is preceded by “`gl.`”, which is the name of the `GL4` object. The period “`.`” after the “`gl`” is significant because “`gl`” is the object on which we are invoking the OpenGL function.

To reiterate, `GL4` is a Java interface to the OpenGL functions. We can obtain it in one of two ways: (a) by calling `drawable.getGL()`, utilizing the “`GLAutoDrawable`” object provided automatically when the various `GLEventListener` functions are invoked (called back) by OpenGL, or (b) by calling `GLContext.getCurrentGL()` as done in Program 2.1. Obtaining the current `GL4` object is important because, in general, any OpenGL function described in the OpenGL documentation can be called from JOGL by preceding it with the name of the appropriate `GL4` object (such as “`gl.`”, here).

The next detail to notice is that the first parameter makes reference to a “`GLenum`”. OpenGL has many predefined constants (enums); this one references the *color buffer* `GL_COLOR` that contains the pixels as they are rendered. In actuality, there is more

than one color buffer, and the second parameter is used to specify which one we are using (in this case, the 0th or first one).

Next, notice that the third parameter is defined in OpenGL as a *C pointer*. While JOGL makes every effort to match the original OpenGL C calls, Java does not have pointers. Any parameter in OpenGL that is a pointer is changed in JOGL. In this case, the JOGL version utilizes a `FloatBuffer` instead. Whenever there is a discrepancy between a JOGL call and the original C call, consult the JOGL Java-doc for details on the revised parameter(s).¹

Immediately before the call to `glClearBufferfv()` is a call to `newDirectFloatBuffer()`. We will learn about this function later in Chapter 4 when we discuss buffers and the JOGL buffer tools.

Finally, besides `display()` and `init()`, we also must implement `reshape()` and `dispose()`. The `reshape()` function is called when a `GLCanvas` is resized, and `dispose()` is called when the application exits. In Program 2.1 we left them both empty.

2.1.2 Vertex and Fragment Shaders

Our first JOGL 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 Java/JOGL application, or they can be hardcoded in the Java code or even in the GLSL code.

¹ In this example, we have described each parameter in the call. However, as the book proceeds, we will sometimes not bother describing a parameter when we believe that doing so would complicate matters unnecessarily. The reader should get used to using the JOGL/OpenGL documentation to fill in such details as necessary.

Before any of this can happen, the Java/JOGL 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 application also is responsible for telling OpenGL to construct triangles. We do this by using JOGL to call 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

```
(.....imports as before)
public class Code extends JFrame implements GLEventListener
{ private int rendering_program; } new declarations
private int vao[ ] = new int[1];

public Code() { .....constructor as before }
public void display(GLAutoDrawable drawable)
{ GL4 gl = (GL4) GLContext.getCurrentGL();
  gl.glUseProgram(rendering_program);
  gl.glDrawArrays(GL_POINTS, 0, 1);
}

public void init(GLAutoDrawable drawable)
{ GL4 gl = (GL4) GLContext.getCurrentGL();
  rendering_program = createShaderProgram();
```

```

gl glGenVertexArrays(vao.length, vao, 0);
gl glBindVertexArray(vao[0]);
}

private int createShaderProgram()
{ GL4 gl = (GL4) GLContext.getCurrentGL();

String vshaderSource[ ] =
{ "#version 430\n",
  "void main(void)\n",
  "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }\n",
};

String fshaderSource[ ] =
{ "#version 430\n",
  "out vec4 color;\n",
  "void main(void)\n",
  "{ color = vec4(0.0, 0.0, 1.0, 1.0); }\n",
};

int vShader = gl.glCreateShader(GL_VERTEX_SHADER);
gl.glShaderSource(vShader, 3, vshaderSource, null, 0); // note: 3 lines of code
gl.glCompileShader(vShader);

int fShader=gl.glCreateShader(GL_FRAGMENT_SHADER);
gl.glShaderSource(fShader, 4, fshaderSource, null, 0); // note: 4 lines of code
gl.glCompileShader(fShader);

int vfprogram = gl.glCreateProgram();
gl.glAttachShader(vfprogram, vShader);
gl.glAttachShader(vfprogram, fShader);
gl.glLinkProgram(vfprogram);

gl.glDeleteShader(vShader);
gl.glDeleteShader(fShader);
return vfprogram;
}
... main(), reshape(), and dispose() as before

```

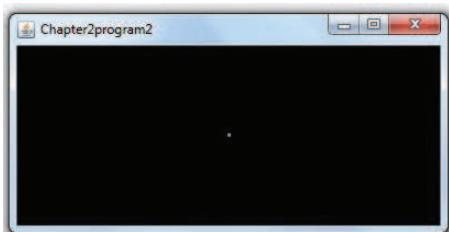


Figure 2.4
Output of Program 2.2.

The program appears to have output a blank canvas. 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.

There are many important details in Program 2.2 (color-coded in the program, for

convenience) for us to discuss. First, note that `init()` is no longer empty—it now calls another function (that we wrote) named “`createShaderProgram()`”. This function starts by declaring two shaders as arrays of strings called `vshaderSource` and `fshaderSource`. It then calls `glCreateShader()`, which generates the desired type of shader (note the predefined value `GL_VERTEX_SHADER`, and then later `GL_FRAGMENT_SHADER`). OpenGL creates the shader object (initially empty), and returns an integer ID that is an index for referencing it later—the code stores this ID in the variable `vShader` (and `fShader`). It then calls `glShaderSource()`, which loads the GLSL code from the string array into the empty shader object. `glShaderSource()` has five parameters: (a) the shader object in which to load the shader, (b) the number of strings in the shader source code, (c) the array of strings containing the source code, and two additional parameters we aren’t using (they will be explained later, in the supplementary chapter notes). Note also the two commented lines of code in the blue section, highlighting the parameter values 3 and 4—these refer to the number of lines of code in the shader (the `\n`s delineate each line in the shader source code). The shaders are then each compiled using `glCompileShader()`.

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 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.

After `init()` finishes, `display()` is called (recall this is also an OpenGL callback). One of the first things that `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 Java/JOGL 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 ahead). As we saw, they have been declared in the Java/JOGL 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 4.30. There follows a “main” function (as we will see, GLSL is somewhat Java-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 vertex moves through the pipeline, eventually reaching 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 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:

```
gl.glPointSize(30.0f);
```

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

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:

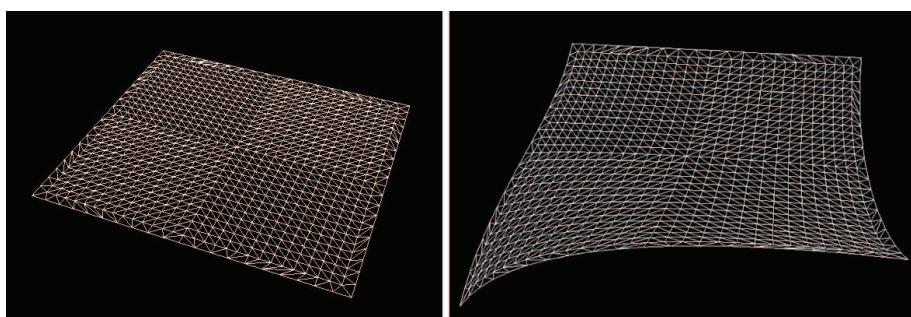


Figure 2.6

Grid produced by tessellator.

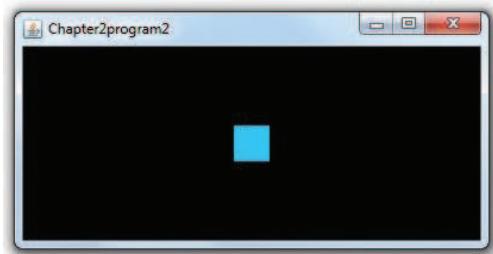


Figure 2.5

Changing `glPointSize`.

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

2.14 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 “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 long, narrow triangles that face outward, we get the result shown in Figure 2.8. This “furry torus” would be computationally expensive to try and model from scratch in the Java/JOGL 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

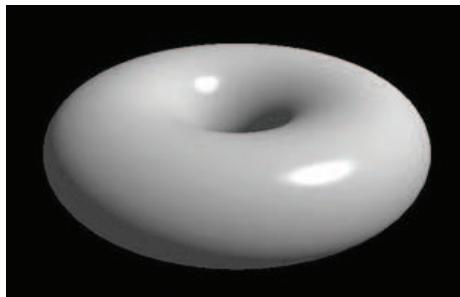


Figure 2.7
Torus model.

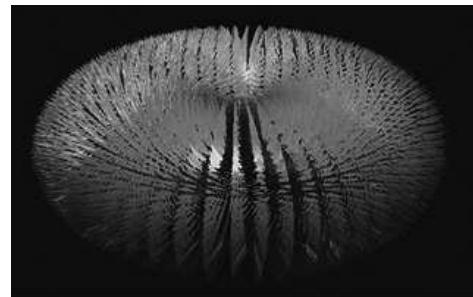


Figure 2.8
Torus modified in geometry shader.

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 Java through a buffer.

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,

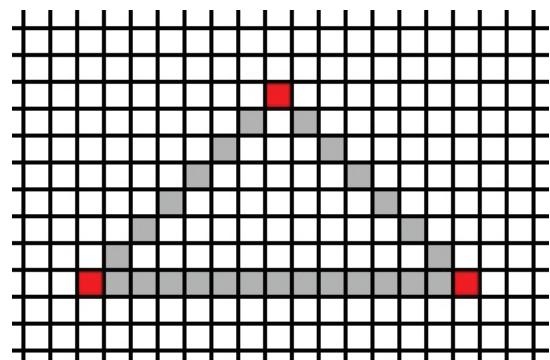


Figure 2.9
Rasterization (step 1).

by adding the following command in the `display()` function, before the call to `glDrawArrays()`:

```
gl.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.

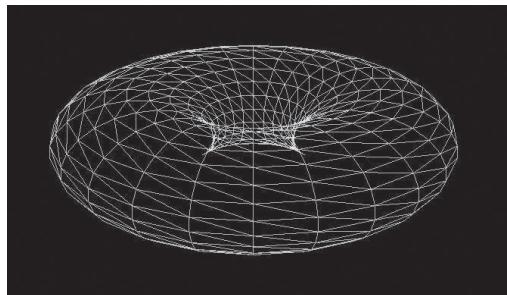


Figure 2.10
Torus with wireframe rendering.

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 solid torus shown in Figure 2.12.

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,

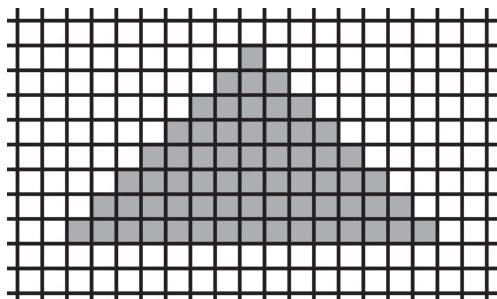


Figure 2.11
Fully rasterized triangle.

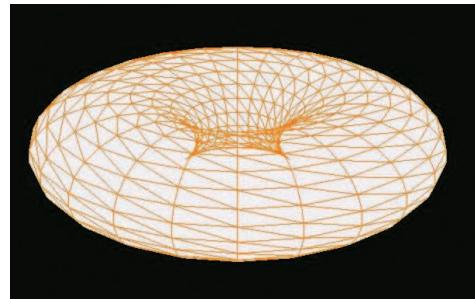


Figure 2.12
Torus with fully rasterized primitives (wireframe grid superimposed).

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 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 < 200) 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.

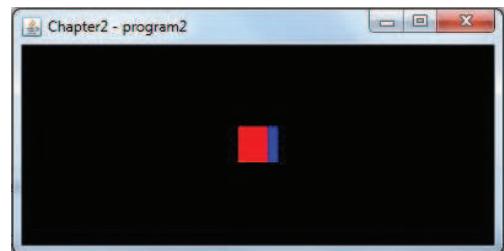


Figure 2.13
Fragment shader color variation.

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 co-ordinated use of two buffers: the *color buffer* 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:

```

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.

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 Java runtime*. Another complication is that GLSL code doesn't run on the CPU (it runs on the GPU), so *the operating 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 error string lookup tool `gluErrorString()` from the “GLU” library [GL16], as well as OpenGL functions `glGetShaderiv()` and `glGetProgramiv()`, which are used to provide information about compiled GLSL shaders and programs. Accompanying them is the `createShaderProgram()` function (on the right) from the previous Program 2.2, but with the error-detecting calls added. These error-detecting modules are also available in `graphicslib3D`.

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 both GLSL compilation errors and OpenGL runtime errors, so it is highly recommended to use it throughout a Java/JOGl 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 JOGL 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

```
.....
import com.jogamp.opengl.glu.GLU;
.....
private void printShaderLog(int shader)
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    int[ ] len = new int[1];
    int[ ] chWrittn = new int[1];
    byte[ ] log = null;

    // determine the length of the shader compilation log
    gl.glGetShaderiv(shader, GL_INFO_LOG_LENGTH, len, 0);
    if (len[0] > 0)
    {
        log = new byte[len[0]];
        gl.glGetShaderInfoLog(shader, len[0], chWrittn, 0, log, 0);
        System.out.println("Shader Info Log: ");
        for (int i = 0; i < log.length; i++)
        {
            System.out.print((char) log[i]);
        } } }

void printProgramLog(int prog)
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    int[ ] len = new int[1];
    int[ ] chWrittn = new int[1];
    byte[ ] log = null;

    // determine the length of the program linking log
    gl.glGetProgramiv(prog, GL_INFO_LOG_LENGTH, len, 0);
    if (len[0] > 0)
    {
        log = new byte[len[0]];
        gl.glGetProgramInfoLog(prog, len[0], chWrittn, 0, log, 0);
        System.out.println("Program Info Log: ");
        for (int i = 0; i < log.length; i++)
        {
            System.out.print((char) log[i]);
        } } }

boolean checkOpenGLError()
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    boolean foundError = false;
    GLU glu = new GLU();
    int glErr = gl.glGetError();
    while (glErr != GL_NO_ERROR)
    {
        System.out.println("glError: " + glu.gluErrorString(glErr));
        foundError = true;
```

```

        glErr = gl.glGetError();
    }
    return foundError;
}

```

Example of checking for OpenGL errors:

```

private int createShaderProgram()
{
    // arrays to collect GLSL compilation status values.
    // note: one-element arrays are used because the associated JOGL calls require arrays.
    int[ ] vertCompiled = new int[1];
    int[ ] fragCompiled = new int[1];
    int[ ] linked = new int[1];
    ...
    // catch errors while compiling shaders

    gl.glCompileShader(vShader);
    checkOpenGLError(); // can use returned boolean
    gl.glGetShaderiv(vShader, GL_COMPILE_STATUS, vertCompiled, 0);
    if (vertCompiled[0] == 1)
    { System.out.println("... vertex compilation success.");
    } else
    { System.out.println("... vertex compilation failed.");
        printShaderLog(vShader);
    }

    gl.glCompileShader();
    checkOpenGLError(); // can use returned boolean
    gl.glGetShaderiv(fShader, GL_COMPILE_STATUS, fragCompiled, 0);
    if (fragCompiled[0] == 1)
    { System.out.println("... fragment compilation success.");
    } else
    { System.out.println("... fragment compilation failed.");
        printShaderLog(fShader);
    }

    if ((vertCompiled[0] != 1) || (fragCompiled[0] != 1))
    { System.out.println("\nCompilation error; return-flags:");
        System.out.println(" vertCompiled = " + vertCompiled[0] + " ; fragCompiled = " + fragCompiled[0]);
    } else
    { System.out.println("Successful compilation");
    }
    ...
}

```

```

// catch errors while linking shaders
gl.glLinkProgram(vfprogram);
checkOpenGLError();
gl.glGetProgramiv(vfprogram, GL_LINK_STATUS, linked,0);
if (linked[0] == 1)
{   System.out.println("... linking succeeded.");
} else
{   System.out.println("... linking failed.");
    printProgramLog(vfprogram);
}
...
}
}

```

Another set of tools that can help in tracking down the source of OpenGL and GLSL errors is to use JOGL's *composable pipeline* mechanism. There is a rich set of capabilities available in the DebugGL and TraceGL JOGL classes, which provide debugging and tracing support, respectively. One way of utilizing these capabilities in simple cases is to add one or both of the following command line options:

```

-Djogl.debug.DebugGL
-Djogl.debug.TraceGL

```

For example, the application can be run with both capabilities enabled as follows:

```
java -Dsun.java2d.d3d=false -Djogl.debug.DebugGL -Djogl.debug.TraceGL Code
```

Enabling debugging causes glGetError() to be invoked at each OpenGL call. Although any error messages generated tend to not be as informative as is the case when retrieving the error codes as shown in Program 2.3, it can be a quick way of narrowing down the likely location where an error occurred.

Enabling tracing causes a line of output on the command window to be displayed for each OpenGL call executed—including those called directly by the application, and others invoked by JOGL. For example, a trace for Program 2.2 produces the following output, which reflects the order of calls in a typical run:

```

glFinish()
glCreateShader(<int> 0x8B31) = 1
glShaderSource(<int> 0x1, <int> 0x3, <[Ljava.lang.String;>, <java.nio.IntBuffer> null)
glCompileShader(<int> 0x1)
glCreateShader(<int> 0x8B30) = 2
glShaderSource(<int> 0x2, <int> 0x4, <[Ljava.lang.String;>, <java.nio.IntBuffer> null)
glCompileShader(<int> 0x2)

```

```

glCreateProgram() = 3
glAttachShader(<int> 0x3, <int> 0x1)
glAttachShader(<int> 0x3, <int> 0x2)
glLinkProgram(<int> 0x3)
glDeleteShader(<int> 0x1)
glDeleteShader(<int> 0x2)
glGenVertexArrays(<int> 0x1, <[l>, <int> 0x0)
glBindVertexArray(<int> 0x1)
glGetError() = 0
glViewport(<int> 0x0, <int> 0x0, <int> 0x180, <int> 0xA2)
glUseProgram(<int> 0x3)
glDrawArrays(<int> 0x0, <int> 0x0, <int> 0x1)

```

Although extremely useful during debugging, as with most debugging tools the composable pipeline incurs considerable overhead, and should not be enabled in production code.

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.

2.3 READING GLSL SOURCE CODE FROM FILES

So far, our GLSL shader code was 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 Java skill, and won't be covered here. However, for practicality, code to read shaders is provided in `readShaderSource()`, shown in Program 2.4 and also available in `graphicslib3D`. 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.

Program 2.4 Reading GLSL Source from Files

```
(....imports as before, plus the following...)
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class Code extends JFrame implements GLEventListener
{ (..... declarations same as before, display() as before)

    private int createShaderProgram()
    { (..... as before plus....)
        vshaderSource = readShaderSource("vert.shader");
        fshaderSource = readShaderSource("frag.shader");

        gl.glShaderSource(vertexShader, vshaderSource.length, vshaderSource, null, 0);
        gl.glShaderSource(fragmentShader, fshaderSource.length, fshaderSource, null, 0);

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

    (.... main, constructor, reshape, init, dispose as before )

    private String[ ] readShaderSource(String filename)
    { Vector<String> lines = new Vector<String>();
        Scanner sc;
        try { sc = new Scanner(new File(filename)); }
        catch (IOException e)
        { System.err.println("IOException reading file: " + e);
            return null;
        }
        while (sc.hasNext())
        { lines.addElement(sc.nextLine());
        }
        String[ ] program = new String[lines.size()];
        for (int i = 0; i < lines.size(); i++)
        { program[i] = (String) lines.elementAt(i) + "\n";
        }
        return program;
    }
}
```

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 Java/JOGL 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 in 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);
}
```

Java/JOGL application – in `display()`

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

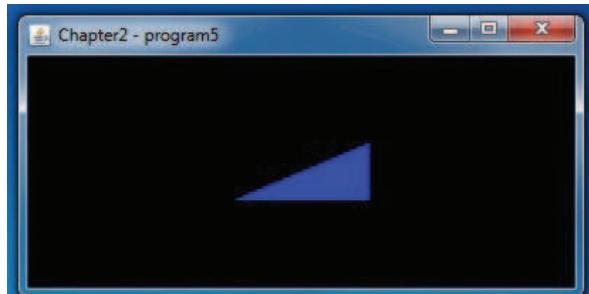


Figure 2.15
Drawing a simple triangle.

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 OpenGL makes a single call to `init()`, and then to `display()`, when it is initialized. After that, if there are changes to our scene, it becomes the programmer's responsibility to tell OpenGL to call `display()` again. This is done by invoking the `display()` function in the `GLCanvas` from the Java/JOGL application, as follows:

```
myCanvas.display();
```

That is, the `GLCanvas` has its own `display()` function. If we call it, the `GLCanvas` will then in turn call back the `display()` function in our Java/JOGL application. Technically, the two `display()`s are completely different functions, but they are intertwined.

One approach is to call `display()` whenever a change in the scene occurs. In a complex scene, this can become unwieldy. A better approach is to make this call repeatedly, at fixed intervals typically called the *frame rate*. Each rendering of our scene is then called a *frame*.

There are many ways to organize the code for animating a scene. One way is to create an “animator” class in Java, using either `Timer`, or `Thread.sleep()`, or better yet, a `ScheduledThreadPoolExecutor`.

An even simpler way to build the animator is to use one of the JOGL-specific animator classes:

- `Animator`
- `FPSAnimator`

These classes are designed specifically for animating 3D scenes. An `FPSAnimator` (“FPS” stands for “frames per second”), when instantiated, calls the `display()` function on a drawable object repeatedly, at a specified frame rate.

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.

The animator is instantiated by the Java/JOGL application in its constructor. After that, the application is free to make changes to the scene at any time and at any speed. The screen, however, will only be updated at intervals corresponding

to the specified frame rate. The first parameter on the `FPSAnimator` constructor call specifies the drawable object, and the second parameter specifies the frame rate (frames per second). In this example, there are 50 calls per second to the `GLCanvas's display()` function.

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 at 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 Java/JOGL application first calls `glGetUniformLocation()` to get a pointer to the "offset" variable, 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

Java/JOGL application:

```
// same imports and declarations as before, plus the following:
import com.jogamp.opengl.util.*;

...
public class Code extends JFrame implements GLEventListener
{ // same declarations as before, plus:
    private float x = 0.0f;           // location of triangle
    private float inc = 0.01f;         // offset for moving the triangle

    public Code()
    { // same constructor as before, plus this at the end, after the call to setVisible(true).
        // The second parameter specifies the frames per second
        FPSAnimator animtr = new FPSAnimator(myCanvas, 50);
        animtr.start();
    }

    public void display(GLAutoDrawable drawable)
    { GL4 gl = (GL4) GLContext.getCurrentGL();
        gl.glUseProgram(rendering_program);

        float bkg[] = { 0.0f, 0.0f, 0.0f, 1.0f }; // clear the background to black, each time
        FloatBuffer bkgBuffer = Buffers.newDirectFloatBuffer(bkg);
        gl.glClearBufferfv(GL_COLOR, 0, bkgBuffer);
```

```

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
int offset_loc = glGetUniformLocation(rendering_program, "offset"); // retrieve pointer to "offset"
gl.glProgramUniform1f(rendering_program, offset_loc, x); // send value in "x" to "offset"

gl.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);
}

```

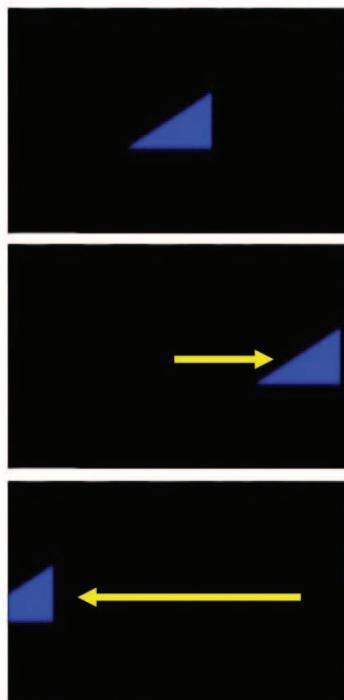


Figure 2.16
An animated, moving triangle.

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.

We ignored 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. JOGL ensures that strings sent to

`glShaderSource()` are null-terminated. However, it is not uncommon to encounter applications that build these arrays manually rather than sending null.

The composable pipeline can also be configured within the Java/JOGL application code [JU16], rather than just enabling it on the command line (as was described in Section 2.2). This can be useful for utilizing the debugging and tracing tools based on interactive input (e.g., a user keystroke).

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 for example 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”:

```
gl.glGetFloatv(GL_POINT_SIZE_RANGE, size, 0)
```

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

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.

References

- [GL16] GLUT and OpenGL Utility Libraries, accessed July 2016, <https://www.opengl.org/resources/libraries/>.
- [JU16] JOGL Users Guide, accessed July 2016, <https://jogamp.org/jogl/doc/userguide/>.
- [OP16] OpenGL 4.5 Reference Pages, accessed July 2016, <https://www.opengl.org/sdk/docs/man/>.
- [SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

MATHEMATICAL FOUNDATIONS

3.1	<i>3D Coordinate Systems</i>	36
3.2	<i>Points</i>	36
3.3	<i>Matrices</i>	37
3.4	<i>Transformation Matrices</i>	40
3.5	<i>Vectors</i>	44
3.6	<i>Local and World Space</i>	48
3.7	<i>Eye Space and the Synthetic Camera</i>	49
3.8	<i>Projection Matrices</i>	52
3.9	<i>Look-At Matrix</i>	56
3.10	<i>GLSL Functions for Building Matrix Transforms</i>	57
	<i>Supplemental Notes</i>	60



Computer graphics makes heavy use of mathematics, particularly matrices and matrix algebra. Although we tend to consider 3D graphics programming to be among the most contemporary of technical fields (and in many respects it is), many of the techniques that are used actually date back hundreds of years. Some of them were first understood and codified by the great philosophers of the Renaissance era.

Virtually every facet of 3D graphics, every effect—movement, scale, perspective, texturing, lighting, shadows, and so on—all will be accomplished largely mathematically. Therefore this chapter lays the groundwork upon which every subsequent chapter relies.

It is assumed the reader has a basic knowledge of matrix operations; a full coverage of basic matrix algebra is beyond the scope of this text. Therefore, if at any point a particular matrix operation is unfamiliar, it may be necessary to do some supplementary background reading to ensure full understanding before proceeding.

3.1 3D COORDINATE SYSTEMS

3D space is generally represented with three axes: X, Y, and Z. The three axes can be arranged into two configurations, *right-handed* or *left-handed*. (The name derives from the orientation of the axes as if constructed by pointing the thumb and first two fingers of the right versus the left hand, at right angles.)

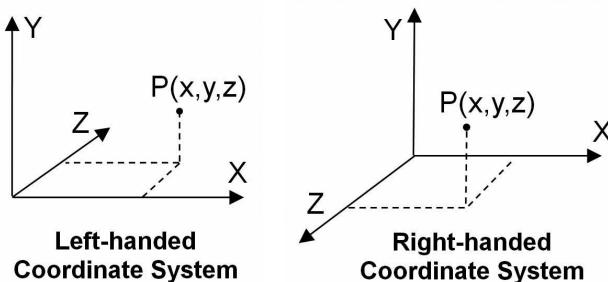


Figure 3.1
3D coordinate systems.

It is important to know which coordinate system your graphics programming environment uses. For example, the majority of coordinate systems in OpenGL are right-handed, whereas in Direct3D the majority are left-handed. Throughout this book, we will assume a right-handed configuration unless otherwise stated.

3.2 POINTS

Points in 3D space can be specified by listing the X, Y, Z values, using a notation such as (2, 8, -3). However, it turns out to be much more useful to specify points using *homogeneous* notation, a representation first described in the early 1800s. Points in homogeneous notation contain four values, the first three corresponding to X, Y, and Z, and the fourth, W, is always a fixed nonzero value, usually 1. Thus, we represent this point as [2, 8, -3, 1]. As we will see shortly, homogeneous notation will make many of our graphics computations more efficient.

The appropriate GLSL data type for storing points in homogeneous 3D notation is `vec4`. (“`vec`” refers to *vector*, but it can also be used for a point). The `graphicslib3D` library includes a *class* for creating and storing homogeneous points in the Java application, called `Point3D`.

3.3 MATRICES

A *matrix* is a rectangular array of values, and its elements are typically accessed by means of subscripts. The first subscript refers to the row number, and the second subscript refers to the column number, with the subscripts starting at 0. Most of the matrices that we will use for 3D graphics computations are of size 4x4, as shown in Figure 3.2:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Figure 3.2

4x4 matrix.

The GLSL language includes a data type called `mat4` that can be used for storing 4x4 matrices. Similarly, `graphicslib3D` includes a *class* called `Matrix3D` for instantiating and storing 4x4 matrices.

The *identity matrix* contains all zeros, with ones along the diagonal. Any item multiplied by the identity matrix is unchanged:

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

In `graphicslib3D` the identity matrix is available through the function `Matrix3D.setIdentity()`.

The *transpose* of a matrix is computed by interchanging its rows and columns. For example:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{10} & A_{20} & A_{30} \\ A_{01} & A_{11} & A_{21} & A_{31} \\ A_{02} & A_{12} & A_{22} & A_{32} \\ A_{03} & A_{13} & A_{23} & A_{33} \end{bmatrix}^T$$

The graphicslib3D library and GLSL both have transpose functions: Matrix3D.transpose() and transpose(mat4), respectively.

Matrix addition is straightforward:

$$\begin{bmatrix} A+a & B+b & C+c & D+d \\ E+e & F+f & G+g & H+h \\ I+i & J+j & K+k & L+l \\ M+m & N+n & O+o & P+p \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} + \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

In GLSL the + operator is overloaded on mat4 to support matrix addition.

There are various multiplication operations that can be done with matrices that are useful in 3D graphics. Matrix multiplication in general can be done either *left-to-right*, or *right-to-left* (note that since these operations are different, it follows that matrix multiplication is not commutative). Most of the time we will use right-to-left multiplication.

In 3D graphics, multiplying a point by a matrix is in most cases done right-to-left, as follows:

$$\begin{bmatrix} AX+BY+CZ+D \\ EX+FY+GZ+H \\ IX+JY+KZ+L \\ MX+NY+OZ+P \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Note that we represent the point (X,Y,Z) in homogeneous notation as a 1-column matrix.

GLSL and graphicslib3D support multiplying a point by a matrix: Point3D.mult(Matrix3D) in graphicslib3D, and in GLSL with the * operator.

Multiplying a 4x4 Matrix by another 4x4 matrix is done as follows:

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} * \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \quad \text{produces:}$$

$$\left[\begin{array}{cccc} Aa + Be + Ci + Dm & Ab + Bf + Cj + Dn & Ac + Bg + Ck + Do & Ad + Bh + Cl + Dp \\ Ea + Fe + Gi + Hm & Eb + Ff + Gj + Hn & Ec + Fg + Gk + Ho & Ed + Fh + Gl + Hp \\ Ia + Je + Ki + Lm & Ib + Jf + Kj + Ln & Ic + Jg + Kk + Lo & Id + Jh + Kl + Lp \\ Ma + Ne + Oi + Pm & Mb + Nf + Oj + Pn & Mc + Ng + Ok + Po & Md + Nh + Ol + Pp \end{array} \right]$$

Matrix multiplication is frequently referred to as *concatenation*, because as will be seen, it is used to combine a set of matrix transforms into a single matrix. This ability to combine matrix transforms is made possible because of the *associative* property of matrix multiplication. Consider the following sequence of operations:

$$\text{New Point} = \text{Matrix}_1 * (\text{Matrix}_2 * (\text{Matrix}_3 * \text{Point}))$$

Here, we multiply a point by Matrix_3 , then multiply that result by Matrix_2 , and that result finally by Matrix_1 . The result is a new point. The associative property ensures that the above computation is equivalent to:

$$\text{New Point} = (\text{Matrix}_1 * \text{Matrix}_2 * \text{Matrix}_3) * \text{Point}$$

Here, we first multiply the three matrices together, forming the *concatenation* of Matrix_1 , Matrix_2 , and Matrix_3 (which itself is also a 4x4 matrix). If we refer to this concatenation as Matrix_C , we can rewrite the above operation as:

$$\text{New Point} = \text{Matrix}_C * \text{Point}$$

The advantage here, as we will see in Chapter 4, is that we will frequently need to apply the same sequence of matrix transformations to every point in our scene. By pre-computing the concatenation of all of those matrices once, it turns out that we can reduce the total number of matrix operations needed manyfold.

GLSL and graphicslib3D both support matrix multiplication; in GLSL with the overloaded `*` operator, and in graphicslib3D with the function `Matrix3D.concatenate(Matrix3D)`.

The *inverse* of a 4x4 matrix M is another 4x4 matrix, denoted M^{-1} , that has the following property under matrix multiplication:

$$M * (M^{-1}) = (M^{-1}) * M = \text{identity matrix}$$

We won't present the details of computing the inverse here. However, it is worth knowing that determining the inverse of a matrix can be computationally expensive; fortunately, we will rarely need it. In the rare instances when we do, it

is available in `graphicslib3D` through the function `Matrix3D.inverse()`, and in GLSL through the `mat4.inverse()` function.

3.4 TRANSFORMATION MATRICES

In graphics, matrices are typically used for performing *transformations* on objects. For example, a matrix can be used to move a point from one location to another. In this chapter we will learn several useful transformation matrices:

- *Translation*
- *Rotation*
- *Scale*
- *Projection*
- *Look-At*

An important property of our transformation matrices is that they are all of size 4×4 . This is made possible by our decision to use the *homogeneous* notation. Otherwise, some of the transforms would be of diverse and incompatible dimensions. As we have seen, ensuring they are the same size is not just for convenience; it also makes it possible to combine them arbitrarily, and pre-compute groups of transforms for improved performance.

3.4.1 Translation

A *translation* matrix is used to move items from one location to another. It consists of an identity matrix, with the X, Y, and Z movement(s) given in locations A_{03} , A_{13} , A_{23} . Figure 3.3 shows the form of a translation matrix, and its effect when multiplied by a homogeneous point; the result is a new point “moved” by the translate values.

$$\begin{pmatrix} X + T_x \\ Y + T_y \\ Z + T_z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Figure 3.3
Translation matrix transform.

Note that point (X, Y, Z) is translated (or moved) to location $(X+T_x, Y+T_y, Z+T_z)$ as a result of being multiplied by the translation matrix. Also note that multiplication is specified *right-to-left*.

For example, if we wish to move a group of points upwards, 5 units along the positive Y direction, we could build a translation matrix by taking an identity matrix and placing the value 5 in the T_y position shown above. Then we simply multiply each of the points we wish to move by the matrix.

There are several functions in graphicslib3D for building translation matrices, and for multiplying points by matrices. Some relevant functions are:

- `Matrix3D.translate(x,y,z)`
- `Point3D.mult(Matrix3D)`

3.4.2 Scaling

A scale matrix is used to change the size of objects, or to move points towards or away from the origin. Although it may initially seem strange to scale a point, objects in OpenGL are defined by groups of points. So, scaling an object involves expanding or contracting its set of points.

The scale matrix transform consists of an identity matrix, with the X , Y , and Z scale factors given in locations A_{00} , A_{11} , A_{22} . Figure 3.4 shows the form of a scale matrix, and its effect when multiplied by a homogeneous point; the result is a new point modified by the scale values.

$$\begin{pmatrix} X * S_x \\ Y * S_y \\ Z * S_z \\ 1 \end{pmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Figure 3.4
Scale matrix transform.

There are several functions in graphicslib3D for building scale matrices, multiplying points by matrices, and scaling a point by a single scale factor. Some relevant functions are:

- *Matrix3D.scale(x,y,z)*
- *Point3D.mult(Matrix3D)*
- *Point3D.scale(f)*

Scaling can be used to switch coordinate systems. For example, we can use scale to determine what the left-hand coordinates would be, given a set of right-hand coordinates. From Figure 3.1 we see that negating the Z coordinate would toggle between right-hand and left-hand systems, so the scale matrix transform to accomplish this is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4.3 | Rotation

Rotation is a bit more complex, because rotating an item in 3D space requires specifying (a) an axis of rotation, and (b) a rotation amount in degrees or radians.

In the mid-1700s, the mathematician Leonhard Euler showed that a rotation around any desired axis could be specified instead as a combination of rotations around the X, Y, and Z axes [EU76]. These three rotation angles, around the respective axes, have come to be known as *Euler angles*. The discovery, known as *Euler's Theorem*, is very useful to us, because rotations around each of the three axes can be specified using matrix transforms.

The three rotation transforms, around the X, Y, and Z axes respectively, are shown in Figure 3.5. There are several functions in `graphicslib3D` for building and using rotation matrices as well:

- *Matrix3D.rotateX(degrees)*
- *Matrix3D.rotateY(degrees)*
- *Matrix3D.rotateZ(degrees)*
- *Matrix3D.rotate($\theta_x, \theta_y, \theta_z$)*
- *Point3D.mult(Matrix3D)*

Rotation around X by θ :

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Rotation around Y by θ :

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Rotation around Z by θ :

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Figure 3.5

Rotation transform matrices.

In practice, using Euler angles to rotate an item around an arbitrary line in 3D space takes a couple of additional steps if the line doesn't pass through the origin. In general:

1. Translate the axis of rotation so that it goes through the origin.
2. Rotate by appropriate Euler angles around X, Y, and Z.
3. Undo the translation of Step 1.

The three rotation transforms shown in Figure 3.5 each have the interesting property that the inverse rotation happens to equal the transpose of the matrix. This can be verified by examining the above matrices, recalling that $\cos(-\theta) = \cos(\theta)$, and $\sin(-\theta) = -\sin(\theta)$. This property will become useful later.

Euler angles can cause certain artifacts in some 3D graphic applications. For that reason it is often advisable to use *quaternions* for computing rotations. Many resources exist for those readers interested in exploring quaternions (e.g., [KU98]). Euler angles will suffice for most of our needs.

3.5 VECTORS

Vectors specify a *magnitude* and *direction*. They are not bound to a specific location; a vector can be “moved” without changing what it represents.

There are various ways to notate a vector, such as a line segment with an arrowhead at one end, or as a pair (magnitude, direction), or as the difference between two points. In 3D graphics, vectors are frequently represented as a single point in space, where the vector is the distance and direction from the origin to that point. In Figure 3.6, vector V (shown in red) can be specified either as the difference between points P_1 and P_2 , or as an equivalent distance from the origin to P_3 . In all of our applications, we specify V as simply (x,y,z) , the same notation used to specify the point P_3 .

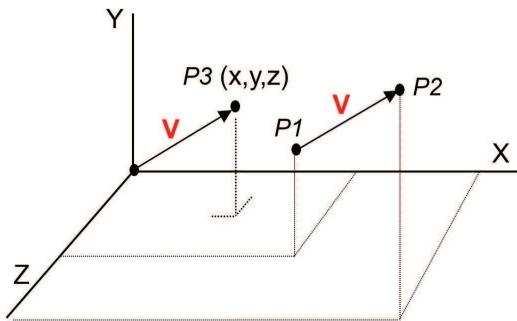


Figure 3.6
Two representations for a vector V .

It is convenient to represent a vector the same way as a point, because we can use our matrix transforms on points or vectors interchangeably. However, it also can be confusing. For this reason we sometimes will notate a vector with a small arrow above it (such as \vec{V}). Some graphics systems do not distinguish between a point and a vector at all, such as in GLSL, which provides datatypes `vec3` and `vec4` that can hold either points or vectors. On the other hand, `graphicslib3D` has separate `Point3D` and `Vector3D` classes, and it enforces appropriate use of one or the

other depending on the operation being done. It is an open debate as to whether it is simpler to use one datatype for both, or separate datatypes.

There are several vector operations that are used frequently in 3D graphics, for which there are functions available in `graphicslib3D` and GLSL. For example, assuming vectors $A(u,v,w)$ and $B(x,y,z)$:

Addition and Subtraction:

$$A \pm B = (u \pm x, v \pm y, w \pm z)$$

graphicslib3D: `Vector3D.add(Vector3D)`
 GLSL: `vec3 ± vec3`

Normalize (change to length=1):

$$\hat{A} = A/|A| = A/\sqrt{u^2+v^2+w^2}, \text{ where } |A| \equiv \text{length of vector } A$$

graphicslib3D: `Vector3D.normalize()`
 GLSL: `normalize(vec3) or normalize(vec4)`

Dot Product:

$$A \bullet B = ux + vy + wz$$

graphicslib3D: `Vector3D.dot(Vector3D)`
 GLSL: `dot(vec3,vec3) or dot(vec4,vec4)`

Cross Product:

$$A \times B = (vz-wy, wx-uz, uy-vx)$$

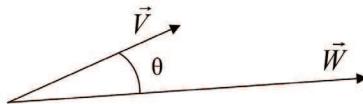
graphicslib3D: `Vector3D.cross(Vector3D)`
 GLSL: `cross(vec3,vec3)`

Other useful vector functions are *magnitude* (which is available in both `graphicslib3D` and GLSL), and *reflection* and *refraction* (both of which are available in GLSL).

We shall now take a closer look at the functions *dot product* and *cross product*.

3.5.1 Uses for Dot Product

Throughout this book, our programs make heavy use of the dot product. The most important and fundamental use is for finding the *angle between two vectors*. Consider two vectors \vec{v} and \vec{w} , and say we wish to find the angle θ separating them.



$$\vec{V} \bullet \vec{W} = |\vec{V}| * |\vec{W}| * \cos(\theta)$$

$$\cos(\theta) = \frac{\vec{V} \bullet \vec{W}}{|\vec{V}| * |\vec{W}|}$$

Therefore, if \vec{V} and \vec{W} are normalized (i.e., of unit length—recall the “^” notation for normalization), then:

$$\cos(\theta) = \hat{V} \bullet \hat{W}$$

$$\theta = \arccos(\hat{V} \bullet \hat{W})$$

Interestingly, we will later see that often it is $\cos(\theta)$ that we need, rather than θ itself. So, both of the above derivations will be directly useful.

The dot product also has a variety of other uses:

- finding a vector’s magnitude: $\sqrt{\vec{V} \bullet \vec{V}}$
- two vectors are perpendicular, if $\vec{V} \bullet \vec{W} = 0$
- two vectors are parallel, if: $\vec{V} \bullet \vec{W} = |\vec{V}| * |\vec{W}|$
- parallel but pointing in opposite directions: $\vec{V} \bullet \vec{W} = -|\vec{V}| * |\vec{W}|$
- angle between vectors lies in the range $[-90^\circ \dots +90^\circ]$: $\hat{V} \bullet \hat{W} > 0$
- minimum signed distance from point $P=(x,y,z)$ to plane $S=(a,b,c,d)$. First, find unit vector normal to S :

$$\hat{n} = \left(\frac{a}{\sqrt{a^2 + b^2 + c^2}}, \frac{b}{\sqrt{a^2 + b^2 + c^2}}, \frac{c}{\sqrt{a^2 + b^2 + c^2}} \right), \text{ and shortest}$$

$$\text{distance } D = \frac{d}{\sqrt{a^2 + b^2 + c^2}} \text{ from the origin to the plane.}$$

Then, the minimum signed distance from P to S is: $(\hat{n} \bullet P) + D$ and the sign of this distance determines on which side of the plane S point P lies.

3.5.2 Uses for Cross Product

An important property of the cross product of two vectors, which we will make heavy use of throughout this book, is that it produces a vector that is *normal* (perpendicular) to the plane defined by the original two vectors.

Any two non-collinear vectors define a plane. For example, consider two arbitrary vectors \vec{v} and \vec{w} . Since vectors can be moved without changing their meaning, they can be moved so that their origins coincide. Figure 3.7 shows a plane defined by \vec{v} and \vec{w} , and the normal vector resulting from their cross product. The direction of the resulting normal obeys the *right-hand rule*, wherein curling the fingers of one's right hand from \vec{v} to \vec{w} causes the thumb to point in the direction of the normal vector \vec{r} .

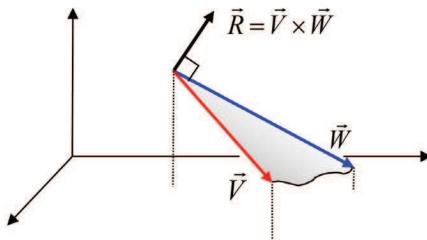


Figure 3.7
Cross product produces normal vector.

Note that the order is significant; $\vec{w} \times \vec{v}$ would produce a vector in the opposite direction from \vec{r} .

The ability to find normal vectors by using the cross product will become extremely useful later when we study *lighting*. In order to determine lighting effects, we will need to know *outward normals* associated with the model we are rendering. Figure 3.8 shows an example of a simple model made up of six points (vertices), and the computation employing cross product that determines the outward normal of one of its faces.

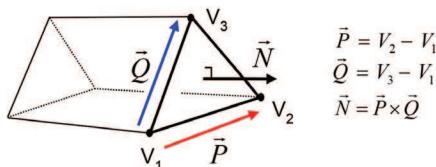


Figure 3.8
Computing outward normals.

3.6 LOCAL AND WORLD SPACE

The most common use for 3D graphics (with OpenGL or any other framework) is to simulate a three-dimensional world, place objects in it, and then view that simulated world on a monitor. The objects placed in the 3D world are usually modeled as collections of triangles. Later, in Chapter 6, we will dive into the modeling process. But we can start looking at the overall process now.

When building a 3D model of an object, we generally orient the model in the most convenient manner for describing it. For example, when modeling a sphere, we might orient the model with the sphere's center at the origin (0,0,0), and give it a convenient radius, such as 1. The space in which a model is defined is called its *local space*, or *model space*. OpenGL documentation uses the term *object space*.

The sphere might then be used as a piece of a larger model, such as becoming the head on a robot. The robot would, of course, be defined in its own local/model space. Positioning the sphere model into the robot model space can be done using the matrix transforms for scale, rotation, and translation, as illustrated in Figure 3.9. In this manner, complex models can be built hierarchically (this is developed further in Section 4.8 of Chapter 4, using a stack of matrices).

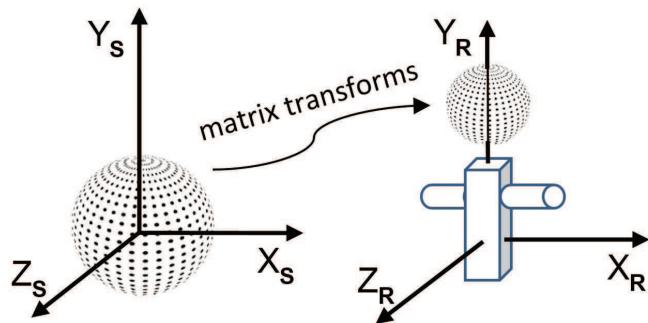


Figure 3.9
Model spaces for a sphere and a robot.

In the same manner, modeled objects are placed in a simulated world by deciding on the orientation and dimensions of that world, called *world space*. The matrix that positions and orients an object into world space is called a *model matrix*, or M.

3.7 EYE SPACE AND THE SYNTHETIC CAMERA

So far, the transform matrices we have seen all operate in 3D space. Ultimately, however, we will want to display our 3D space—or a portion of it—on a 2D monitor. In order to do this, we need to decide on a vantage point. Just as we see our real world through our eyes from a particular point, in a particular direction, so too must we establish a position and orientation as the window into our virtual world. This vantage point is called “view” or “eye” space, or the “synthetic camera.”

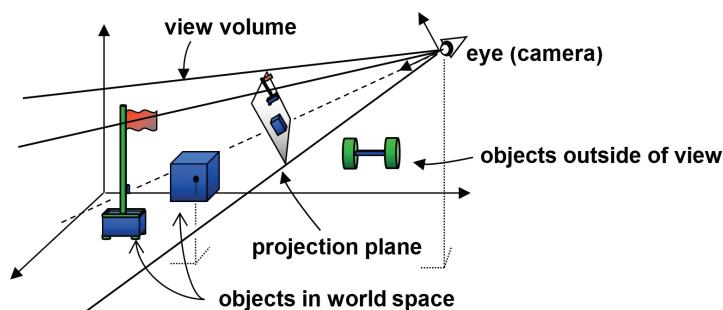


Figure 3.10
Positioning a camera in the 3D world.

As shown in Figure 3.10 and Figure 3.12, viewing involves: (a) placing the camera at some world location; (b) orienting the camera, which usually requires maintaining its own set of orthogonal axes $U/V/N$; (c) defining a *view volume*; and (d) projecting objects within the volume onto a *projection plane*.

OpenGL includes a camera that is permanently fixed at the origin $(0,0,0)$, and faces down the negative Z-axis, as shown in Figure 3.11.

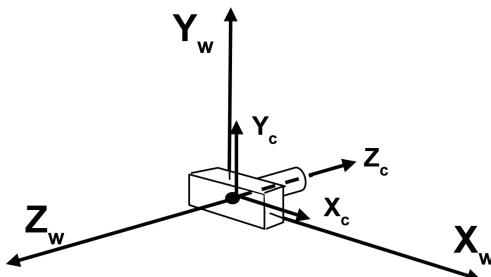


Figure 3.11
OpenGL fixed camera.

In order to use the OpenGL camera, one of the things we need to do is *simulate* moving it to some desired location and orientation. This is done by figuring out where our objects in the world are located relative to the desired camera position (i.e., where they are located in “camera space,” as defined by the U, V, and N axes of the camera as illustrated in Figure 3.12). Given a point at world space location P_w , we need a transform to convert it to the equivalent point in camera space, making it *appear* as though we are viewing it from the desired camera location C_w . We do this by computing its camera space position P_c . Knowing that the OpenGL camera location is always at the fixed position (0,0,0), what transform would achieve this?

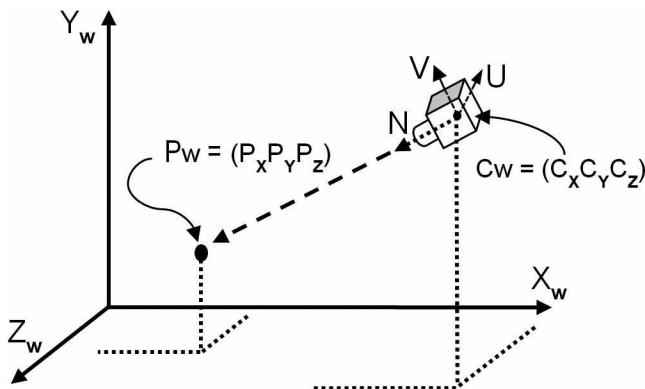


Figure 3.12
Camera orientation.

The necessary transforms are:

1. Translate P_w by the negative of the desired camera location.
2. Rotate P_w by the negative of the desired camera orientation Euler angles.

We can build a single transform that does both the rotation and the translation in one matrix, called the *viewing transform* matrix, or V . The matrix V is produced by concatenating the two matrices T (a translation matrix containing the negative of the desired camera location), and R (a rotation matrix containing the negative of the desired camera orientation). In this case, working from right to left, we first translate world point P , then rotate it:

$$P_c = R * (T * P_w)$$

As we saw earlier, the associative rule allows us to group the operations instead thusly:

$$P_c = (R * T) * P_w$$

If we save the concatenation $R*T$ in the matrix V , the operation now looks like:

$$P_c = V * P_w$$

The complete computation, and the exact contents of matrices T and R , is shown in Figure 3.13 (we omit the derivation of matrix R —a derivation is available in [FV95]).

$$\begin{matrix} & \text{negative of camera} & \text{negative of} \\ & \text{rotation angles} & \text{camera location} \\ \left(\begin{array}{c} X_c \\ Y_c \\ Z_c \\ 1 \end{array} \right) & = & \left[\begin{array}{cccc} \hat{U}_x & \hat{U}_y & \hat{U}_z & 0 \\ \hat{V}_x & \hat{V}_y & \hat{V}_z & 0 \\ \hat{N}_x & \hat{N}_y & \hat{N}_z & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] * \left[\begin{array}{cccc} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{array} \right] * \left(\begin{array}{c} P_x \\ P_y \\ P_z \\ 1 \end{array} \right) \\ & \underbrace{\qquad\qquad\qquad}_{\substack{R \text{ (rotation)} \\ V \text{ (viewing transform)}}} & \underbrace{\qquad\qquad\qquad}_{T \text{ (translation)}} \\ \text{point } P_c \text{ in} & & \text{world} \\ \text{eye space} & & \text{point } P_w \end{matrix}$$

Figure 3.13
Deriving a View matrix.

More commonly, the V matrix is concatenated with the model matrix M to form a single *model-view* (MV) matrix:

$$MV = V * M$$

Then, a point P_M in its own model space is transformed directly to camera space in one step as follows:

$$P_c = MV * P_M$$

The advantage of this approach becomes clear when one considers that, in a complex scene, we will need to perform this transformation not on just one point, but *on every vertex in the scene*. By pre-computing MV , transforming each

point into view space will require us to do just one matrix multiplication per vertex, rather than two. Later, we will see that we can extend this process to pre-computing several matrix concatenations, reducing the per-vertex computations considerably.

3.8 PROJECTION MATRICES

Now that we have established the camera, we can examine *projection matrices*. Two important projection matrices that we will now examine are (a) *perspective* and (b) *orthographic*.

3.8.1 The Perspective Projection Matrix

Perspective projection attempts to make a 2D picture appear 3D, by utilizing the concept of perspective to mimic what we see when we look at the real world.

Objects that are close appear larger than objects that are far away, and in some cases, lines that are parallel in 3D space are no longer parallel when drawn with perspective.

Perspective was one of the great discoveries of the Renaissance era in the 1400–1500s, when artists started painting with more realism than did their predecessors.

An excellent example can be seen in Figure 3.14, the “Annunciation with St. Emidius,” by Carlo Crivelli, painted in 1486 (currently held at the National Gallery in London [CR86]). The intense use of perspective is clear—the receding lines of the left-facing wall of the building on the right are slanted towards each other dramatically. This creates the illusion of depth and 3D space, and in the process lines



Figure 3.14

Annunciation, with Saint Emidius (Crivelli – 1486).

that are parallel in reality are not parallel in the picture. Also, the people in the foreground are larger than the people in the background. While today we take these devices for granted, finding a transformation matrix to accomplish this requires some mathematical analysis.

We achieve this effect by using a matrix transform that converts parallel lines into appropriate non-parallel lines. Such a matrix is called a *perspective matrix* or *perspective transform*, and is built by defining the four parameters of a *view volume*. Those parameters are (a) *aspect ratio*, (b) *field of view*, (c) *projection plane* or *near clipping plane*, and (d) *far clipping plane*.

Only objects between the *near* and *far* clipping planes are rendered. The *near* clipping plane also serves as the plane on which objects are *projected*, and is generally positioned close to the eye or camera (shown on the left in Figure 3.15). Selection of an appropriate value for the *far* clipping plane is discussed in Chapter 4. The *field of view* is the vertical angle of viewable space. The *aspect ratio* is the ratio width/height of the *near* and *far* clipping planes. The shape formed by these elements and shown in Figure 3.15 is called a *frustum*.

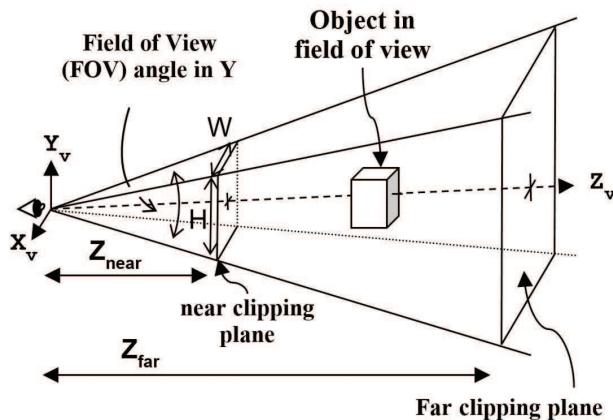


Figure 3.15
Perspective view volume or frustum.

The perspective matrix is used to transform points in 3D space to their appropriate position on the *near* clipping plane, and is built by first computing values q , A , B , and C , and then using those values to construct the matrix, as shown in Figure 3.16 (and derived in [FV95]).

$$q = \frac{1}{\tan(\frac{\text{fieldOfView}}{2})}$$

$$A = \frac{q}{\text{aspectRatio}}$$

$$B = \frac{Z_{\text{near}} + Z_{\text{far}}}{Z_{\text{near}} - Z_{\text{far}}}$$

$$C = \frac{2 * (Z_{\text{near}} * Z_{\text{far}})}{Z_{\text{near}} - Z_{\text{far}}}$$

$$\begin{bmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Figure 3.16
Building a Perspective matrix.

In Chapter 4 we will implement a function that generates the perspective transform matrix.

3.8.2 The Orthographic Projection Matrix

In *orthographic* projection, parallel lines remain parallel; that is, perspective isn't employed. Instead, objects that are within the view volume are projected directly, without any adjustment of their sizes due to their distances from the camera.

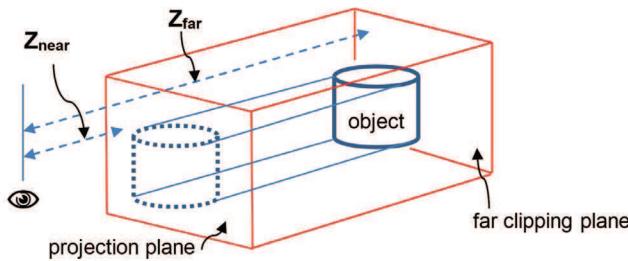


Figure 3.17
Orthographic projection.

An orthographic projection is a parallel projection in which all projections are at right angles with the projection plane. An orthographic matrix is built by defining the following parameters: (a) the distance Z_{near} from the camera to the projection plane, (b) the distance Z_{far} from the camera to the far clipping plane, and (c) values for L , R , T , and B , with L and R corresponding to the X coordinates of the left and right boundaries of the projection plane, respectively, and T and B corresponding to the Y coordinates of the top and bottom boundaries of the projection plane, respectively. The orthographic projection matrix, as derived in [FV95] is then:

$$\begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{1}{Z_{\text{far}}-Z_{\text{near}}} & -\frac{Z_{\text{near}}}{Z_{\text{far}}-Z_{\text{near}}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.18
Orthographic projection matrix.

Not all parallel projections are orthographic, but others are out of the scope of this textbook.

Parallel projections don't match what the eye sees when looking at the real world. But they are useful in a variety of situations, such as in casting shadows, performing 3D clipping, and in CAD (computer aided design)—the latter because they preserve measurement regardless of the placement of the objects. Regardless, the great majority of examples in this book use perspective projection.

3.9 LOOK-AT MATRIX

The final transformation we will examine is the *look-at* matrix. This is handy when you wish to place the camera at one location, and look towards a particular other location, as illustrated in Figure 3.19. Of course, it would be possible to achieve this using the methods we have already seen, but it is such a common operation that building one matrix transform to do it is often useful.

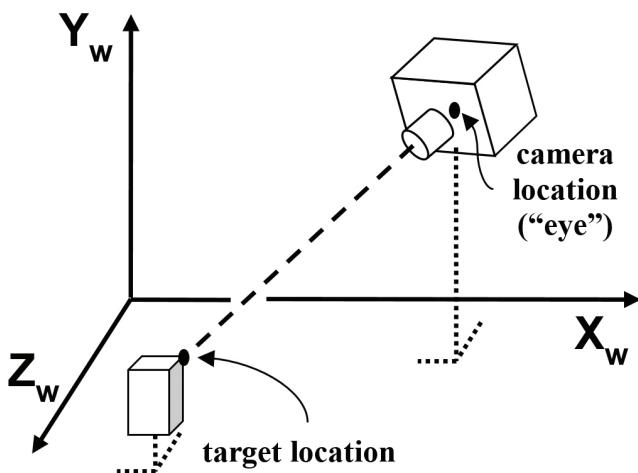


Figure 3.19
Elements of Look-At.

A look-at transform still requires deciding on a camera orientation. We do this by specifying a vector approximating the general orientation desired (such as the world Y axis). Typically, a sequence of cross products can be used to then generate a suitable set of forward, side, and up vectors for the desired camera orientation. Figure 3.20 shows the computations, starting with the camera location (eye), target location, and initial up vector \bar{Y} , to build the look-at matrix, as derived in [FV95].

$$\begin{aligned}\overrightarrow{fwd} &= \text{normalize}(\overrightarrow{\text{eye}} - \overrightarrow{\text{target}}) \\ \overrightarrow{\text{side}} &= \text{normalize}(-\overrightarrow{fwd} \times \overrightarrow{Y}) \\ \overrightarrow{\text{up}} &= \text{normalize}(\overrightarrow{\text{side}} \times -\overrightarrow{fwd})\end{aligned}$$

The look-at matrix then equals:

$$\left[\begin{array}{cccc} \overrightarrow{\text{side}}_x & \overrightarrow{\text{side}}_y & \overrightarrow{\text{side}}_z & -(\overrightarrow{\text{side}} \bullet \overrightarrow{\text{eye}}) \\ \overrightarrow{\text{up}}_x & \overrightarrow{\text{up}}_y & \overrightarrow{\text{up}}_z & -(\overrightarrow{\text{up}} \bullet \overrightarrow{\text{eye}}) \\ -\overrightarrow{fwd}_x & -\overrightarrow{fwd}_y & -\overrightarrow{fwd}_z & -(-\overrightarrow{fwd} \bullet \overrightarrow{\text{eye}}) \\ 0 & 0 & 0 & 1 \end{array} \right]$$

Figure 3.20
Look-At matrix.

We can encode this as a simple Java/JOGL utility function that builds a look-at matrix, given specified values for camera location, target location, and the initial “up” vector \overrightarrow{Y} . This function, shown in Figure 3.21, will be useful later in this textbook, particularly in Chapter 8 when we generate shadows.

3.10 GLSL FUNCTIONS FOR BUILDING MATRIX TRANSFORMS

Although graphicslib3D includes predefined functions for performing many of the 3D transformations covered in this chapter, such as translation, rotation, and scale, GLSL only includes basic matrix operations such as addition, concatenation, and so on. It is therefore sometimes necessary to write our own GLSL utility functions for building 3D transformation matrices when we need them to perform certain 3D computations in a shader. The appropriate datatype to hold such a matrix in GLSL is `mat4`.

The syntax for initializing `mat4` matrices in GLSL loads values by columns. The first four values are put into the first column, the next four into the next column, and so forth, as illustrated in the following example:

```

private Matrix3D lookAt(Point3D eye, Point3D target, Vector3D y)
{
    Vector3D eyeV = new Vector3D(eye);
    Vector3D targetV = new Vector3D(target);
    Vector3D fwd = (targetV.minus(eyeV)).normalize();
    Vector3D side = (fwd.cross(y)).normalize();
    Vector3D up = (side.cross(fwd)).normalize();

    Matrix3D look = new Matrix3D();
    look.setElementAt(0,0, side.getX());
    look.setElementAt(1,0, up.getX());
    look.setElementAt(2,0, -fwd.getX());
    look.setElementAt(3,0, 0.0f);
    look.setElementAt(0,1, side.getY());
    look.setElementAt(1,1, up.getY());
    look.setElementAt(2,1, -fwd.getY());
    look.setElementAt(3,1, 0.0f);
    look.setElementAt(0,2, side.getZ());
    look.setElementAt(1,2, up.getZ());
    look.setElementAt(2,2, -fwd.getZ());
    look.setElementAt(3,2, 0.0f);
    look.setElementAt(0,3, side.dot(eyeV.mult(-1)));
    look.setElementAt(1,3, up.dot(eyeV.mult(-1)));
    look.setElementAt(2,3, (fwd.mult(-1)).dot(eyeV.mult(-1)));
    look.setElementAt(3,3, 1.0f);
    return(look)
}

```

Figure 3.21

Java/Jogl function for generating a Look-At matrix.

```

mat4 translationMatrix =
mat4( 1.0, 0.0, 0.0, 0.0,           // note this is the leftmost column, not the top row
      0.0, 1.0, 0.0, 0.0,
      0.0, 0.0, 1.0, 0.0,
      tx, ty, tz, 1.0 );

```

which builds the translation matrix described previously in Figure 3.3.

Program 3.1 includes five GLSL functions for building 4x4 translation, rotation, and scale matrices, each corresponding to formulas given earlier in this chapter. We will use some of these functions later in the book.

Program 3.1 Building Transformation Matrices in GLSL

```

// builds and returns a translation matrix
mat4 buildTranslate(float x, float y, float z)
{
    mat4 trans = mat4( 1.0, 0.0, 0.0, 0.0,
                      0.0, 1.0, 0.0, 0.0,
                      0.0, 0.0, 1.0, 0.0,
                      x, y, z, 1.0 );
    return trans;
}

// builds and returns a matrix that performs a rotation around the X axis
mat4 buildRotateX(float rad)
{
    mat4 xrot = mat4( 1.0, 0.0, 0.0, 0.0,
                      0.0, cos(rad), -sin(rad), 0.0,
                      0.0, sin(rad), cos(rad), 0.0,
                      0.0, 0.0, 0.0, 1.0 );
    return xrot;
}

// builds and returns a matrix that performs a rotation around the Y axis
mat4 buildRotateY(float rad)
{
    mat4 yrot = mat4( cos(rad), 0.0, sin(rad), 0.0,
                      0.0, 1.0, 0.0, 0.0,
                      -sin(rad), 0.0, cos(rad), 0.0,
                      0.0, 0.0, 0.0, 1.0 );
    return yrot;
}

// builds and returns a matrix that performs a rotation around the Z axis
mat4 buildRotateZ(float rad)
{
    mat4 zrot = mat4( cos(rad), -sin(rad), 0.0, 0.0,
                      sin(rad), cos(rad), 0.0, 0.0,
                      0.0, 0.0, 1.0, 0.0,
                      0.0, 0.0, 0.0, 1.0 );
    return zrot;
}

// builds and returns a scale matrix
mat4 buildScale(float x, float y, float z)
{
    mat4 scale = mat4( x, 0.0, 0.0, 0.0,
                      0.0, y, 0.0, 0.0,
                      0.0, 0.0, z, 0.0,
                      0.0, 0.0, 0.0, 1.0 );
    return scale;
}

```

SUPPLEMENTAL NOTES

In this chapter we have seen examples of applying matrix transformations to points. Later, we will also want to apply these same transforms to vectors. In order to accomplish a transform on a vector V equivalent to applying some matrix transform M to a point, it is necessary in the general case to compute the *inverse transpose* of M , denoted $(M^{-1})^T$, and multiply V by that matrix. In some cases, $M = (M^{-1})^T$, and in those cases it is possible to simply use M . For example, the basic rotation matrices we have seen in this chapter are equal to their own inverse transpose, and can be applied directly to vectors as well as points. Thus, the examples in this book sometimes use $(M^{-1})^T$ when applying a transform to a vector, and sometimes simply use M .

One of the things we haven't discussed in this chapter is techniques for moving the camera smoothly through space. This is very useful, especially for games and CGI movies, but also for visualization, virtual reality, and for 3D modeling.

We didn't include complete derivations for all of the matrix transforms that were presented (they can be found in other sources, such as [FV95]). We strove instead for a concise summary of the point, vector, and matrix operations necessary for doing basic 3D graphics programming. As this book proceeds, we will encounter many practical uses for the methods presented.

Exercises

- 3.1** Modify Program 2.5 so that the vertex shader includes one of the `buildRotate()` functions from Program 3.1, and applies it to the points comprising the triangle. This should cause the triangle to be rotated from its original orientation. You don't need to animate the rotation.
- 3.2** (*RESEARCH*) At the end of Section 3.4 we indicated that Euler angles can in some cases lead to undesirable artifacts. The most common is called "gimbal lock." Describe gimbal lock, give an example, and explain why gimbal lock can cause problems.
- 3.3** (*RESEARCH*) One way of avoiding the artifacts that can manifest when using Euler angles is to use *quaternions*. We didn't study quaternions; however,

graphicslib3D includes a Quaternion class. Do some independent study on quaternions, and familiarize yourself with the related graphicslib3D Quaternion class.

References

- [CR86] C. Crivelli, *The Annunciation, with Saint Emidius* (1486) - in the National Gallery, London, England, accessed July 2016, <https://www.nationalgallery.org.uk/paintings/carlo-crivelli-the-annunciation-with-saint-emidius>.
- [EU76] L. Euler, *Formulae generales pro translatione quacunque coporum rigidorum* (General formulas for the translation of arbitrary rigid bodies), Novi Commentarii academiae scientiarum Petropolitanae 20, 1776.
- [FV95] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics - Principles and Practice*, 2nd ed. (Addison-Wesley, 1995).
- [KU98] J. B. Kuipers, *Quaternions and Rotation Sequences* (Princeton University Press, 1998).

MANAGING 3D GRAPHICS DATA

4.1	<i>Buffers & Vertex Attributes</i>	64
4.2	<i>Uniform Variables</i>	67
4.3	<i>Interpolation of Vertex Attributes</i>	68
4.4	<i>Model-View and Perspective Matrices</i>	69
4.5	<i>Our First 3D Program – a 3D Cube.</i>	71
4.6	<i>Rendering Multiple Copies of an Object</i>	80
4.7	<i>Rendering Multiple Different Models in a Scene.</i>	85
4.8	<i>Matrix Stacks</i>	87
4.9	<i>Combating “Z-Fighting” Artifacts</i>	94
4.10	<i>Other Options for Primitives</i>	95
4.11	<i>Back-Face Culling</i>	97
	<i>Supplemental Notes</i>	99



Using OpenGL to render 3D images generally involves sending several datasets through the OpenGL shader pipeline. For example, to draw a simple 3D object such as a *cube*, you will need to at least send the following items:

- the vertices for the cube model
- some transformation matrices to control the appearance of the cube’s orientation in 3D space

To complicate matters a bit, there are *two* ways of sending data through the OpenGL pipeline:

- through a *buffer* to a *vertex attribute*, or
- directly to a *uniform variable*

It is important to understand exactly how these two mechanisms work, so as to use the appropriate method for each item we are sending through.

Let's start by rendering a simple cube.

4.1 BUFFERS & VERTEX ATTRIBUTES

For an object to be drawn, its vertices must be sent to the vertex shader. Vertices are usually sent by putting them in a *buffer* on the Java side, and associating that buffer with a *vertex attribute* declared in the shader. There are several steps to accomplish this, some of which only need to be done once, and some of which—if the scene is animated—must be done at every frame:

Done once – typically in `init()`:

1. create a buffer
2. copy the vertices into the buffer

Done at each frame – typically in `display()`:

1. enable the buffer containing the vertices
2. associate the buffer with a vertex attribute
3. enable the vertex attribute
4. use `glDrawArrays(...)` to draw the object

Buffers are typically created all at once at the start of the program, either in `init()` or in a function called by `init()`. In OpenGL, a buffer is contained in a *Vertex Buffer Object*, or *VBO*, which is declared and instantiated in the Java/JOGL application. A scene may require many VBOs, so it is customary to generate and then fill several of them in `init()`, so that they are available whenever your program needs to draw one or more of them.

A buffer interacts with a vertex attribute in a specific way. When `glDrawArrays()` is executed, the data in the buffer starts flowing, sequentially from the beginning of the buffer, through the vertex shader. As described in Chapter 2, the vertex shader executes *once per vertex*. A vertex in 3D space requires three values, so an appropriate vertex attribute in the shader to receive these three values would be of type `vec3`. Then, for each *three values* in the buffer, the shader is invoked, as illustrated in Figure 4.1:

A related structure in OpenGL is called a *Vertex Array Object*, or *VAO*. VAOs were introduced in version 3.0 of OpenGL, and are provided as a way of organizing buffers and making them easier to manipulate in complex scenes. OpenGL requires at least one VAO be created, and for our purposes one will be sufficient.

For example, suppose that we wish to display two objects. On the Java/JOGL side, we could do this by declaring a single VAO and an associated set of two VBOs (one per object), as follows:

```
private int vao[ ] = new int[1];
private int vbo[ ] = new int[2];
...
gl glGenVertexArrays(1, vao, 0);
gl glBindVertexArray(vao[0]);
gl glGenBuffers(2, vbo, 0);  

// OpenGL requires these values be specified in arrays
```

The two OpenGL commands `glGenVertexArrays()` and `glGenBuffers()` create VAOs and VBOs, respectively, and return integer IDs for them. We store those IDs into the int arrays `vao` and `vbo`. The three parameters on each of them refer to how many are created, an array to hold the returned IDs, and an offset into that array (usually set to 0). The purpose of `glBindVertexArray()` is to make the specified VAO “active” so that the generated buffers¹ will be associated with that VAO.

A buffer needs to have a corresponding *vertex attribute* variable declared in the vertex shader. Vertex attributes are generally the first variables declared in a

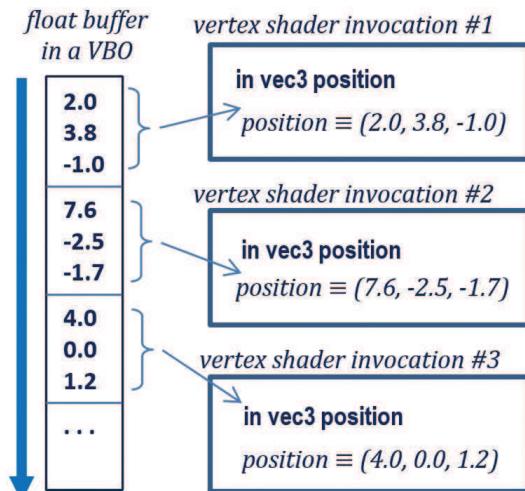


Figure 4.1

Data transmission between a VBO and a vertex attribute.

¹ Throughout this example, two buffers are declared, to emphasize that *usually* we will use several buffers. Later we will use the additional buffer(s) to store other information associated with the vertex, such as color. In the current case we are using only one of the declared buffers, so it would have been sufficient to declare just one VBO.

shader. In our cube example, a vertex attribute to receive the cube vertices could be declared in the vertex shader as follows:

```
layout (location = 0) in vec3 position;
```

The keyword `in` means “input” and indicates that this vertex attribute will be receiving values from a buffer (as we will see later, vertex attributes can also be used for “output”). As seen before, the “`vec3`” means that each invocation of the shader will grab *three* float values (presumably x , y , z , comprising a single vertex). The variable name is “`position`”. The “`layout (location=0)`” portion of the command is called a “`layout qualifier`” and is how we will associate the vertex attribute with a particular buffer. Thus this vertex attribute has an identifier 0 that we will use later for this purpose.

The manner in which we load the vertices of a model into a buffer (VBO) depends on where the model’s vertex values are stored. In Chapter 6 we will see how models are commonly built in a modeling tool (such as *Blender* [B16] or *Maya* [M16]), exported to a standard file format (such as `.obj`—also described in Chapter 6), and imported into the Java/JOGL program. We will also see how a model’s vertices can be calculated on the fly, or generated inside the pipeline, such as in the tessellation shader.

For now, let’s say that we wish to draw a *cube*, and let’s presume that the vertices of our cube are hardcoded in an array in the Java/JOGL application. In that case, we need to copy those values into one of our two buffers that we generated above. To do that, we need to (a) make that buffer (say, the 0th buffer) “active” with the OpenGL `glBindBuffer()` command, (b) copy the vertices into a Java `FloatBuffer`, and (c) use the `glBufferData()` command to copy the `FloatBuffer` into the active buffer (the 0th VBO in this case). Presuming that the vertices are stored in a float array named `vPositions`, the following JOGL code² would copy those values into the 0th VBO:

```
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
FloatBuffer vBuf = Buffers.newDirectFloatBuffer(vPositions);
gl glBufferData(GL_ARRAY_BUFFER, vBuf.limit()*4, vBuf, GL_STATIC_DRAW);
```

² Note that here, for the first time, we are refraining from describing every parameter in one or more JOGL calls. As mentioned in Chapter 2, the reader is encouraged to utilize the OpenGL documentation for such details as needed.

Java has two types of buffers: *direct* and *non-direct*. For performance reasons, *direct* buffers should be used in JOGL applications. JOGL provides tools in the class `com.jogamp.common.nio.Buffers` that facilitate the use of direct buffers. In the example above, the JOGL function `newDirectFloatBuffer()` copies values from an array to a `FloatBuffer`; in this case the vertices of the cube to be drawn.

Next, we add code to `display()` that will cause the values in the buffer to be sent to the `vertex` attribute in the shader. We do this with the following three steps: (a) make that buffer “active” with the `glBindBuffer()` command as we did above, (b) associate the active buffer with a vertex attribute in the shader, and (c) enable the `vertex` attribute. The following three lines of code will accomplish these steps:

```
gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);      // make the 0th buffer "active"
gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0); // associate 0th vertex attribute with active buffer
gl glEnableVertexAttribArray(0);                  // enable the 0th vertex attribute
```

Now when we execute `glDrawArrays()`, data in the 0th VBO will be transmitted to the `vertex` attribute that has a layout qualifier with location 0. This sends the cube vertices to the shader.

4.2 UNIFORM VARIABLES

Rendering a scene so that it appears 3D requires building appropriate transformation matrices, such as those described in Chapter 3, and applying them to each of the models’ vertices. It is most efficient to apply the required matrix operations in the vertex shader, and it is customary to send these matrices from the Java/JOGL application to the shader in a *uniform variable*.

Uniform variables are declared in a shader by using the “uniform” keyword. The following example, declaring variables to hold model-view and projection matrices, will be suitable for our cube program:

```
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

The keyword “mat4” indicates that these are 4×4 matrices. Here we have named the variables `mv_matrix` to hold the model-view matrix, and `proj_matrix` to hold the projection matrix. Since 3D transformations are 4×4 , `mat4` is a commonly used datatype in GLSL shader uniforms.

Sending data from a Java/JOGL application to a uniform variable requires the following steps: (a) acquire a pointer to the uniform variable, and (b) associate a Java float array containing the desired values with the acquired uniform pointer. Assuming that the linked rendering program is saved in a variable called “program”, the following lines of code would specify that we will be sending model-view and projection matrices to the two uniforms `mv_matrix` and `proj_matrix` in our cube example:

```
mLoc=gl.glGetUniformLocation(program,"mv_matrix");           // get the locations of the uniforms
pLoc = gl.glGetUniformLocation(program,"proj_matrix");        //   in the shader program
gl.glUniformMatrix4fv(mLoc, 1, false, mvMat.getFloatValues(), 0); // send matrix data to the
gl.glUniformMatrix4fv(pLoc, 1, false, pMat.getFloatValues(), 0); //   uniform variables
```

The above example assumes that we have utilized the `graphicslib3D` utilities to build model-view and projection matrix transforms `mvMat` and `pMat`, as will be discussed in greater detail shortly. They are of type `Matrix3D` (a `graphicslib3D` class), and the associated function call to `getFloatValues()` returns those matrix values in a float array, as needed by `glUniformMatrix4fv()`.

4.3 INTERPOLATION OF VERTEX ATTRIBUTES

It is important to understand how *vertex attributes* are processed in the OpenGL pipeline, versus how *uniform variables* are processed. Recall that immediately before the fragment shader is *rasterization*, where primitives (e.g., triangles) defined by vertices are converted to fragments. Rasterization *linearly interpolates vertex attribute* values so that the displayed pixels seamlessly connect the modeled surfaces.

By contrast, *uniform* variables behave like initialized constants, and remain unchanged across each vertex shader invocation (i.e., for each vertex sent from the buffer). A uniform variable is not interpolated; it always contains the same value regardless of the number of vertices.

The interpolation done on vertex attributes by the rasterizer is useful in many ways. Later, we will use rasterization to interpolate colors, texture coordinates, and surface normals. It is important to understand that *all values sent through a buffer to a vertex attribute will be interpolated* further down the pipeline.

We have seen vertex attributes in a vertex shader declared as “`in`”, to indicate that they receive values from a buffer. Vertex attributes may instead be declared

as “out”, meaning that they send their values forward towards the next stage in the pipeline. For example, the following declaration in a vertex shader specifies a vertex attribute named “color” that outputs a vec4:

```
out vec4 color;
```

It is not necessary to declare an “out” variable for the vertex positions, because OpenGL has a built-in out vec4 variable named `gl_Position` for that purpose. In the vertex shader, we apply the matrix transformations to the incoming vertex (declared earlier as `position`) assigning the result to `gl_Position`:

```
gl_Position = proj_matrix * mv_matrix * position;
```

The transformed vertices will then be automatically output to the rasterizer, with corresponding pixel locations ultimately sent to the fragment shader.

The rasterization process is illustrated in Figure 4.2. When specifying `GL_TRIANGLES` in the `glDrawArrays()` function, rasterization is done per triangle. Interpolation starts along the lines connecting the vertices, at a level of precision corresponding to the pixel display density. The pixels in the interior space of the triangle are then filled by interpolating along the horizontal lines connecting the edge pixels.

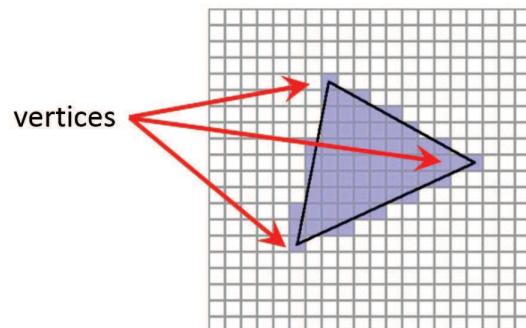


Figure 4.2
Rasterization of vertices.

4.4 MODEL-VIEW AND PERSPECTIVE MATRICES

A fundamental step in rendering an object in 3D is to create appropriate transformation matrices and send them to uniform variables like we did in Section 4.2. We start by defining three matrices:

1. a *Model* matrix
2. a *View* matrix
3. a *Perspective* matrix

The *Model* matrix positions and orients the object in the world coordinate space. Each model has its own model matrix, and that matrix would need to be continuously rebuilt if the model moves.

The *View* matrix moves and rotates the models in the world to simulate the effect of a camera at a desired location. Recall from Chapter 2 that the OpenGL camera exists at location (0,0,0) and faces down the negative Z axis. To simulate the appearance of that camera being moved a certain way, we will need to move the objects themselves in the opposite way. For example, moving a camera to the right would cause the objects in the scene to appear to move the left; although the OpenGL camera is fixed, we can make it appear as though we have moved it to the right by moving the objects to the left.

The *Perspective* matrix is a transform that provides the 3D effect according to the desired frustum, as described earlier in Chapter 3.

It is also important to understand when to compute each type of matrix. Matrices that never change can be built in `init()`, but those that change would need to be built in `display()` so that they are rebuilt for each frame. Let's assume that the models are animated, and the camera is movable. Then:

- A model matrix needs to be created *for each model*, and *at each frame*.
- The view matrix needs to be created *once per frame* (because the camera can be moved), but is the same for all objects rendered during that frame.
- The perspective matrix is created once (in `init()`) using the screen window's width and height (and desired frustum parameters), and usually remains unchanged unless the window is resized.

Generating model and view transformation matrices then happens in the `display()` function, as follows:

1. Build the view matrix based on the desired camera location and orientation.
2. For each model, do the following:
 - i. Build a model matrix based on the model's location and orientation.
 - ii. Concatenate the model and view matrices into a single “MV” matrix.
 - iii. Send the MV and projection matrices to the corresponding shader uniforms.

Technically, it isn't necessary to combine the model and view matrices into a single matrix. That is, they could be sent to the vertex shader in individual, separate matrices. However, there are certain advantages to combining them, while keeping

the perspective matrix separate. For example, in the vertex shader, each vertex in the model is multiplied by the matrices. Since complex models may have hundreds or even thousands of vertices, performance can be improved by pre-multiplying the model and view matrices once before sending them to the vertex shader. Later, we will see the need to keep the perspective matrix separate for lighting purposes.

4.5 OUR FIRST 3D PROGRAM – A 3D CUBE

It's time to put all the pieces together! In order to build a complete Java/JOGL/GLSL system to render our cube in a 3D “world,” all of the mechanisms described so far will need to be put together and perfectly coordinated. We can reuse some of the code that we have seen previously in Chapter 2. Specifically, we won't repeat the following functions for reading in files containing shader code, compiling and linking them, and detecting GLSL errors (but you still should use them):

- `createShaderProgram()`
- `readShaderSource()`
- `checkOpenGLError()`
- `printProgramLog()`
- `printShaderLog()`

First, let's create a utility function that builds a perspective matrix, given a specified field-of-view angle for the Y axis, the screen aspect ratio, and the desired near and far clipping planes (selecting appropriate values for near and far clipping planes is discussed in Section 4.9). This function, based on the formulas given in Figure 3.16, will be useful in our final cube program:

```
private Matrix3D perspective(float fovy, float aspect, float n, float f)
{   float q=1.0f / ((float) Math.tan(Math.toRadians(0.5f * fovy)));
    float A = q / aspect;
    float B = (n + f) / (n - f);
    float C = (2.0f * n * f) / (n - f);
    Matrix3D r = new Matrix3D();
    r.setElementAt(0,0, A);
    r.setElementAt(1,1, q);
    r.setElementAt(2,2, B);
    r.setElementAt(3,2, -1.0f);
    r.setElementAt(2,3, C);
    r.setElementAt(3,3, 0.0f);
    return r;
}
```

We can now build the complete 3D cube program, shown in Program 4.1 that follows:

Program 4.1 Plain Red Cube

Java/JOGL Application

```
import graphicslib3D.*;
import graphicslib3D.GLSLUtils.*;
import java.nio.*;
import javax.swing.*;
import static com.jogamp.opengl.GL4.*;
import com.jogamp.opengl.*;
import com.jogamp.opengl.awt.GLCanvas;
import com.jogamp.common.nio.Buffers;
import com.jogamp.opengl.GLContext;

public class Code extends JFrame implements GLEventListener
{
    private GLCanvas myCanvas;
    private int rendering_program;
    private int vao[ ] = new int[1];
    private int vbo[ ] = new int[2];
    private float cameraX, cameraY, cameraZ;
    private float cubeLocX, cubeLocY, cubeLocZ;
    private GLSLUtils util = new GLSLUtils();
    private Matrix3D pMat;

    public Code()
    {
        setTitle("Chapter4 - program1");
        setSize(600, 600);
        myCanvas = new GLCanvas();
        myCanvas.addGLEventListener(this);
        this.add(myCanvas);
        setVisible(true);
    }

    public void init(GLAutoDrawable drawable)
    {
        GL4 gl = (GL4) GLContext.getCurrentGL();
        rendering_program = createShaderProgram();
        setupVertices();
        cameraX=0.0f; cameraY=0.0f; cameraZ=8.0f;
        cubeLocX=0.0f; cubeLocY=-2.0f; cubeLocZ=0.0f; // shifted down along the Y-axis to reveal
                                                       // perspective
        // Create a perspective matrix, this one has fovy=60, aspect ratio matches screen window.
        // Values for near and far clipping planes can vary as discussed in Section 4.9.
    }
}
```

```

float aspect = (float) myCanvas.getWidth() / (float) myCanvas.getHeight();
pMat = perspective(60.0f, aspect, 0.1f, 1000.0f);
}

// main(), reshape(), and dispose() are unchanged
public static void main(String[ ] args)
{ new Code();
}

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) { }

public void dispose(GLAutoDrawable drawable) { }

public void display(GLAutoDrawable drawable)
{ GL4 gl = (GL4) GLContext.getCurrentGL();
  gl.glClear(GL_DEPTH_BUFFER_BIT);
  gl.glUseProgram(rendering_program);

  // build view matrix
  Matrix3D vMat = new Matrix3D();
  vMat.translate(-cameraX,-cameraY,-cameraZ);
  // build model matrix
  Matrix3D mMat = new Matrix3D();
  mMat.translate(cubeLocX, cubeLocY, cubeLocZ);
  // concatenate model and view matrix to create MV matrix
  Matrix3D mvMat = new Matrix3D();
  mvMat.concatenate(vMat);
  mvMat.concatenate(mMat);

  // copy perspective and MV matrices to corresponding uniform variables
  int mv_loc = gl.glGetUniformLocation(rendering_program, "mv_matrix");
  int proj_loc = gl.glGetUniformLocation(rendering_program, "proj_matrix");
  gl.glUniformMatrix4fv(proj_loc, 1, false, pMat.getFloatValues(), 0);
  gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);

  // associate VBO with the corresponding vertex attribute in the vertex shader
  gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
  gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
  gl glEnableVertexAttribArray(0);

  // adjust OpenGL settings and draw model
  gl glEnable(GL_DEPTH_TEST);
  gl glDepthFunc(GL_LEQUAL);
  gl glDrawArrays(GL_TRIANGLES, 0, 36);
}

private void setupVertices()
{ GL4 gl = (GL4) GLContext.getCurrentGL();
  // 36 vertices of the 12 triangles making up a 2 x 2 x 2 cube centered at the origin
}

```

```

float[ ] vertex_positions =
{
    -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f,
    1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f,
    1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f
};

gl glGenVertexArrays(vao.length, vao, 0);
gl glBindVertexArray(vao[0]);
gl glGenBuffers(vbo.length, vbo, 0);

gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
FloatBuffer vertBuf = Buffers.newDirectFloatBuffer(vertex_positions);
gl glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4, vertBuf, GL_STATIC_DRAW);
}

// ... createShaderProgram(), and perspective() are unchanged
}

```

Vertex shader

```

#version 430

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}

```

Fragment shader

```

#version 430

out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
    color = vec4(1.0, 0.0, 0.0, 1.0);
}

```

Let's take a close look at the code in Program 4.1. It is important that we understand how *all* of the pieces work, *and how they work together*.

Start by examining the function near the end of the Java/JOGL listing, `setupVertices()`, called by `init()`. At the start of this function, an array is declared called `vertex_positions` that contains the 36 vertices comprising the cube. At first you might wonder why this cube has 36 vertices, when logically a cube should only require 8. The answer is that we need to build our cube out of triangles, and so each of the six cube faces needs to be built of two triangles, for a total of $6 \times 2 = 12$ triangles (see Figure 4.4). Since each triangle is specified by three vertices, this totals 36 vertices. Since each vertex has three values (x, y, z), there are a total of $36 \times 3 = 108$ values in the array. It is true that each vertex participates in multiple triangles, but we still specify each vertex separately because for now we are sending the vertices of each triangle down the pipeline separately.

The cube is defined in its own coordinate system, with $(0,0,0)$ at its center, and with its corners ranging from -1.0 to $+1.0$ along the x , y , and z axes. The rest of the `setupVertices()` function sets up the VAO, two VBOs (although only one is used), and loads the cube vertices into the 0th VBO buffer.

Note that the `init()` function performs tasks that only need to be done once: reading in the shader code and building the rendering program, building the perspective matrix, and loading the cube vertices into the buffer. Note that it also positions the cube and the camera in the world . . . later we will animate the cube and also see how to move the camera around, at which point we may need to remove this hardcoded positioning.

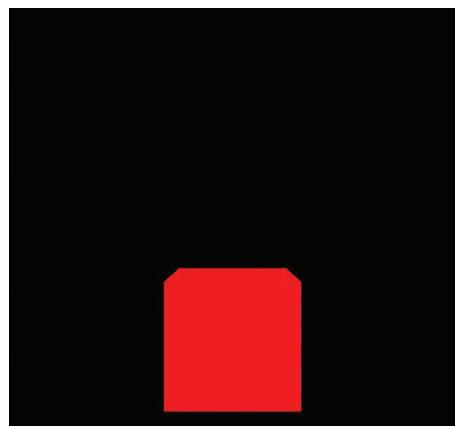


Figure 4.3

Output of Program 4.1. Red cube positioned at $(0, -2, 0)$ viewed from $(0, 0, 8)$.

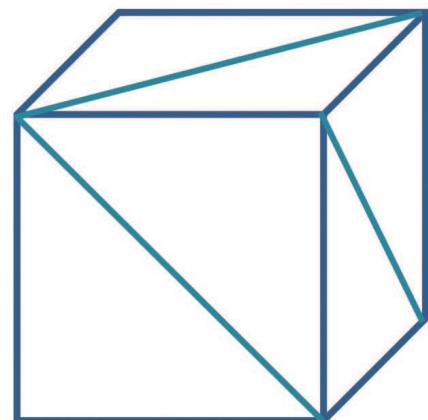


Figure 4.4

Cube made of triangles.

Now let's look at the `display()` function. Recall that `display()` may be called repeatedly and the rate at which it is called is referred to as the *frame rate*. That is, animation works by continually drawing and re-drawing the scene, or frame, very quickly. It is usually necessary to clear the depth buffer before rendering a frame, so that hidden surface removal occurs properly (not clearing the depth buffer can sometimes result in every surface being removed, resulting in a completely black screen). By default, depth values in OpenGL range from 0.0 to 1.0. Clearing the depth buffer is done by calling `glClear(GL_DEPTH_BUFFER_BIT)`, which fills the depth buffer with the default value (usually 1.0).

Next, `display()` enables the shaders by calling `glUseProgram()`, which installs the GLSL code on the GPU. Recall that this doesn't run the program, but it does allow JOGL to determine the shader's *vertex attribute* and *uniform* locations. The `display()` function next builds the view and model matrices, concatenates them into a single MV matrix, gets the uniform variable locations, and assigns the perspective and MV matrices to the corresponding uniforms.

Next, `display()` enables the buffer containing the cube vertices, and attaches it to 0th vertex attribute to prepare for sending the vertices to the shader.

The last thing `display()` does is draw the model by calling `glDrawArrays()`, specifying that the model is comprised of triangles and that model has 36 total vertices. The call to `glDrawArrays()` is typically preceded by additional commands that adjust rendering settings for this model.³ In this example, there are two such commands, both of which are related to depth testing. Recall from Chapter 2 that depth testing is used by OpenGL to perform hidden surface removal. Here, we enable depth testing, and specify the particular depth test we wish OpenGL to use. The settings shown here correspond to the description in Chapter 2; later in the book we will see other uses for these commands.

Finally, consider the shaders. First, note that they both include the same block of uniform variable declarations. Although this is not always required, it is often a good practice to include the same block of uniform variable declarations in all of the shaders within a particular rendering program.

³ Often, these calls may be placed in `init()` rather than in `display()`. However, it is necessary to place one or more of them in `display()` when drawing multiple objects with different properties. For simplicity, we always place them in `display()`.

Note also in the vertex shader the presence of the layout qualifier on the incoming vertex attribute position. Since the location is specified as “0”, the `display()` function can reference this variable simply by using 0 in the first parameter of the `glVertexAttribPointer()` function call, and in the `glEnableVertexAttribArray()` function call. Note also that the position vertex attribute is declared as a `vec3`, and so it needs to be converted to a `vec4` in order to be compatible with the 4×4 matrices with which it will be multiplied. This conversion is done with `vec4(position, 1.0)`, which builds a `vec4` out of the variable named “`position`”, putting a value of 1.0 in the newly added 4th spot.

The multiplication in the vertex shader applies the matrix transforms to the vertex, converting it to camera space (note the right-to-left concatenation order). Those values are put in the built-in OpenGL output variable `gl_Position`, and then proceed through the pipeline and are interpolated by the rasterizer.

The interpolated pixel locations (referred to as *fragments*) are then sent to the *fragment shader*. Recall that the primary purpose of the fragment shader is to set the color of an outputted pixel. In a manner similar to the vertex shader, the fragment shader processes the pixels one-by-one, with a separate invocation for each pixel. In this case, it outputs a hardcoded value corresponding to red. For reasons indicated earlier, the uniform variables have been included in the fragment shader even though they aren’t being used there in this example.

An overview of the flow of data starting with the Java/JOGL application and passing through the pipeline is shown in Figure 4.5, below:

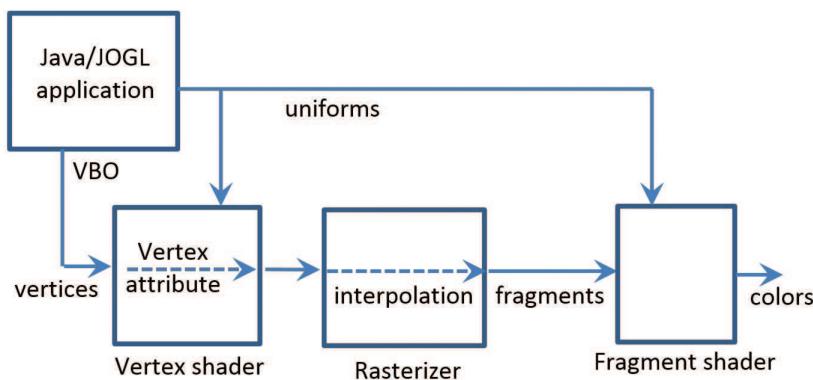


Figure 4.5

Data flow through Program 4.1.

Let's make a slight modification to the shaders. In particular, we will assign a color to each vertex according to its location, and put that color in the outgoing vertex attribute `varyingColor`. The fragment shader is similarly revised to accept the incoming color (interpolated by the rasterizer) and use that to set the color of the output pixel. Note that the code also multiplies the location by $\frac{1}{2}$ and then adds $\frac{1}{2}$ to convert the range of values from $(-1,+1)$ to $(0,1)$. Note also the use of the common convention of assigning variable names that include the word "varying" to programmer-defined interpolated vertex attributes. The changes in each shader are highlighted, and the resulting output shown below.

Revised vertex shader:

```
#version 430

layout (location=0) in vec3 position;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec4 varyingColor;

void main(void)
{   gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
    varyingColor = vec4(position,1.0) * 0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}
```

Revised fragment shader:

```
#version 430

in vec4 varyingColor;

out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

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

Note that because the colors are sent out from the vertex shader in a vertex attribute (`varyingColor`), that *they too are interpolated* by the rasterizer! The effect of this can be seen in Figure 4.6, where the colors from corner to corner are clearly interpolated smoothly throughout the cube.

Note also that the “out” variable `varyingColor` in the vertex shader is also the “in” variable in the fragment shader. The two shaders know which variable from the vertex shader feeds which variable in the fragment shader because they have the same name “`varyingColor`” in both shaders.

We can animate the cube using the `FPSAnimator` class as in Program 2.6, by building the model matrix using a varying translation and rotation based on the elapsed time. For example, the code in the `display()` function in Program 4.1 could be modified as follows (changes are highlighted):

```

gl.glClear(GL_DEPTH_BUFFER_BIT);
float bkg[] = { 0.0f, 0.0f, 0.0f, 1.0f };
FloatBuffer bkgBuffer = Buffers.newDirectFloatBuffer(bkg);
gl.glClearBufferfv(GL_COLOR, 0, bkgBuffer);

...
Matrix3D mMat = new Matrix3D();

// use system time to generate slowly-increasing sequence of floating-point values
double t = (double)(System.currentTimeMillis()) / 10000.0;

// use t to compute different translations in x, y, and z
mMat.translate(Math.sin(2*t)*2.0, Math.sin(3*t)*2.0, Math.sin(4*t)*2.0);
mMat.rotate(1000*t, 1000*t, 1000*t);      // the 1000 adjusts the rotation speed

```

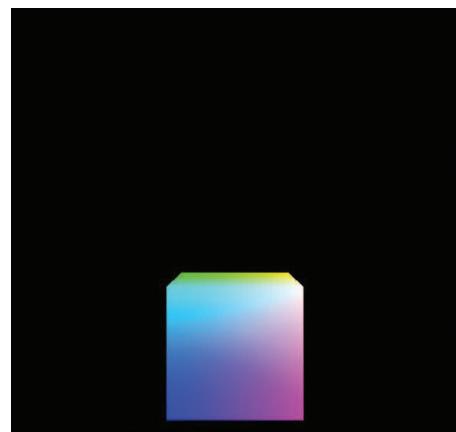


Figure 4.6
Cube with interpolated colors.

The use of elapsed time (and a variety of trigonometric functions) in the model matrix causes the cube to appear to tumble around in space. Note that adding this animation illustrates the importance of clearing the depth buffer each

time through `display()` to ensure correct hidden surface removal. It also necessitates clearing the `GL_COLOR` buffer as shown; otherwise, the cube will leave a trail as it moves. The Java `FloatBuffer` is filled with zeros, resulting in a black background color.

The `translate()` and `rotate()` functions are part of the `graphicslib3D` library. Note that in the code, `translate()` is called before `rotate()`. This results in a concatenation of the two transforms, with translation on the left and rotation on the right. When a vertex is subsequently multiplied by this matrix, the computation is right-to-left, meaning that the rotation is done first, followed by the translation. The order of application of transforms is significant and changing the order would result in different behavior. Figure 4.7 shows some of the frames that are displayed after animating the cube:

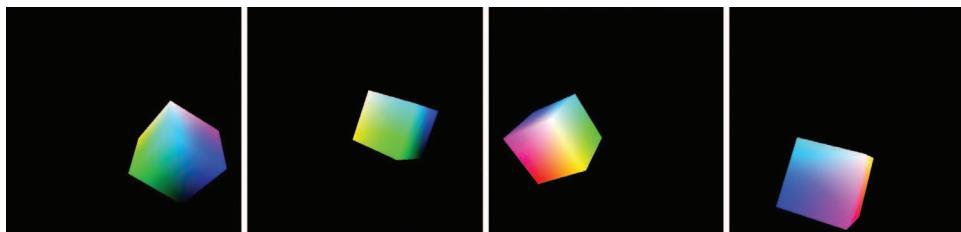


Figure 4.7
Animated (“tumbling”) 3D cube.

4.6 RENDERING MULTIPLE COPIES OF AN OBJECT

We now extend what we have learned to rendering multiple objects. Before we tackle the general case of rendering a variety of models in a single scene, let's consider the simpler case of *multiple occurrences of the same model*. Suppose, for instance, that we wish to expand the previous example so that it renders a “swarm” of 24 tumbling cubes. We can do this by moving the portions of the code in `display()` that build the MV matrix and that draw the cube (shown below in blue) into a loop that executes 24 times. We incorporate the loop variable into the cube's rotation and translation, so that each time the cube is drawn a different model matrix is built. (We also positioned the camera further down the positive Z axis so we can see all of the cubes.) The resulting animated scene is shown thereafter in Figure 4.8.

```

public void display(GLAutoDrawable drawable)
{
    ...
    double timeFactor = (double) (System.currentTimeMillis()) / 10000.0;

    for (int i=0; i<24; i++)
    {
        double x = i + timeFactor;
        Matrix3D mMat = new Matrix3D();
        mMat.translate(Math.sin(2*x)*6.0, Math.sin(3*x)*6.0, Math.sin(4*x)*6.0);
        mMat.rotate(1000*x, 1000*x, 1000*x);

        Matrix3D mvMat = new Matrix3D();
        mvMat.concatenate(vMat);
        mvMat.concatenate(mMat);

        gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);
        gl.glEnable(GL_DEPTH_TEST);
        gl.glDepthFunc(GL_EQUAL);
        gl.glDrawArrays(GL_TRIANGLES, 0, 36);
    }
}

```

4.6.1 Instancing

Instancing provides a mechanism for telling the graphics card to render multiple copies of an object using only a single Java call. This can result in a significant performance benefit, particularly when there are thousands or millions of copies of the object being drawn—such as when rendering many flowers in a field, or many zombies in an army.

We start by changing the `glDrawArrays()` call in our Java/JOGL application to `glDrawArraysInstanced()`. Now, we can ask OpenGL to draw as many copies as we want. We can specify drawing 24 cubes as follows:

```
glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 24);
```

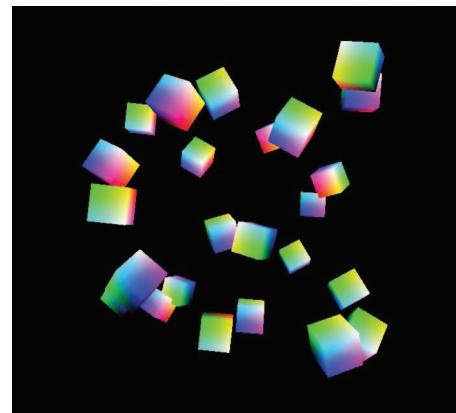


Figure 4.8
Multiple tumbling cubes.

When using instancing, the vertex shader has access to a built-in variable `gl_InstanceID`, an integer that refers to which numeric instance of the object is currently being processed.

To replicate our previous tumbling cubes example using instancing, we will need to move the computations that build the different model matrices (previously inside a loop in `display()`) into the vertex shader. Since GLSL does not provide translate or rotate functions, and we cannot make calls to `graphicslib3D` from inside a shader, we will need to use the utility functions from Program 3.1. We will also need to pass the “time factor” variable from the Java/JOGL application to the vertex shader in a uniform. We also need to pass the model and view matrices into separate uniforms because the rotation computations are applied to each cube’s model matrix. The revisions, including those in the Java/JOGL application and those in the new vertex shader, are shown in Program 4.2.

Program 4.2 Instancing – 24 Animated Cubes

Vertex Shader:

```
#version 430
layout (location=0) in vec3 position;

uniform mat4 m_matrix;           // these are now separate model and view matrices
uniform mat4 v_matrix;
uniform mat4 proj_matrix;
uniform float tf;                // time factor for animation and placement of cubes

out vec4 varyingColor;

mat4 buildRotateX(float rad);    // declaration of matrix transformation utility functions
mat4 buildRotateY(float rad);    // (GLSL requires functions to be declared prior to invocation)
mat4 buildRotateZ(float rad);
mat4 buildTranslate(float x, float y, float z);

void main(void)
{   float i = gl_InstanceID + tf; // value based on time factor, but different for each cube instance
    float a = sin(2.0 * i) * 8.0; // these are the x, y, and z components for the translation, below
    float b = sin(3.0 * i) * 8.0;
    float c = sin(4.0 * i) * 8.0;

    // build the rotation and translation matrices to be applied to this cube's model matrix
    mat4 localRotX = buildRotateX(1000*i);
```

```

mat4 localRotY = buildRotateY(1000*i);
mat4 localRotZ = buildRotateZ(1000*i);
mat4 localTrans = buildTranslate(a,b,c);

// build the model matrix and then the model-view matrix
mat4 newM_matrix = m_matrix * localTrans * localRotX * localRotY * localRotZ;
mat4 mv_matrix = v_matrix * newM_matrix;

gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
varyingColor = vec4(position,1.0) * 0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}

// utility function to build a translation matrix (from Chapter 3)
mat4 buildTranslate(float x, float y, float z)
{
    mat4 trans = mat4( 1.0, 0.0, 0.0, 0.0,
                      0.0, 1.0, 0.0, 0.0,
                      0.0, 0.0, 1.0, 0.0,
                      x, y, z, 1.0 );
    return trans;
}
// similar functions included for rotation around the X, Y, and Z axes (also from Chapter 3)
...

```

Java/JOGL Application (in `display()`)

```

...
// since the timeFactor is used in the shader, use mod()% to reduce its magnitude
double timeFactor = (double) (System.currentTimeMillis()%3600000)/10000.0;

// computations that transform mMat have been moved to the vertex shader.
gl.glUniformMatrix4fv(m_loc, 1, false, mMat.getFloatValues(), 0); // getGetUniform() is now split
gl.glUniformMatrix4fv(v_loc, 1, false, vMat.getFloatValues(), 0); // into two separate calls.
int tf_loc = gl.glGetUniformLocation(rendering_program, "tf"); // uniform for the time factor
gl.glUniform1f(tf_loc, (float)timeFactor);

...
gl.glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 24);

```

The resulting output of Program 4.2 is identical to that for the previous example, and can be seen in the previous Figure 4.8.

Instancing makes it possible to greatly expand the number of copies of an object; in this example animating 100,000 cubes is still feasible even for a modest

GPU. The changes to the code—mainly just a few modified constants to spread the large number of cubes further apart—are as follows:

Vertex Shader:

```
...
float a = sin(203.0 * i/4000.0) * 403.0;
float b = cos(301.0 * i/2001.0) * 401.0;
float c = sin(400.0 * i/3003.0) * 405.0;
...
```

Java/Jogl Application

```
...
gl.glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 100000);
```

The resulting output is shown below in Figure 4.9.

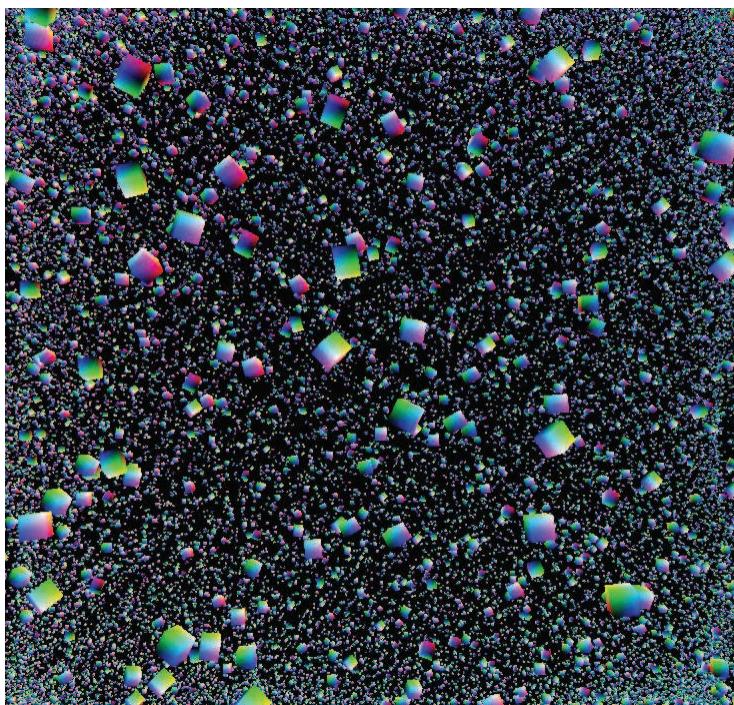


Figure 4.9

Instancing: 100,000 animated cubes.

4.7 RENDERING MULTIPLE DIFFERENT MODELS IN A SCENE

To render more than one model in a single scene, a simple approach is to use a separate buffer for each model. Each model will need its own model matrix, and thus a new model-view matrix will be generated for each model that we render. There will also need to be separate calls to `glDrawArrays()` for each model. There are changes both in `init()` and in `display()`.

Another consideration is whether or not we will need different shaders—or a different rendering program—for each of the objects we wish to draw. As it turns out, *in many cases we can use the same shaders* (and thus the same rendering program) for the various objects we are drawing. We usually only need to employ different rendering programs for the various objects if they are built of different primitives (such as lines instead of triangles), or if there are complex lighting or other effects involved. For now, that isn't the case, so we can reuse the same vertex and fragment shaders, and just modify our Java/JOGL application to send each model down the pipeline when `display()` is called.

Let's proceed by adding a simple pyramid, so our scene includes both a single cube and a pyramid. The code is shown in Program 4.3. A few of the key details are highlighted, such as where we specify one or the other buffer, and where we specify the number of vertices contained in the model. Note that the pyramid is comprised of 6 triangles—four on the sides and two on the bottom, totaling $6 \times 3 = 18$ vertices.

The resulting scene, containing both the cube and the pyramid, is then shown in Figure 4.10.

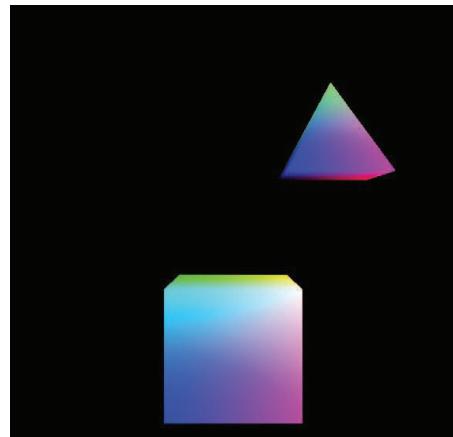


Figure 4.10
3D cube and pyramid.

Program 4.3 Cube and Pyramid

```

private void setupVertices()
{ GL4 gl = (GL4) GLContext.getCurrentGL();

    float[ ] cube_positions =
    { -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
      1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f,
      ... same as before, for the rest of the cube vertices
    };

    // pyramid with 18 vertices, comprising 6 triangles (four sides, and two on the bottom)
    float[ ] pyramid_positions =
    { -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f,           // front face
      1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f,           // right face
      1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,         // back face
      -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,         // left face
      -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f,        // base - left front
      1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,        // base - right back
    };

    gl glGenVertexArrays(vao.length, vao, 0);
    gl glBindVertexArray(vao[0]);
    gl glGenBuffers(vbo.length, vbo, 0);
    gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    FloatBuffer cubeBuf = Buffers.newDirectFloatBuffer(cube_positions);
    glBufferData(GL_ARRAY_BUFFER, cubeBuf.limit()*4, cubeBuf, GL_STATIC_DRAW);
    gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    FloatBuffer pyrBuf = Buffers.newDirectFloatBuffer(pyramid_positions);
    glBufferData(GL_ARRAY_BUFFER, pyrBuf.limit()*4, pyrBuf, GL_STATIC_DRAW);
}

public void display(GLAutoDrawable drawable)
{ GL4 gl = (GL4) GLContext.getCurrentGL();
  . . . clear the color and depth buffers as before (not shown here)
  . . . use rendering program and obtain uniform locations as before (not shown here)

// ----- set up the projection and view matrices
  float aspect = (float) myCanvas.getWidth() / (float) myCanvas.getHeight();
  Matrix3D pMat = perspective(60.0f, aspect, 0.1f, 1000.0f);
  Matrix3D vMat = new Matrix3D();
  vMat.translate(-cameraX,-cameraY,-cameraZ);

// ----- draw the cube (use buffer #0)
  Matrix3D mMat = new Matrix3D();
  mMat.translate(cubeLocX, cubeLocY, cubeLocZ);

  Matrix3D mvMat = new Matrix3D();
  mvMat.concatenate(vMat);
}

```

```

mvMat.concatenate(mMat);

gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);
gl.glUniformMatrix4fv(proj_loc, 1, false, pMat.getFloatValues(), 0);

gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

gl glEnable(GL_DEPTH_TEST);
gl depthFunc(GL_LEQUAL);

gl.drawArrays(GL_TRIANGLES, 0, 36);

// ----- draw the pyramid (use buffer #1)
mMat = new Matrix3D();
mMat.translate(pyrLocX, pyrLocY, pyrLocZ);

mvMat = new Matrix3D();
mvMat.concatenate(vMat);
mvMat.concatenate(mMat);

gl glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);
gl glUniformMatrix4fv(proj_loc, 1, false, pMat.getFloatValues(), 0); // (repeated for clarity)

gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

gl glEnable(GL_DEPTH_TEST);
gl depthFunc(GL_LEQUAL);

gl.drawArrays(GL_TRIANGLES, 0, 18);
}

```

A few other minor details to note regarding Program 4.3:

- The variables `pyrLocX`, `pyrLocY`, and `pyrLocZ` need to be declared in the class, and initialized in `init()` to the desired pyramid location, as was done for the cube location.
- The view matrix `vMat` is built at the top of `display()`, and then used in both the cube's and the pyramid's model-view matrices.
- The vertex and fragment shaders are not shown—they are unchanged from Section 4.5.

4.8 MATRIX STACKS

So far, the models we have rendered have each been constructed of a single set of vertices. It is often desired, however, to build complex models by assembling

smaller, simple models. For example, a model of a “robot” could be created by separately drawing the head, body, legs, and arms, where each of those is a separate model. An object built in this manner is often called a *hierarchical model*. The tricky part of building hierarchical models is keeping track of all the model-view matrices, and making sure they stay perfectly coordinated—otherwise the robot might fly apart into pieces!

Hierarchical models are useful not only for building complex objects—they can also be used to generate complex scenes. For example, consider how our planet Earth revolves around the sun, and in turn how the moon revolves around the Earth. Computing the moon’s actual path through space could be complex. However, if we can *combine* the transforms representing the two simple circular paths—the moon’s path around the Earth and the Earth’s path around the sun—we avoid having to explicitly compute the moon trajectory.

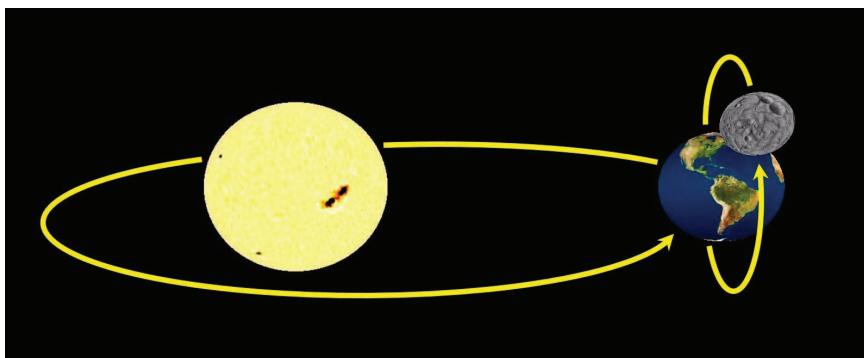


Figure 4.11

Animated planetary system (sun and earth textures from [HT16], moon texture from [NA16]).

It turns out that we can do this fairly easily with a *matrix stack*. A matrix stack is, as its name implies, a stack of transformation matrices. As we will see, matrix stacks make it easy to create and manage complex hierarchical objects and scenes, where transforms can be built upon (and removed from) other transforms.

OpenGL has a built-in matrix stack, but as part of the older fixed-function (non-programmable) pipeline it has long been deprecated. It was nicely devised and convenient to use, so we have created a similar Java class in our graphicslib3D library called `MatrixStack`, patterned after the one that became popular in older versions of OpenGL [OL16]. As we will see, many of the model, view, and model-view

matrices that would normally be needed in a complex scene can be replaced by a single instance of `MatrixStack`.

We will first examine the basic commands for instantiating and utilizing a matrix stack, then use `MatrixStack` to build a complex animated scene. The most important `MatrixStack` functions are:

- `pushMatrix()` – makes a copy of the top matrix, and pushes the copy onto the stack
 - `popMatrix()` – removes and returns the top matrix
 - `multMatrix(m)` – concatenates matrix `m` onto the top matrix in the stack
 - `peek()` – returns the topmost matrix, without removing it from the stack
 - `rotate(d,x,y,z)`
 - `scale(x,y,z)`
 - `translate(x,y,z)`
- } applied directly to the
top matrix in the stack

Now, rather than building transforms by creating instances of `Matrix3D`, we instead use the `pushMatrix()` command to create new matrices at the top of the stack. Desired transforms are then concatenated as needed to the newly created matrix on the top of the stack.

The first matrix pushed on the stack is frequently the `VIEW` matrix. The matrices above it are model-view matrices of increasing complexity; that is, they have an increasing number of model transforms applied to them. These transforms can either be applied directly, or by first concatenating other matrices.

In our planetary system example, the matrix positioned immediately above the `VIEW` matrix would be the sun’s `MV` matrix. The matrix on top of that matrix would be the earth’s `MV` matrix, *which consists of the sun’s MV matrix concatenated with the Earth’s model matrix*. That is, the Earth’s `MV` matrix is built by incorporating the planet’s transforms into the sun’s transforms. Similarly, the moon’s `MV` matrix sits on top of the planet’s `MV` matrix, and is constructed by concatenating the planet’s `MV` matrix with the moon’s model matrix.

After rendering the moon, a second “moon” could be rendered by “popping” the first moon’s matrix off of the stack (restoring the top of the stack to the planet’s model-view matrix) and then repeating the process for the second moon.

The basic approach is as follows:

1. When a new object is introduced relative to a parent object, do a “push” and a “multiply.”

2. When an object or sub-object is completed, “pop” its model-view matrix, removing it from atop the matrix stack.

In later chapters we will learn how to create spheres and make them look like planets and moons. For now, to keep things simple, we will build a “planetary system” using our pyramid and a couple of cubes.

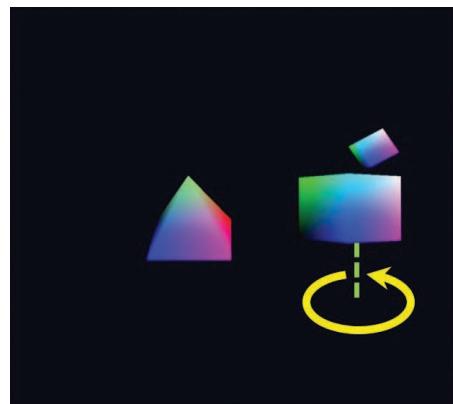
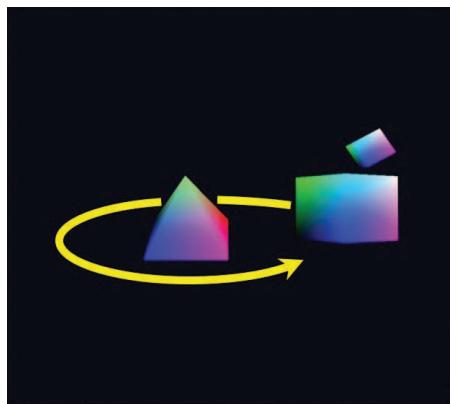
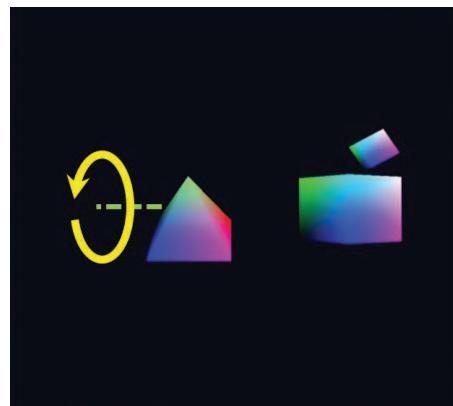
Here is an overview of how a `display()` function using a matrix stack is typically organized:

setup	<ul style="list-style-type: none"> ● Instantiate the MatrixStack
camera	<ul style="list-style-type: none"> ● Push a new matrix onto the stack (this will instantiate an empty VIEW matrix) ● apply transform(s) to the view matrix on the top of the stack
Parent	<ul style="list-style-type: none"> ● Push a new matrix onto the stack (this will be the parent MV matrix – for the first parent, it duplicates the view matrix) ● apply transforms to incorporate the parent’s M matrix into the duplicated view matrix ● send the topmost matrix (“peek”) to the MV uniform variable in the vertex shader ● draw the parent object
Child	<ul style="list-style-type: none"> ● Push a new matrix onto the stack. This will be the child’s MV matrix, duplicating the parent MV matrix. ● apply transforms to incorporate the child’s M matrix into the duplicated parent MV matrix ● send the topmost matrix (“peek”) to the MV uniform variable in the vertex shader ● draw the child object
cleanup	<ul style="list-style-type: none"> ● Pop the child’s MV matrix off the stack ● Pop the parent’s MV matrix off the stack ● Pop the VIEW matrix off the stack

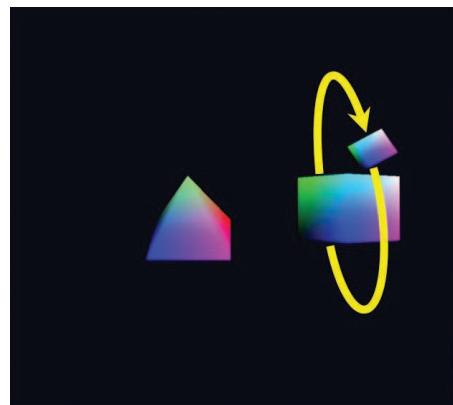
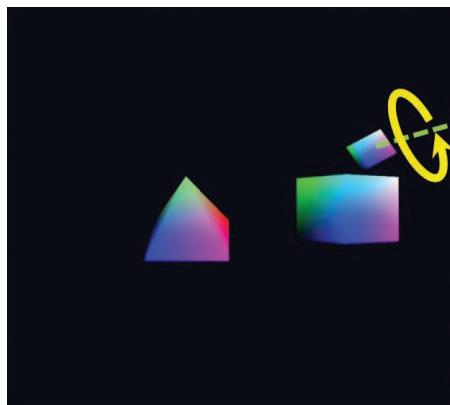
The pyramid (“sun”) rotation on its axis is in its own local coordinate space, and does not affect the “children.” Therefore, the sun’s rotation is pushed onto the stack,

but then after drawing the sun, it would be removed (popped) from the stack.

The big cube's (planet) revolution around the sun (left figure below) will affect the moon's movement, and so it is pushed on the stack and remains there when drawing the moon as well. By contrast, the planet's rotation on its axis (right figure below) is local and does not affect the moon, so it is popped off the stack before drawing the moon.



Similarly, we would push transforms onto the stack for the moon's rotations (on its axis, and around the planet), indicated in the following images.



Here are the steps for the “planet”:

- **push()** a new matrix. This will be the portion of the planet’s MV matrix that will also affect children.
- **translate(...)** to incorporate the planet movement around the sun into the planet’s MV matrix. In this example we use trigonometry to calculate the planet movement as a translation.
- **push()** a new matrix. This will be the planet MV matrix.
- **rotate(...)** to incorporate the planet’s axis rotation (this will later be popped and not affect children).
- **peek()** to obtain the MV matrix and then send it to the MV uniform.
- Draw the planet.
- **pop()** the planet MV matrix off the stack, exposing underneath it an earlier copy of the planet MV matrix without the planet’s axis rotation (so that only the planet’s translation affects the moon).

We now can write the complete `display()` routine, shown in Program 4.4.

Program 4.4 Simple Solar System Using Matrix Stack

```
public void display(GLAutoDrawable drawable)
{
    // setup of background, depth buffer, rendering program, and proj matrices unchanged
    ...
    MatrixStack mvStack = new MatrixStack(20);

    // push view matrix onto the stack
    mvStack.pushMatrix();
    mvStack.translate(-cameraX, -cameraY, -cameraZ);
    double amt = (double)(System.currentTimeMillis()) / 1000.0;

    // ----- pyramid == sun -----
    mvStack.pushMatrix();
    mvStack.translate(pyrLocX, pyrLocY, pyrLocZ);                                // sun's position
    mvStack.pushMatrix();
    mvStack.rotate((System.currentTimeMillis()) / 10.0, 1.0, 0.0, 0.0); // sun's rotation on its axis
    gl.glUniformMatrix4fv(mv_loc, 1, false, mvStack.peek().getFloatValues(), 0);
    gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    gl glEnableVertexAttribArray(0);
    gl glEnable(GL_DEPTH_TEST);
```

```

gl.glDepthFunc(GL_LESS);
gl.glDrawArrays(GL_TRIANGLES, 0, 18);    // draw the sun
mvStack.popMatrix();                      // remove the sun's axial rotation from the stack

//----- cube == planet -----
mvStack.pushMatrix();
mvStack.translate(Math.sin(amt)*4.0f, 0.0f, Math.cos(amt)*4.0f); // planet moves around sun
mvStack.pushMatrix();
mvStack.rotate((System.currentTimeMillis())*10.0, 0.0, 1.0, 0.0); // planet axis rotation
gl.glUniformMatrix4fv(mv_loc, 1, false, mvStack.peek().getFloatValues(), 0);
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);
gl.glDrawArrays(GL_TRIANGLES, 0, 36); // draw the planet
mvStack.popMatrix();                      // remove the planet's axial rotation from the stack

//----- smaller cube == moon -----
mvStack.pushMatrix();
mvStack.translate(0.0f, Math.sin(amt)*2.0f, Math.cos(amt)*2.0f); // moon moves around planet
mvStack.rotate((System.currentTimeMillis())*10.0, 0.0, 0.0, 1.0); // moon's rotation on its axis
mvStack.scale(0.25, 0.25, 0.25); // make the moon smaller
gl.glUniformMatrix4fv(mv_loc, 1, false, mvStack.peek().getFloatValues(), 0);
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);
gl.glDrawArrays(GL_TRIANGLES, 0, 36); // draw the moon

// remove moon scale/rotation/position, planet position, sun position, and view matrices from stack
mvStack.popMatrix(); mvStack.popMatrix(); mvStack.popMatrix(); mvStack.popMatrix();
}

```

The matrix stack operations have been highlighted. There are several details worth noting:

- We have introduced a *scale* operation in a model matrix. We want the moon to be a smaller cube than the planet, so we use a call to *scale()* when building the MV matrix for the moon.
- In this example, we are using the trigonometric operations *sin()* and *cos()* to compute the revolution of the planet around the sun (as a translation), and also for the moon around the planet.
- The two buffers #0 and #1 contain cube and pyramid vertices respectively.

- Note the use of the `peek()` function call within the `glUniformMatrix()` command. The `peek()` call retrieves the values in the matrix on top of the stack, and those values are then sent to the uniform variable (in this case, the sun, planet, and then moon's MV matrices).

The vertex and fragment shaders are not shown—they are unchanged from the previous example. We also moved the initial position of the pyramid (sun) and the camera to center the scene on the screen.

4.9 COMBATING “Z-FIGHTING” ARTIFACTS

Recall that when rendering multiple objects, OpenGL uses the *Z-buffer algorithm* (shown earlier in Figure 2.14) for performing hidden surface removal. Ordinarily, this resolves which object surfaces are visible and rendered to the screen, versus which surfaces lie behind other objects and thus should not be rendered, by choosing a pixel's color to be that of the corresponding fragment closest to the camera.

However, there can be occasions when two object surfaces in a scene overlap and lie in coincident planes, making it problematic for the Z-buffer algorithm to determine which of the two surfaces should be rendered (since neither is “closest” to the camera). When this happens, floating point rounding errors can lead to some portions of the rendered surface using the color of one of the objects, and other portions using the color of the other object. This artifact is known as *Z-fighting* or *depth-fighting*, because the effect is the result of rendered fragments “fighting” over mutually corresponding pixel entries in the Z-buffer. Figure 4.12 shows an example of Z-fighting between two boxes with overlapping coincident (top) faces.

Situations like this often occur when creating terrain or shadows. It is often possible to predict Z-fighting in such instances, and a common way of correcting it in these cases is to move one object slightly, so that the surfaces are no longer coplanar. We will see an example of this in Chapter 8.

Z-fighting can also occur due to limited precision of the values in the depth buffer. For each pixel processed by the Z-buffer algorithm, the accuracy of its depth

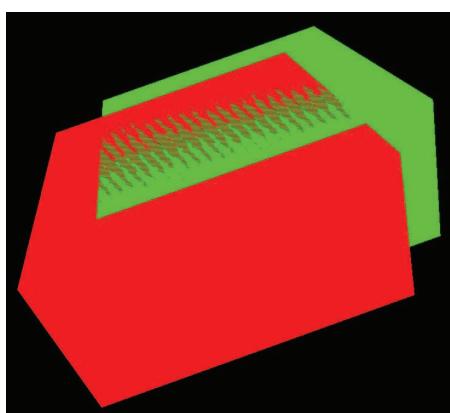


Figure 4.12
Z-fighting example.

information is limited by the number of bits available for storing it in the depth buffer. The greater the range between near and far clipping planes, the more likely two object's points with similar (but not equal) actual depths will end being represented by the same numeric value in the depth buffer. Therefore, it is up to the programmer to select near and far clipping plane values to minimize the distance between the two planes, while still ensuring that all objects essential to the scene lie within the viewing frustum.

It is also important to understand that, due to the effect of the perspective transform, changing the near clipping plane value can have a greater impact on the likelihood of Z-fighting artifacts than making an equivalent change in the far clipping plane. Therefore, it is advisable to avoid selecting a near clipping plane that is too close to the eye.

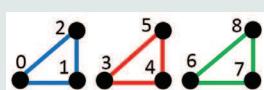
Previous examples in this book have simply used values of 0.1 and 1000 (in our calls to `perspective()`) for the near and far clipping planes. These may need to be adjusted for your scene.

4.10 OTHER OPTIONS FOR PRIMITIVES

OpenGL supports a number of primitive types—so far we have seen two: `GL_TRIANGLES` and `GL_POINTS`. In fact, there are several others. All of the available primitive types supported in OpenGL fall into the categories of *triangles*, *lines*, *points*, and *patches*. Here is a complete list:

Triangle primitives:

`GL_TRIANGLES`

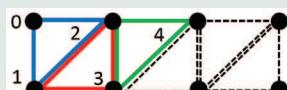


The most common primitive type in this book. Vertices that pass through the pipeline form distinct triangles:

vertices: [0 1 2] [3 4 5] [6 7 8] etc.

triangles: ✓ ✓ ✓

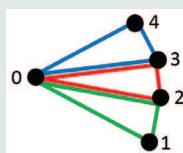
`GL_TRIANGLE_STRIP`



Each vertex that passes through the pipeline efficiently forms a triangle with the previous two vertices:

vertices: [0 1 2 3] 4 etc.

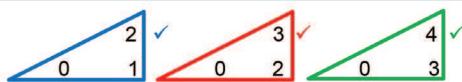
triangles: ✓ ✓ ✓

GL_TRIANGLE_FAN

Each pair of vertices that passes through the pipeline forms a triangle with the very first vertex:

vertices: 0 1 2 3 4 etc.

triangles:

**GL_TRIANGLES_ADJACENCY**

Intended only for use with geometry shaders. Allows the shader to access the vertices in the current triangle, plus additional adjacent vertices.

GL_TRIANGLE_STRIP_ADJACENCY

Intended only for use with geometry shaders.

Similar to GL_TRIANGLES_ADJACENCY, except that triangle vertices overlap as for GL_TRIANGLE_STRIP.

Line primitives:**GL_LINES**

Vertices that pass through the pipeline form distinct lines:

vertices: 0 1 2 3 4 5 etc.

lines:

**GL_LINE_STRIP**

Each vertex that passes through the pipeline efficiently forms a line with the previous vertex:

vertices: 0 1 2 3 etc.

lines:

**GL_LINE_LOOP**

Same as GL_LINE_STRIP, except a line is also formed between the very first and very last vertices.

GL_LINES_ADJACENCY

Intended for use with geometry shaders. Allows the shader to access the vertices in the current line, plus additional adjacent vertices.

GL_LINE_STRIP_ADJACENCY

Similar to GL_LINES_ADJACENCY, except that line vertices overlap as for GL_LINE_STRIP.

Point primitives:**GL_POINTS**

Each vertex that passes through the pipeline is a point.

Patch primitives:

GL_PATCH	Intended for use only with tessellation shaders. Indicates that a set of vertices passes from the vertex shader to the tessellation control shader, where they are typically used to shape a tessellated grid into a curved surface.
----------	--

4.11 BACK-FACE CULLING

As the complexity of our 3D scenes grows, we will become increasingly concerned with performance. One simple and effective way of improving rendering efficiency is to take advantage of OpenGL’s ability to do *back-face culling*.

When a 3D model is entirely “closed,” meaning the interior is never visible (such as for the cube and for the pyramid), then it turns out that those portions of the outer surface that are angled away from the viewer will always be obscured by some other portion of the same model. That is, those triangles that face away from the viewer cannot possibly be seen (they would be overwritten by hidden surface removal anyway), and thus there is no reason to rasterize or render them.

We can ask OpenGL to identify and “cull” (not render) back-facing triangles with the command `glEnable(GL_CULL_FACE)`. We can also disable face culling with `glDisable(GL_CULL_FACE)`. By default, face culling is disabled, so if you want OpenGL to cull back-facing triangles, you must enable it.

When face culling is enabled, by default triangles are rendered only if they are *front-facing*. Also by default a triangle is considered front-facing if its three vertices progress in a *counter-clockwise* direction (based on the order that they were defined in the buffer) as viewed from the OpenGL camera. Triangles whose vertices progress in a clockwise direction (as viewed from the OpenGL camera) are *back-facing*, and are not rendered. This counter-clockwise definition of “front-facing” is sometimes called the *winding order*, and can be set explicitly using the function call `gl_FrontFace(GL_CCW)` for counter-clockwise (the default), or `gl_FrontFace(GL_CW)` for clockwise. Similarly, whether it is the front-facing or the back-facing triangles that are rendered can also be set explicitly. Actually, for this purpose we specify which ones are *not* to be rendered—that is, which ones are “culled.” We can specify that the back-facing triangles be culled (although this would be unnecessary because it is the default) by calling `glCullFace(GL_BACK)`.

Alternatively, we can specify instead that the front-facing triangles be culled, or even that all of the triangles be culled, by replacing the parameter `GL_BACK` with either `GL_FRONT` or `GL_FRONT_AND_BACK`, respectively.

As we will see in Chapter 6, 3D models are typically designed so that the outer surface is constructed of triangles with the same winding order—most commonly counter-clockwise—so that if culling is enabled, then by default the portion of the model’s outer surface that faces the camera is rendered. Since by default OpenGL assumes the winding order is counter-clockwise, if a model is designed to be displayed with a *clockwise* winding order, it is up to the programmer to call `glFrontFace(GL_CW)` to account for this if back face culling is enabled.

Note that in the case of `GL_TRIANGLE_STRIP`, the winding order of each triangle alternates. OpenGL compensates for this by “flipping” the vertex sequence when building each successive triangle, as follows: 0-1-2, then 2-1-3, 2-3-4, 4-3-5, 4-5-6, and so on.

Back face culling improves performance by ensuring that OpenGL doesn’t spend time rasterizing and rendering surfaces that are never intended to be seen. Most of the examples we have seen in this chapter are so small that there is little motivation to enable face culling (an exception is the example shown in Figure 4.9, with the 100,000 instanced animated cubes, which may pose a performance challenge on some systems). In practice, it is common for most 3D models to be “closed,” and so it is customary to routinely enable back-face culling. For example, we can add back-face culling to Program 4.3 by modifying the `display()` function as follows:

```
public void display(GLAutoDrawable drawable)
{
    ...
    gl.glEnable(GL_CULL_FACE);
    // ----- draw the cube
    ...
    gl.glEnable(GL_DEPTH_TEST);
    gl.glDepthFunc(GL_LEQUAL);
    gl.glFrontFace(GL_CW);           // the cube vertices have clockwise winding order
    gl.glDrawArrays(GL_TRIANGLES, 0, 36);
    // ----- draw the pyramid (use buffer #1)
    ...
    gl.glEnable(GL_DEPTH_TEST);
    gl.glDepthFunc(GL_LEQUAL);
```

```

gl.glFrontFace(GL_CCW);           // the pyramid vertices have counter-clockwise winding order
gl.glDrawArrays(GL_TRIANGLES, 0, 18);
}

```

Properly setting the winding order is important when using back-face culling. An incorrect setting, such as GL_CW when it should be GL_CCW, can lead to the interior of an object being rendered rather than its exterior, which in turn can produce distortion similar to that of an incorrect perspective matrix.

Efficiency isn't the only reason for doing face culling. In later chapters we will see other uses, such as for those circumstances when we want to see the *inside* of a 3D model, or when using transparency.

SUPPLEMENTAL NOTES

As mentioned in this chapter, when using Java buffers it is advisable to ensure that only direct buffers are used. For example, it is tempting to use the Java `FloatBuffer.wrap()` function to put vertex data stored in an array into a buffer. However, `wrap()` produces a *non-direct* buffer, in which the resulting buffer is a wrapper for data that is still stored in an array⁴. One approach for copying the data into a *direct* buffer is to allocate a direct `ByteBuffer`, use `ByteBuffer.asFloatBuffer()` to view it as a `FloatBuffer`, and then use a loop with the `put()` command to copy the data one at a time. As described earlier, the JOGL `com.jogamp.common.nio.Buffers` class makes it easier to populate direct buffers by providing convenience methods for loading array data into a variety of buffer types:

```

newDirectByteBuffer(byte[])
newDirectCharBuffer(char[])
newDirectDoubleBuffer(double[])
newDirectFloatBuffer(float[])
newDirectIntBuffer(int[])
newDirectLongBuffer(long[])
newDirectShortBuffer(short[])

```

Using the above JOGL methods for populating buffers with vertex data (that is, data intended to be used by JOGL applications) also insures correct native byte

⁴ Using non-direct buffers is actually allowable in JOGL. However, when a non-direct buffer is passed to a JOGL method, JOGL will convert it to a direct buffer, which incurs a performance penalty.

ordering. It also ensures correct size allocation, and makes an explicit allocation unnecessary.

There is a myriad of other capabilities and structures available for managing and utilizing data in OpenGL/JOGL/GLSL, and we have only scratched the surface in this chapter. We haven't, for example, described a *uniform block*, which is a mechanism for organizing uniform variables similar to a *struct* in C. Uniform blocks can even be set up to receive data from buffers. Another powerful mechanism is a *shader storage block*, which is essentially a buffer into which a shader can write.

An excellent reference on the many options for managing data (albeit in C++) is the *OpenGL SuperBible* [SW15], particularly the chapter entitled "Data" (Chapter 5 in the 7th edition). It also describes many of the details and options for the various commands that we have covered. The first two example programs in this chapter, Program 4.1 and Program 4.2, were inspired by similar examples in the *SuperBible*.

There are other types of data that we will need to learn how to manage, and how to send down the OpenGL pipeline. One of these is a *texture*, which contains color image data (such as in a photograph) which can be used to "paint" the objects in our scene. We will study texture images in Chapter 5. Another important buffer that we will study further is the *depth buffer* (or Z-buffer). This will become important when we study shadows in Chapter 8. We still have much to learn about managing graphics data in OpenGL!

Exercises

- 4.1 (*PROJECT*) Modify Program 4.1 to replace the cube with some other simple 3D shape of your own design. Be sure to properly specify the number of vertices in the `glDrawArrays()` command.
- 4.2 (*PROJECT*) In Program 4.1, the "view" matrix is defined in the `display()` function simply as the negative of the camera location:

```
Matrix3D vMat = new Matrix3D();
vMat.translate(-cameraX, -cameraY, -cameraZ);
```

Replace this code with an implementation of the computation shown in Figure 3.13. This will allow you to position the camera by specifying a camera position and three orientation axes. You will find it necessary to store the vectors U,V,N described in Section 3.7. Then, experiment with different camera viewpoints, and observe the resulting appearance of the rendered cube.

- 4.3 (*PROJECT*) Modify Program 4.4 to include a second “planet,” which is your custom 3D shape from Exercise 4.1. Make sure that your new “planet” is in a different orbit than the existing planet, so that they don’t collide.
- 4.4 (*PROJECT*) Modify Program 4.4 so that the “view” matrix is constructed using the “look-at” function from Figure 3.21. Then experiment with setting the “look-at” parameters to various locations, such as looking at the sun (in which case the scene should appear normal), looking at the planet, or looking at the moon.
- 4.5 (*RESEARCH*) Propose a practical use for `glCullFace(GL_FRONT_AND_BACK)`.

References

- [BL16] Blender, The Blender Foundation, accessed July 2016, <https://www.blender.org/>.
- [HT16] J. Hastings-Trew, *JHT’s Planetary Pixel Emporium*, accessed July 2016, <http://planetpixelemporium.com/>.
- [MA16] Maya, AutoDesk, Inc., accessed July 2016, <http://www.autodesk.com/products/maya/overview>.
- [NA16] NASA 3D Resources, accessed July 2016, <http://nasa3d.arc.nasa.gov/>.
- [OL16] Legacy OpenGL, accessed July 2016, https://www.opengl.org/wiki/Legacy_OpenGL.
- [SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

TEXTURE MAPPING

5.1	<i>Loading Texture Image Files</i>	104
5.2	<i>Texture Coordinates</i>	106
5.3	<i>Creating a Texture Object</i>	108
5.4	<i>Constructing Texture Coordinates</i>	108
5.5	<i>Loading Texture Coordinates into Buffers</i>	110
5.6	<i>Using the Texture in a Shader: Sampler Variables and Texture Units</i>	110
5.7	<i>Texture Mapping: Example Program</i>	111
5.8	<i>Mipmapping</i>	114
5.9	<i>Anisotropic Filtering</i>	120
5.10	<i>Wrapping and Tiling</i>	121
5.11	<i>Perspective Distortion</i>	123
5.12	<i>Loading Texture Image Files Using Java AWT Classes</i>	124
	<i>Supplemental Notes</i>	126



Texture mapping is the technique of overlaying an image across a rasterized model surface. It is one of the most fundamental and important ways of adding realism to a rendered scene.

Texture mapping is so important that there is hardware support for it, allowing for very high performance resulting in real-time photorealism. *Texture Units* are hardware components designed specifically for texturing, and modern graphics cards typically come with several texture units included.

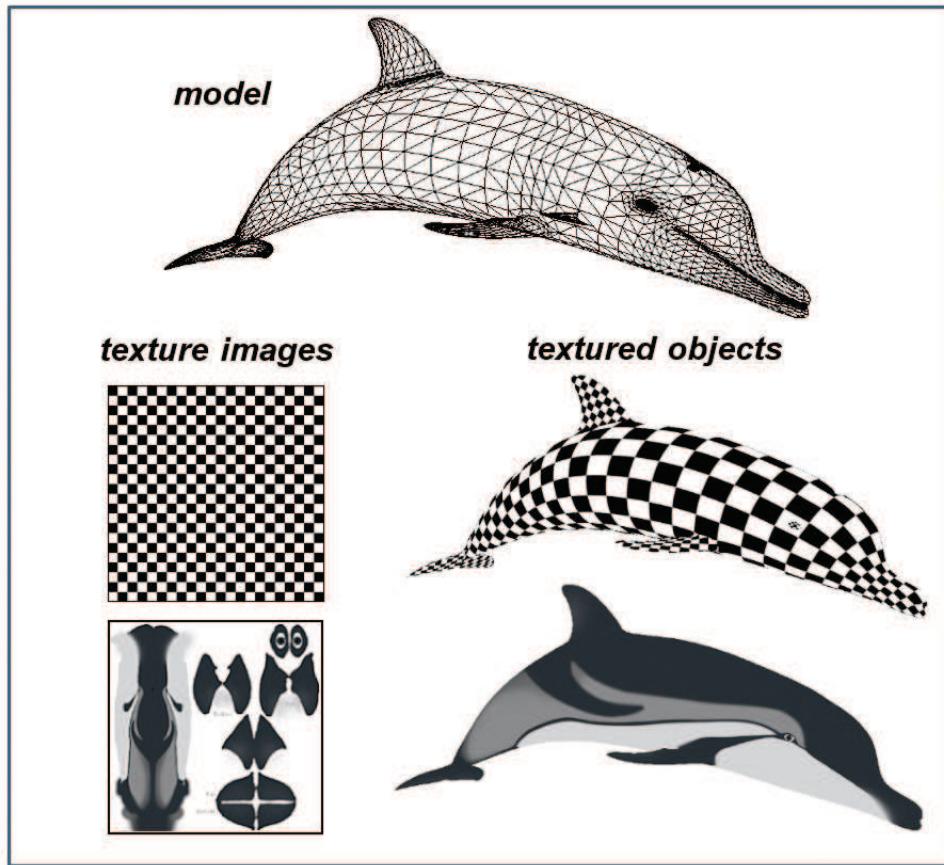


Figure 5.1

Texturing a dolphin model with two different images [JT16].

5.1 LOADING TEXTURE IMAGE FILES

There are a number of datasets and mechanisms that need to be coordinated to accomplish texture mapping efficiently in JOGL / GLSL:

- a *texture object* to hold the texture image (in this chapter we consider only 2D images)
- a special uniform *sampler* variable so that the vertex shader can access the texture
- a buffer to hold the *texture coordinates*
- a vertex attribute for passing the texture coordinates down the pipeline

- a *texture unit* on the graphics card

A texture image can be a picture of anything. It can be a picture of something man-made or occurring in nature, such as cloth, grass, or a planetary surface; or, it could be a geometric pattern, such as the checkerboard in Figure 5.1. In video-games and animated movies, texture images are commonly used to paint faces and clothing on characters, or skin on creatures such as on the dolphin in Figure 5.1.

Images are typically stored in image files, such as .jpg, .png, .gif, or .tiff. In order to make a texture image available to shaders in the OpenGL pipeline, we need to extract the colors from the image and put them into an OpenGL *texture object* (a built-in OpenGL structure for holding a texture image).

Java has some useful image file tools in its `imageio` and `awt` packages that can be used to read texture images. The steps are: (a) read the image data into a `ByteBuffer`, using the JOGL buffer tools we saw in Chapter 4, (b) use `glGenTextures()` to instantiate a texture object and assign it an integer ID, (c) call `glBindTexture()` to make the newly created texture object active, (d) load the previously read-in image data into the texture object with the `glTexImage2D()` command, and (e) adjust the texture settings using the `glTexParameter()` function. The result is an integer pointer to the now available OpenGL texture object. We will walk through these steps at the end of this chapter, for completeness.

However, JOGL includes its own tools for working with textures that make it considerably simpler to load a texture image file into an OpenGL texture object. Those tools are found in the JOGL classes `Texture`, `TextureIO`, and `TextureData`.

Texturing an object starts by declaring a variable of type `Texture`. This is a JOGL class; a JOGL `Texture` object serves as a wrapper for an OpenGL texture object. Next, we call `newTexture()`—a static method in the `TextureIO` class—to actually generate the texture object. The `newTexture()` function accepts an image file name as one of its parameters (several standard image file types are supported, including the four mentioned above). These steps are implemented in the following function:

```
public Texture loadTexture(String textureFileName)
{ Texture tex = null;
  try { tex = TextureIO.newTexture(new File(textureFileName), false); }
  catch (Exception e) { e.printStackTrace(); }
  return tex;
}
```

The Java/JOGL application calls the above `loadTexture()` function to create the JOGL texture object, and extracts the OpenGL texture object from it using `getTextureObject()`. For example:

```
Texture joglTexture = loadTexture("image.jpg");
int openglTexture = joglTexture.getTextureObject();
```

where `image.jpg` is a texture image file.

5.2 TEXTURE COORDINATES

Now that we have a means for loading a texture image into OpenGL, we need to specify how we want the texture to be applied to the rendered surface of an object. We do this by specifying *texture coordinates* for each vertex in our model.

Texture coordinates are references to the pixels in a (usually 2D) texture image. Pixels in a texture image are referred to as *texels*, in order to differentiate them from the pixels being rendered on the screen. Texture coordinates are used to map points on a 3D model to locations in a texture. Each point on the surface of the model has, in addition to (x,y,z) coordinates that place it in 3D space, texture coordinates (s,t) that specify which texel in the texture image provides its color. Thus, the surface of the object is “painted” by the texture image. The orientation of a texture across the surface of an object is determined by the assignment of texture coordinates to object vertices.

In order to use texture mapping, *it is necessary to provide texture coordinates for every vertex* in the object to be textured. OpenGL will use these texture coordinates to determine the color of each rasterized pixel in the model, by looking up the color stored at the referenced texel in the image. In order to ensure that every pixel in your rendered model is painted with an appropriate texel from the texture image, *the texture coordinates are put into a vertex attribute so that they are also interpolated by the rasterizer*. In that way the texture image is interpolated, or filled in, along with the model vertices.

For each set of vertex coordinates (x,y,z) passing through the vertex shader, there will be an accompanying set of texture coordinates (s,t) . We will thus set up two buffers, one for the vertices (with three components x , y , and z in each entry) and one for the corresponding texture coordinates (with two components s and t in each entry). Each vertex shader invocation thus receives one vertex, now comprised of both its spatial coordinates and its corresponding texture coordinates.

Texture coordinates are most often 2D (OpenGL does support some other dimensionalities but we won't cover them in this chapter). It is assumed that the image is rectangular, with location $(0,0)$ at the lower left and $(1,1)$ at the upper right.¹ Texture coordinates, then, should ideally have values in the range $(0,1)$.

Consider the example shown in Figure 5.2. The cube model, recall, is constructed of triangles. The four corners of one side of the cube are highlighted, but remember that it takes two triangles to specify each square side. The texture coordinates for each of the six vertices that specify this one cube side are listed alongside the four corners, with the corners at the upper left and lower right each comprised of a pair of vertices. A texture image is also shown. The texture coordinates (indexed by s and t) have mapped portions of the image (the texels) onto the rasterized pixels of the front face of the model. Note that all of the intervening pixels in between the vertices have been painted with the intervening texels in the image. This is achieved because the texture coordinates are sent to the fragment shader in a *vertex attribute*, and thus are *also interpolated* just like the vertices themselves.

In this example, we purposely specified texture coordinates that result in an oddly-painted surface, for purposes of illustration. If you look closely, you can also see that the image appears slightly stretched—that is because the aspect ratio of the texture image doesn't match the aspect ratio of the cube face relative to the given texture coordinates.

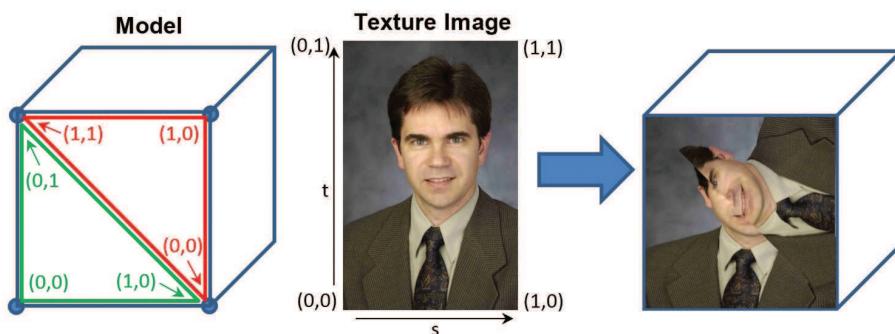


Figure 5.2
Texture coordinates.

¹ This is the orientation that OpenGL texture objects assume. However, this is different from the orientation of an image stored in many standard image file formats, in which the origin is at the upper left. Re-orienting the image by flipping it vertically so that it corresponds to OpenGL's expected format is one of the operations performed by the JOGL `newTexture()` call that we made from the `loadTexture()` function.

For simple models like cubes or pyramids, selecting texture coordinates is relatively easy. But for more complex curved models with lots of triangles, it isn't practical to determine them by hand. In the case of curved geometric shapes, such as a sphere or torus, texture coordinates can be computed algorithmically or mathematically. In the case of a model built with a modeling tool such as Maya [MA16] or Blender [BL16], such tools offer "UV-mapping" (outside of the scope of this book) to make this task easier.

Let us return to rendering our pyramid, only this time texturing it with an image of bricks. We will need to specify: (a) a texture object to hold the texture image, (b) texture coordinates for the model vertices, (c) a buffer for holding the texture coordinates, (d) vertex attributes so that the vertex shader can receive and forward the texture coordinates through the pipeline, (e) a texture unit on the graphics card for holding the texture object, and (f) a *uniform sampler* variable for accessing the texture unit in GLSL, which we will see shortly. These are each described in the next sections.

5.3 CREATING A TEXTURE OBJECT



Suppose the adjacent image is stored in a file named "brick1.jpg" [LU16].

As shown previously, we can load this image by calling our `loadTexture()` function, as follows:

```
Texture joglBrickTexture = loadTexture("brick1.jpg");
int brickTexture = joglBrickTexture.getTextureObject();
```

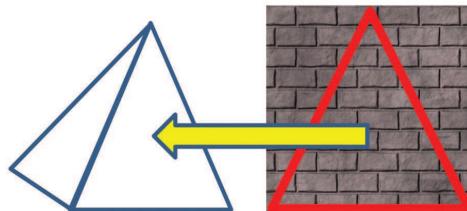
Recall that texture objects are identified by integer IDs, so `brickTexture` is of type `int`.

5.4 CONSTRUCTING TEXTURE COORDINATES

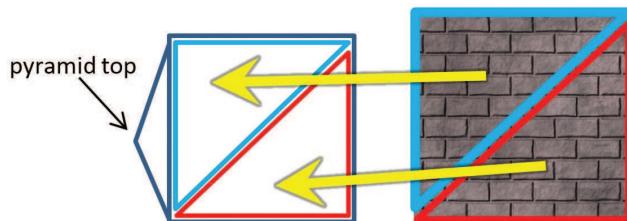
Our pyramid has four triangular sides and a square on the bottom. Although geometrically this only requires five (5) points, we have been rendering it with triangles. This requires four triangles for the sides, and two triangles for the square bottom, for a total of six triangles. Each triangle has three vertices, for a total of $6 \times 3 = 18$ vertices that must be specified in the model.

We already listed the pyramid's geometric vertices in Program 4.3 in the float array `pyramid_positions[]`. There are many ways that we could orient our texture coordinates so as to draw our bricks onto the pyramid. One simple (albeit imperfect)

way would be to make the top center of the image correspond to the peak of the pyramid, as follows:



We can do this for all four of the triangle sides. We also need to paint the bottom square of the pyramid, which is comprised of two triangles. A simple and reasonable approach would be to texture it with the entire area from the picture (the pyramid has been tipped back and is sitting on its side):



The corresponding set of vertex and texture coordinates using this very simple strategy, for the first eight of the pyramid vertices from Program 4.3, is shown below, in Figure 5.3:

<i>vertices</i>	<i>texture coordinates</i>	
(-1.0, -1.0, 1.0)	(0, 0)	// front face
(1.0, -1.0, 1.0)	(1, 0)	
(0, 1.0, 0)	(.5, 1)	
(1.0, -1.0, 1.0)	(0, 0)	// right face
(1.0, -1.0, -1.0)	(1, 0)	
(0, 1.0, 0)	(.5, 1)	
(1.0, -1.0, -1.0)	(0, 0)	// back face
(-1.0, -1.0, -1.0)	(1, 0)	
(0, 1.0, 0)	(.5, 1)	
etc.		

Figure 5.3

Texture coordinates for the pyramid (partial list).

5.5 LOADING TEXTURE COORDINATES INTO BUFFERS

We can load the texture coordinates into a VBO in a similar manner as we saw previously for loading the vertices. In `setupVertices()`, we add the following declaration of the texture coordinate values:

```
float[ ] pyr_texture_coordinates =
{ 0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,  0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f, // top and right faces
  0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,  0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f, // back and left faces
  0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f,  1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f }; // base triangles
```

Then, after the creation of at least two VBOs (one for the vertices, and one for the texture coordinates), we add the following lines of code to load the texture coordinates into VBO #1:

```
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
FloatBuffer pTexBuf = Buffers.newDirectFloatBuffer(pyr_texture_coordinates);
gl glBufferData(GL_ARRAY_BUFFER, pTexBuf.limit()*4, pTexBuf, GL_STATIC_DRAW);
```

5.6 USING THE TEXTURE IN A SHADER: SAMPLER VARIABLES AND TEXTURE UNITS

To maximize performance, we will want to perform the texturing in hardware. This means that our fragment shader will need a way of accessing the texture object that we created in the Java/JOGL application. The mechanism for doing this is via a special GLSL tool called a *uniform sampler variable*. This is a variable designed for instructing a texture unit on the graphics card as to which texel to extract or “sample” from a loaded texture object.

Declaring a sampler variable in the shader is easy—just add it to your set of uniforms:

```
layout (binding=0) uniform sampler2D samp;
```

Ours is named “`samp`”. The “`layout (binding=0)`” portion of the declaration specifies that this sampler is to be associated with texture unit 0.

A texture unit (and associated sampler) can be used to sample whichever texture object you wish, and that can change at runtime. Your `display()` function will

need to specify which texture object you want the texture unit to sample for the current frame. So each time you draw an object, you will need to activate a texture unit and bind it to a particular texture object, as follows:

```
gl.glActiveTexture(GL_TEXTURE0);
gl glBindTexture(GL_TEXTURE_2D, brickTexture);
```

The number of available texture units depends on how many are provided on the graphics card. According to the OpenGL API documentation, OpenGL version 4.5 requires that this be at least 16 per shader stage, and at least 80 total units across all stages [OP16]. In this example, we have made the 0th texture unit active by specifying `GL_TEXTURE0` in the `glActiveTexture()` call.

To actually perform the texturing, we will need to modify how our fragment shader outputs colors. Previously, our fragment shader either output a constant color, or it obtained colors from a vertex attribute. This time instead, we need to use the interpolated texture coordinates received from the vertex shader (through the rasterizer) to sample the texture object, by calling the `texture()` function as follows:

```
in vec2 tc;           // texture coordinates
...
color = texture(samp, tc);
```

5.7 TEXTURE MAPPING: EXAMPLE PROGRAM

Program 5.1 combines the previous steps into a single program. The result, showing the pyramid textured with the brick image, appears in Figure 5.4. Two rotations (not shown in the code listing) were added to the pyramid's model matrix to expose the underside of the pyramid.

It is now a simple matter to replace the brick texture image with other texture images, as desired, simply by

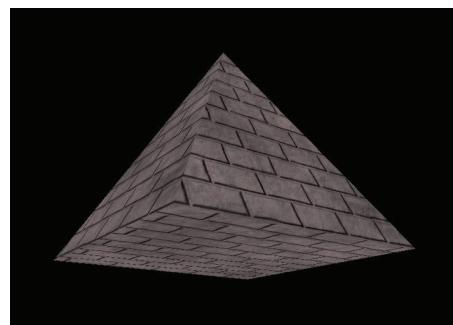


Figure 5.4
Pyramid texture mapped with brick image.

changing the filename in the `loadTexture()` call. For example, if we replace “brick1.jpg” with the image file “ice.jpg” [LU16], we get the result shown in Figure 5.5.

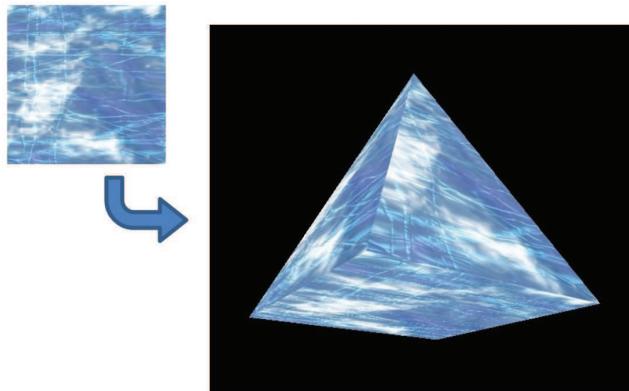


Figure 5.5
Pyramid texture mapped with “ice” image.

Program 5.1 Pyramid with brick texture.

JAVA/JOGL Application

```
// the following additional imports will be necessary for the loadTexture:
import java.io.*;
import java.nio.*;
import com.jogamp.opengl.util.texture.*;
import com.jogamp.opengl.util.texture.TextureIO.*;

public class Code extends JFrame implements GLEventListener
{ // previous declarations and constructor code applies. We just need to add declarations for the texture:
  ...
  private int brickTexture;
  ...

  public Code()
  { // unchanged ...
  }

  public void display(GLAutoDrawable drawable)
  { GL4 gl = (GL4) GLContext.getCurrentGL();
    ...
    // setup of background color, depth buffer, VIEW and PROJ matrices unchanged
    ...
  }
}
```

```

// this time we are drawing only the pyramid.
// setup of M and MV matrices is unchanged.
...
// activate buffer #0, which contains the vertices
gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

// activate buffer #1, which contains the texture coordinates
gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
gl.glVertexAttribPointer(1, 2, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(1);

// activate texture unit #0 and bind it to the brick texture object
gl.glActiveTexture(GL_TEXTURE0);
gl glBindTexture(GL_TEXTURE_2D, brickTexture);

gl glEnable(GL_DEPTH_TEST);
gl glDepthFunc(GL_EQUAL);

gl.glDrawArrays(GL_TRIANGLES, 0, 18);
}

public Texture loadTexture(String textureFileName)
{
    Texture tex = null;
    try { tex = TextureIO.newTexture(new File(textureFileName), false); }
    catch (Exception e) { e.printStackTrace(); }
    return tex;
}

public void init(GLAutoDrawable drawable)
{
    // setup of rendering program, camera and object location unchanged
    ...
    joglBrickTexture = loadTexture("brick1.jpg");
    brickTexture = joglBrickTexture.getTextureObject();
}

private void setupVertices()
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    float[] pyramid_positions = { /* data as listed previously in Program 4.2 */ };
    float[] pyr_texture_coordinates =
    {
        0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,    0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,
        0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,    0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f,    1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f
    };
    // ... generate the VAO as before, and at least two VBOs, then load the two buffers as follows:
    gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
}

```

```

FloatBuffer pyrBuf = Buffers.newDirectFloatBuffer(pyramid_positions);
gl.glBufferData(GL_ARRAY_BUFFER, pyrBuf.limit()*4, pyrBuf, GL_STATIC_DRAW);

gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
FloatBuffer pTexBuf = Buffers.newDirectFloatBuffer(pyr_texture_coordinates);
gl.glBufferData(GL_ARRAY_BUFFER, pTexBuf.limit()*4, pTexBuf, GL_STATIC_DRAW);
}
... // remainder of the class definition and utility functions here
}

```

Vertex shader

```

#version 430
layout (location=0) in vec3 pos;
layout (location=1) in vec2 texCoord;
out vec2 tc;           // texture coordinate output to rasterizer for interpolation
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler2D samp;    // not used in vertex shader

void main(void)
{
    gl_Position = proj_matrix * mv_matrix * vec4(pos,1.0);
    tc = texCoord;
}

```

Fragment shader

```

#version 430
in vec2 tc;           // interpolated incoming texture coordinate
out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler2D samp;

void main(void)
{
    color = texture(samp, tc);
}

```

5.8 MIPMAPPING

Texture mapping commonly produces a variety of undesirable artifacts in the rendered image. This is because the resolution or aspect ratio of the texture image rarely matches that of the region in the scene being textured.

A very common artifact occurs when the image resolution is *less than* that of the region being drawn. In this case, the image would need to be stretched to cover

the region, becoming blurry (and possibly distorted). This can sometimes be combated, depending on the nature of the texture, by assigning the texture coordinates differently so that applying the texture requires less stretching. Another solution is to use a higher resolution texture image.

The reverse situation is when the resolution of the image texture is *greater than* that of the region being drawn. It is probably not at all obvious why this would pose a problem, but it does! In this case, noticeable *aliasing* artifacts can occur, giving rise to strange-looking false patterns, or “shimmering” effects in moving objects.

Aliasing is caused by *sampling errors*. It is most often associated with signal processing, where an inadequately sampled signal appears to have different properties (such as wavelength) than it actually does when it is reconstructed. An example is shown in Figure 5.6. The original waveform is shown in red. The yellow dots along the waveform represent the samples. If they are used to reconstruct the wave, and if there aren’t enough of them, they can define a different wave (shown in blue).

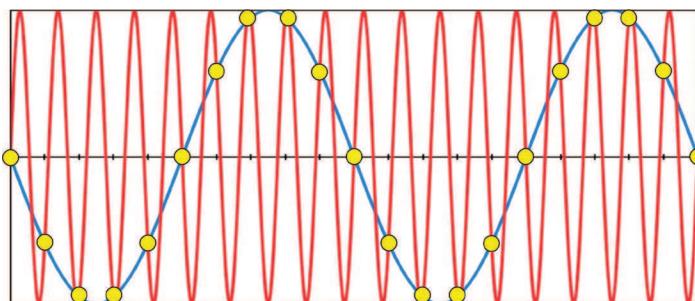


Figure 5.6
Aliasing due to inadequate sampling.

Similarly, in texture-mapping, when a high-resolution (and highly detailed) image is sparsely sampled (such as when using a uniform sampler variable), the colors retrieved will be inadequate to reflect the actual detail in the image, and may instead seem random. If the texture image has a repeated pattern, aliasing can result in a *different* pattern being produced than the one in the original image. If the object being textured is moving, rounding errors in looking up the texels can result in constant changes in the sampled pixel at a given texture coordinate, resulting in a sparkling effect across the surface of the object being drawn.

Figure 5.7 shows a tilted close-up rendering of the top of a cube which has been textured by a large, high-resolution image of a checkerboard.

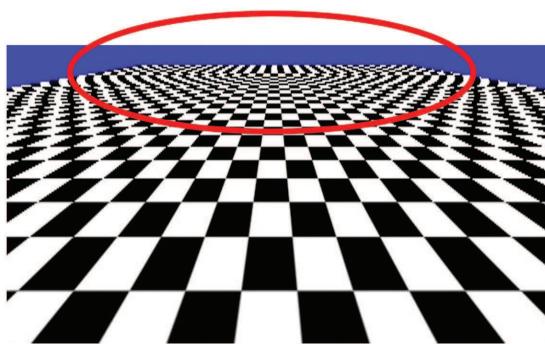


Figure 5.7
Aliasing in a texture map.

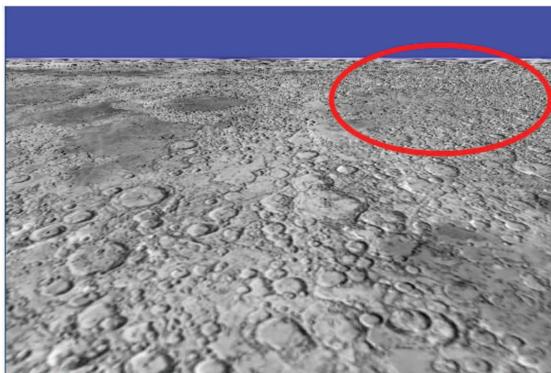


Figure 5.8
“Sparkling” in a texture map.

error artifacts can be largely corrected by a technique called *mipmapping*, in which different versions of the texture image are created at various resolutions. The texture image is used that most closely matches the resolution at the point being textured. Even better, colors can be averaged between the images closest in resolution to that of the region being textured. Results of applying mipmapping to the images in Figure 5.7 and Figure 5.8 are shown in Figure 5.9.

Mipmapping works by a clever mechanism for storing a series of successively lower-resolution copies of the same image, in a texture image $\frac{1}{3}$ larger than the original image. This is achieved by storing the R, G, and B components of the image separately in three (out of four) quadrants of the texture image, then repeating the process in the fourth quadrant for the same image at $\frac{1}{4}$ the original resolution. This subdividing repeats until the remaining quadrant is too small to contain any

Aliasing is evident near the top of the image, where the undersampling of the checkerboard has produced a “striped” effect. Although we can’t show it here in a still image, if this were an animated scene, the patterns would likely seem to undulate between various incorrect patterns such as this one.

Another example appears in Figure 5.8, in which the cube has been textured with an image of the surface of the moon [HT16]. At first glance, this image appears sharp and full of detail. However, some of the detail at the upper right of the image is false, and causes “sparkling” as the cube object (or the camera) moves. (Unfortunately, we can’t show the sparkling effect clearly in a still image.)

These and similar sampling

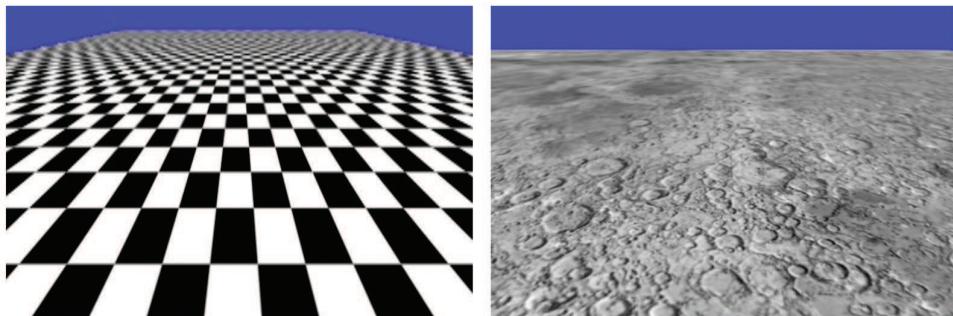


Figure 5.9
Mipmapped results

useful image data. An example image and the resulting mipmap is shown below in Figure 5.10.

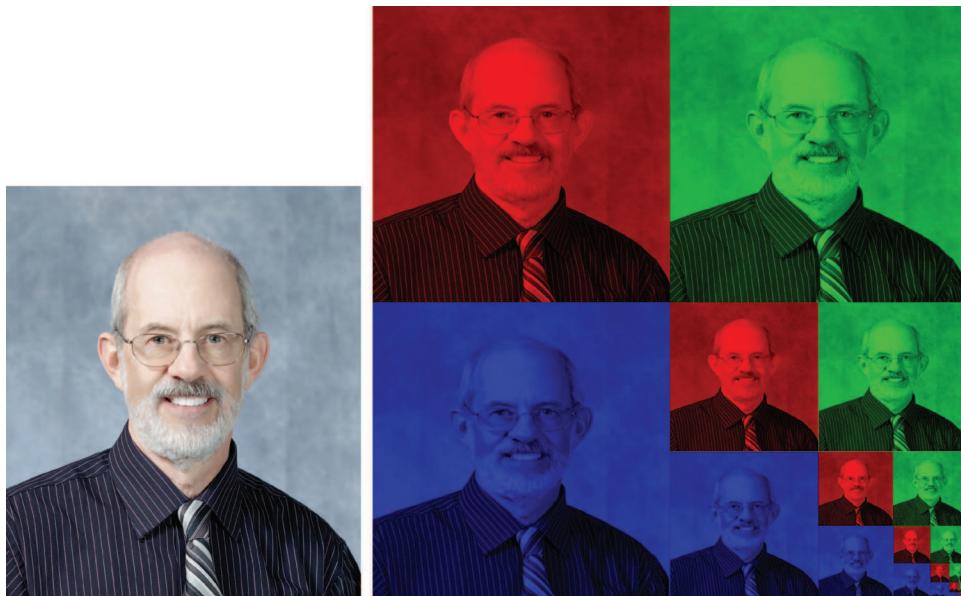


Figure 5.10
Mipmapping an image.

This method of stuffing several images into a small space (well, just a bit bigger than the space needed to store the original image) is how mipmapping got its name. MIP stands for *Multum In Parvo* [WI83], which is Latin for “much in a small space.”

When actually texturing an object, the mipmap can be sampled in several ways. In OpenGL, the manner in which the mipmap is sampled can be chosen by setting the `GL_TEXTURE_MIN_FILTER` parameter to the desired *minification* technique, which is one of the following:

- `GL_NEAREST_MIPMAP_NEAREST`
chooses the mipmap with the resolution most similar to that of the region of pixels being textured. It then obtains the nearest texel to the desired texture coordinates.
- `GL_LINEAR_MIPMAP_NEAREST`
chooses the mipmap with the resolution most similar to that of the region of pixels being textured. It then interpolates the four texels nearest to the texture coordinates. This is called “linear filtering.”
- `GL_NEAREST_MIPMAP_LINEAR`
chooses the two mipmaps with resolutions nearest to that of the region of pixels being textured. It then obtains the nearest texel to the texture coordinates from each mipmap, and interpolates them. This is called “bilinear filtering.”
- `GL_LINEAR_MIPMAP_LINEAR`
chooses the two mipmaps with resolutions nearest to that of the region of pixels being textured. It then interpolates the four nearest texels in each mipmap, and interpolates those two results. This is called “trilinear filtering” and is illustrated below in Figure 5.11.

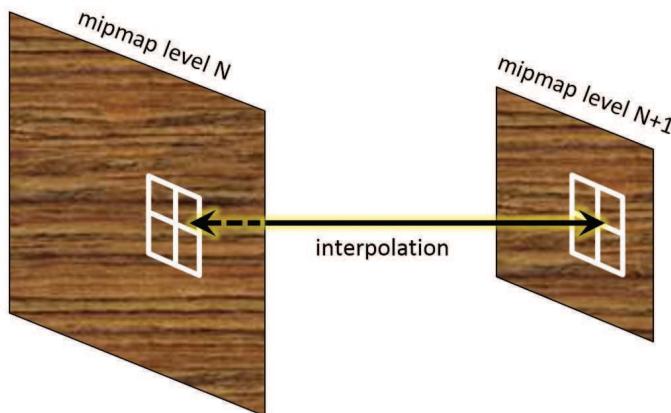


Figure 5.11

Trilinear filtering.

Trilinear filtering is usually preferable, as lower levels of blending often produce artifacts, such as visible separations between mipmap levels. Figure 5.12 shows a close-up of the checkerboard using mipmaping with only linear filtering enabled. Note the circled artifacts, where the vertical lines suddenly change from thick to thin at a mipmap boundary. By contrast, the example in Figure 5.9 used trilinear filtering.

Mipmapping is richly supported in OpenGL. There are mechanisms provided for building your own mipmap levels, or having OpenGL build them for you. In most cases, the mipmaps built automatically by OpenGL are sufficient. The following lines of code added to Program 5.1 in `init()`, immediately after the `loadTexture()` and `getTextureObject()` function calls, cause OpenGL to generate the mipmaps:

```
gl glBindTexture(GL_TEXTURE_2D, brickTexture);
gl TexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
gl GenerateMipmap(GL_TEXTURE_2D);
```

The brick texture is made active with the `glBindTexture()` call, and then the `glTexParameter()` function call enables one of the minification factors listed above, such as `GL_LINEAR_MIPMAP_LINEAR` shown in the above call, which enables trilinear filtering.

Once the mipmap is built, the filtering option can be changed (although this is rarely necessary) by calling `glTexParameter()` again, such as in the display function. Mipmapping can even be disabled by selecting `GL_NEAREST` or `GL_LINEAR`.

For critical applications, it is possible to build the mipmaps yourself, using whatever is your preferred image editing software. They can then be added as mipmap levels when creating the texture object by repeatedly calling OpenGL's `glTexImage2D()` function, or JOGL's `updateSubImage()` function, for each mipmap level. This approach is outside the scope of this book.

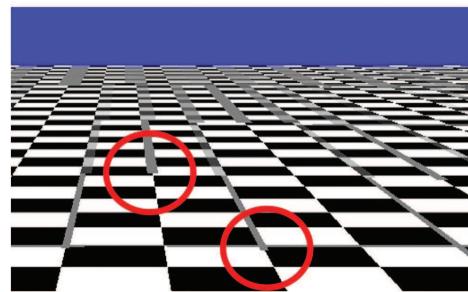


Figure 5.12
Linear filtering artifacts.

5.9 ANISOTROPIC FILTERING

Mipmapped textures can sometimes appear more blurry than non-mipmapped textures, especially when the textured object is rendered at a heavily tilted viewing angle. We saw an example of this back in Figure 5.9, where reducing artifacts with mipmapping led to reduced detail (comparing with Figure 5.8).

This loss of detail occurs because when an object is tilted, its primitives appear smaller along one axis (i.e., width vs. height) than the other. When OpenGL textures a primitive, it selects the mipmap appropriate for the *smaller* of the two axes (to avoid “sparkling” artifacts). In Figure 5.9 the surface is tilted heavily away from the viewer, so each rendered primitive will utilize the mipmap appropriate for its reduced *height*, which is likely to have a resolution lower than appropriate for its *width*.

One way of restoring some of this lost detail is to use *anisotropic filtering* (AF). Whereas standard mipmapping samples a texture image at a variety of square resolutions (e.g., 256×256 , 128×128 , etc.), AF samples the textures at a number of rectangular resolutions as well, such as 256×128 , 64×128 , and so on. This enables viewing at various angles while retaining as much detail in the texture as possible.

Anisotropic filtering is more computationally expensive than standard mipmapping, and is not a required part of OpenGL. However, most graphics cards support AF (this is referred to as an *OpenGL extension*), and OpenGL does provide both a way of querying the card to see if it supports AF, and a way of accessing AF if it does. The code is added immediately after generating the mipmap:

```
...
// if mipmapping
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
gl.glGenerateMipmap(GL_TEXTURE_2D);

// if also anisotropic filtering
if (gl.isExtensionAvailable("GL_EXT_texture_filter_anisotropic"))
{
    float max[ ] = new float[1];
    gl.glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, max, 0);
    gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, max[0]);
}
```

The call to `gl.isExtensionAvailable()` tests whether the graphics card supports AF. If it does, we set it to the maximum degree of sampling supported, a value retrieved using `glGetFloatv()` as shown. It is then applied to the active texture object using `glTexParameterf()`. The result is shown in Figure 5.13. Note that much of the lost detail from Figure 5.8 has been restored, while still removing the sparkling artifacts.

5.10 WRAPPING AND TILING

So far we have assumed that texture coordinates all fall in the range (0,1). However, OpenGL actually supports texture coordinates of any value. There are several options for specifying what happens when texture coordinates fall outside the range (0,1). The desired behavior is set using `glTexParameter()`, and the options are as follows:

- `GL_REPEAT`: The integer portion of the texture coordinates are ignored, generating a repeating or “tiling” pattern. This is the default behavior.
- `GL_MIRRORED_REPEAT`: The integer portion is ignored, except that the coordinates are reversed when the integer portion is odd, so the repeating pattern alternates between normal and mirrored.
- `GL_CLAMP_TO_EDGE`: Coordinates less than 0 and greater than 1 are set to 0 and 1, respectively.
- `GL_CLAMP_TO_BORDER`: Texels outside of (0,1) will be assigned some specified border color.

For example, consider a pyramid in which the texture coordinates have been defined in the range (0,4) rather than the range (0,1). The default behavior (`GL_REPEAT`), using the texture image shown previously in Figure 5.2, would result in the texture repeating four times across the surface (sometimes called “tiling”), as shown below in Figure 5.14:

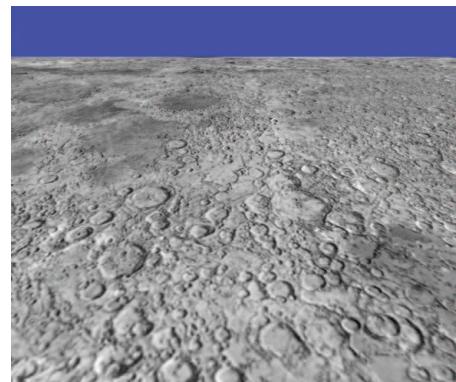


Figure 5.13
Anisotropic filtering.



Figure 5.14
Texture coordinate wrapping with `GL_REPEAT`.

To make the tiles' appearance alternate between normal and mirrored, we can specify the following:

```
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Specifying that values less than 0 and greater than 1 be set to 0 and 1 respectively, can be done by replacing `GL_MIRRORED_REPEAT` with `GL_CLAMP_TO_EDGE`.

Specifying that values less than 0 and greater than 1 result in a “border” color can be done as follows:

```
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float[] redColor = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
gl.glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, redColor, 0);
```

The effect of each of these options (mirrored repeat, clamp to edge, and clamp to border), with texture coordinates ranging from -1 to +2, are shown respectively (left to right) in Figure 5.15.



Figure 5.15
Textured pyramid with various wrapping options.

In the center example (clamp-to-edge), the pixels along the edges of the texture image are replicated outward. Note that as a side effect, the lower-left and lower-right regions of the pyramid faces obtain their color from the lower-left and lower-right pixels of the texture image, respectively.

5.11 PERSPECTIVE DISTORTION

We have seen that as texture coordinates are passed from the vertex shader to the fragment shader, they are interpolated as they pass through the rasterizer. We have also seen that this is the result of the automatic linear interpolation that is always performed on vertex attributes.

However, in the case of texture coordinates, linear interpolation can lead to noticeable distortion in a 3D scene with perspective projection.

Consider a rectangle made of two triangles, and textured with a checkerboard image, facing the camera. As the rectangle is rotated around the X axis, the top part of the rectangle tilts away from the camera, while the lower part of the rectangle swings closer to the camera. Thus, we would expect the squares at the top to become smaller, and the squares at the bottom to become larger. However, linear interpolation of the texture coordinates will instead cause the height of *all* squares to be equal. The distortion is exacerbated along the diagonal defining the two triangles that make up the rectangle. The resulting distortion is shown in Figure 5.16.

Fortunately, there are algorithms for correcting perspective distortion, and by default, OpenGL applies a *perspective correction* algorithm [OP14] when sampling texture coordinates. Figure 5.17 shows the same rotating checkerboard, properly rendered by OpenGL.

Although not common, it is possible to disable OpenGL’s perspective correction by adding the keyword “*noperspective*” in the declaration of the vertex attribute containing the texture coordinates. This has to be done in both the vertex and fragment shaders. For example, the vertex attribute in the vertex shader would be declared as follows:

```
noperspective out vec2 texCoord;
```

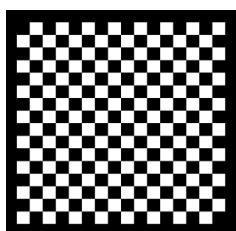


Figure 5.16

Texture perspective distortion.

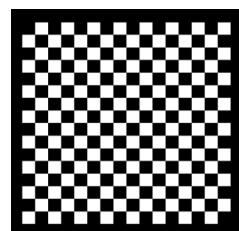
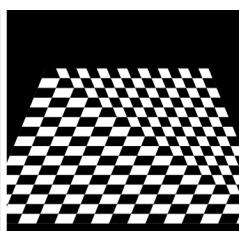


Figure 5.17

OpenGL perspective correction.

The corresponding attribute in the fragment shader would be declared:

```
noperspective in vec2 texCoord;
```

This second syntax was in fact used to produce the distorted checkerboard in Figure 5.16.

5.12 LOADING TEXTURE IMAGE FILES USING JAVA AWT CLASSES

Throughout the rest of this textbook we use the JOGL Texture, TextureIO, and TextureData classes as described earlier in this chapter to load texture image data into OpenGL texture objects. However, it is possible to load texture image file data into OpenGL textures *directly*, using Java AWT classes and some additional OpenGL commands. The process is quite a bit more complicated, so for simplicity and clarity we will use the JOGL classes in this book whenever possible. However, it is useful to understand the process (and the particular commands) one could use in lieu of the JOGL texture classes. For example, the JOGL texture classes don't support 3D textures, so as we will see later, building an OpenGL 3D texture object will require doing many of the steps ourselves in Java.

Building a `loadTexture()` function analogous to the one in Program 5.1, without using JOGL texture classes, is shown in Program 5.2. It starts by calling two utility functions (also shown). The first one is `getBufferedImage()`, which reads the specified image file, assumed to be in a recognized format such as `.jpg` or `.png`, and returns a Java `BufferedImage` containing the image file data. The second utility function, `getRGBAPixelData()`, extracts the RGBA pixel colors from the specified `BufferedImage` and returns them in a byte array organized in the form expected by OpenGL.

The `loadTexture()` function then continues by copying the byte array returned from `getRGBAPixelData()` into a Java `ByteBuffer`, using the JOGL `Buffers.newDirectByteBuffer()` method described in Chapter 4. It then creates the texture object, in a manner similar to the steps we used for creating VBOs. Textures, like buffers, are given integer IDs by calling `glGenTextures()`. Here, the variable `textureID` is used to hold the ID of a generated texture object. Next, the texture object is made active by calling `glBindTexture()`, and then we load the previously read-in image data into the active texture object by using the `glTexImage2D()` command. Note the first parameter on this call specifies the type of texture object—in this case `GL_TEXTURE_2D`.

(later we will use this command to create other types of OpenGL textures, such as texture cube maps in Chapter 9 and 3D textures in Chapter 14). The next command, `glTexParameteri()`, can be used to adjust some of the texture settings, such as building mipmaps. When `loadTexture()` finishes, it returns the integer ID for the now available OpenGL texture object, rather than the earlier version that returned a JOGL Texture object.

The call to `loadTexture()` in `init()` would also change slightly. Since the version of `loadTexture()` in Program 5.2 returns an int (rather than a JOGL Texture), there is no need to call `getTextureObject()`. Instead, the following single call to `loadTexture()` suffices to create the integer pointer to the OpenGL texture object:

```
int brickTexture = loadTexture("brick1.jpg");
```

Program 5.2 Java AWT Routines for Loading Texture Images

```
private int loadTexture(String textureFileName)
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    BufferedImage textureImage = getBufferedImage(textureFileName);
    byte[ ] imgRGBA = getRGBAPixelData(textureImage);
    ByteBuffer rgbaBuffer = Buffers.newDirectByteBuffer(imgRGBA);

    int[ ] textureIDs = new int[1];                                // array to hold generated texture IDs
    gl.glGenTextures(1, textureIDs, 0);
    int textureID = textureIDs[0];                                // ID for the 0th texture object
    gl glBindTexture(GL_TEXTURE_2D, textureID);                  // specifies the active 2D texture
    gl.glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,                      // MIPMAP level, color space
                    textureImage.getWidth(), textureImage.getHeight(), 0 // image size, border (ignored)
                    GL_RGBA, GL_UNSIGNED_BYTE,                            // pixel format and data type
                    rgbaBuffer);                                         // buffer holding texture data
    gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    return textureID;
}

private BufferedImage getBufferedImage(String fileName)
{
    BufferedImage img;
    try { img = ImageIO.read(new File(fileName)); }
    catch (IOException e)
    { System.err.println("Error reading " + fileName + ""); throw new RuntimeException(e); }
    return img;
}

private byte[ ] getRGBAPixelData(BufferedImage img)
{
    byte[ ] imgRGBA;
```

```

int height = img.getHeight(null);
int width = img.getWidth(null);

WritableRaster raster =
    Raster.createInterleavedRaster(DataBuffer.TYPE_BYTE, width, height, 4, null);
ComponentColorModel colorModel = new ComponentColorModel(
    ColorSpace.getInstance(ColorSpace.CS_sRGB),
    new int[ ] { 8, 8, 8, 8 }, true, false,           // bits, has Alpha, isAlphaPremultiplied
    ComponentColorModel.TRANSLUCENT, // transparency
    DataBuffer.TYPE_BYTE);          // data transfer type
BufferedImage newImage = new BufferedImage(colorModel, raster, false, null);

// use an affine transform to "flip" the image to conform to OpenGL orientation.
// In Java the origin is at the upper left. In OpenGL the origin is at the lower left.
AffineTransform gt = new AffineTransform();
gt.translate(0, height);
gt.scale(1, -1d);
Graphics2D g = newImage.createGraphics();
g.transform(gt);
g.drawImage(img, null, null);
g.dispose();
DataBufferByte dataBuf = (DataBufferByte) raster.getDataBuffer();
imgRGBA = dataBuf.getData();
return imgRGBA;
}

```

SUPPLEMENTAL NOTES

Researchers have developed a number of uses for texture units beyond just texturing models in a scene. In later chapters, we will see how texture units can be used for altering the way light reflects off an object, making it appear bumpy. We can also use a texture unit to store “height maps” for generating terrain, and for storing “shadow maps” to efficiently add shadows to our scenes. These uses will be described in subsequent chapters.

Shaders can also *write to textures*, allowing shaders to modify texture images, or even copy part of one texture into some portion of another texture.

Mipmaps and anisotropic filtering are not the only tools for reducing aliasing artifacts in textures. *Full-scene anti-aliasing* (FSAA) and other super-sampling methods, for example, can also improve the appearance of textures in a 3D scene. Although not part of the OpenGL core, they are supported on many graphics cards through OpenGL’s extension mechanism [OE16].

There is an alternative mechanism for configuring and managing textures and samplers. Version 3.3 of OpenGL introduced *sampler objects* (sometimes called “*sampler states*”—not to be confused with sampler variables) that can be used to hold a set of texture settings independent of the actual texture object. Sampler objects are attached to texture units and allow for conveniently and efficiently changing texture settings. The examples shown in this textbook are sufficiently simple that we decided to omit coverage of sampler objects. For interested readers, sampler objects are easy to learn how to use, and there are many excellent online tutorials (such as [GE11]).

The JOGL Texture class makes a number of OpenGL texture-related functions available directly, without extracting the actual OpenGL texture object as we did in this chapter. For example, there are `bind()` and `setTexParameter()` functions that invoke the OpenGL functions `glBindTexture()` and `glSetTexParameter()`. We will explore more of the functionality in the JOGL texture classes later in the book when we study cube maps and 3D textures. An excellent source of information on the JOGL Texture, TextureIO, and TextureData classes is their extensive Javadoc pages.

Exercises

- 5.1** Modify Program 5.1 by adding the “`noperspective`” declaration to the texture coordinate vertex attributes, as described in Section 5.11. Then re-run the program and compare the output with the original. Is any perspective distortion evident?
- 5.2** Using a simple “paint” program (such as Windows “Paint” or GIMP [Gl16]), draw a freehand picture of your own design. Then use your image to texture the pyramid in Program 5.1.
- 5.3 (PROJECT)** Modify Program 4.4 so that the “sun,” “planet,” and “moon” are textured. You may continue to use the shapes already present, and you may use any texture you like. Texture coordinates for the cube are available by searching through some of the posted code examples, or you can build them yourself by hand (although that is a bit tedious).

References

- [BL16] Blender, The Blender Foundation, accessed July 2016, <https://www.blender.org/>.
- [GE11] Geeks3D, “OpenGL Sampler Objects: Control Your Texture Units,” September 8, 2011, accessed July 2016, <http://www.geeks3d.com/20110908/>.
- [GI16] GNU Image Manipulation Program, accessed July 2016, <http://www.gimp.org>.
- [HT16] J. Hastings-Trew, *JHT’s Planetary Pixel Emporium*, accessed July 2016, <http://planetpixelemporium.com/>.
- [LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).
- [MA16] Maya, AutoDesk, Inc., accessed July 2016, <http://www.autodesk.com/products/maya/overview>.
- [OE16] OpenGL Registry, The Khronos Group, accessed July 2016, <https://www.opengl.org/registry/>.
- [OP14] OpenGL Graphics System: A Specification (version 4.4), M. Segal and K. Akeley, March 19, 2014, accessed July 2016, <https://www.opengl.org/registry/doc/glspec44.core.pdf>.
- [OP16] OpenGL 4.5 Reference Pages, accessed July 2016, <https://www.opengl.org/sdk/docs/man/>.
- [TU16] J. Turberville, Studio 522 Productions, Scottsdale, AZ, www.studio522.com (dolphin model developed 2016).
- [WI83] L. Williams, “Pyramidal Parametrics,” *Computer Graphics* 17, no. 3 (July 1983).

3D MODELS

6.1	<i>Procedural Models – Building a Sphere</i>	<i>130</i>
6.2	<i>OpenGL Indexing – Building a Torus</i>	<i>137</i>
6.3	<i>Loading Externally Produced Models</i>	<i>144</i>
	<i>Supplemental Notes</i>	<i>153</i>



So far we have dealt only with very simple 3D objects, such as cubes and pyramids. These objects are so simple that we have been able to explicitly list all of the vertex information in our source code and place it directly into buffers.

However, most interesting 3D scenes include objects that are too complex to continue building them as we have, by hand. In this chapter, we will explore more complex object models, how to build them, and how to load them into our scenes.

3D modeling is itself an extensive field, and our coverage here will necessarily be very limited. We will focus on the following two topics:

- *building models procedurally*
- *loading models produced externally*

While this only scratches the surface of the rich field of 3D modeling, it will give us the capability to include a wide variety of complex and realistically detailed objects in our scenes.

6.1 PROCEDURAL MODELS – BUILDING A SPHERE

Some types of objects, such as spheres, cones, and so forth, have mathematical definitions that lend themselves to algorithmic generation. Consider for example a circle of radius R —coordinates of points around its perimeter are well defined:

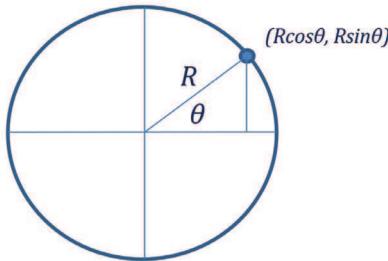


Figure 6.1
Points on a circle.

We can systematically use our knowledge of the geometry of a circle to algorithmically build a sphere model. Our strategy is as follows:

1. Select a *precision* representing a number of circular “horizontal slices” through the sphere. See the left side of Figure 6.2.
2. Subdivide the circumference of each circular slice into some number of points. See the right side of Figure 6.2. More points and horizontal slices produces a more accurate and smoother model of the sphere.

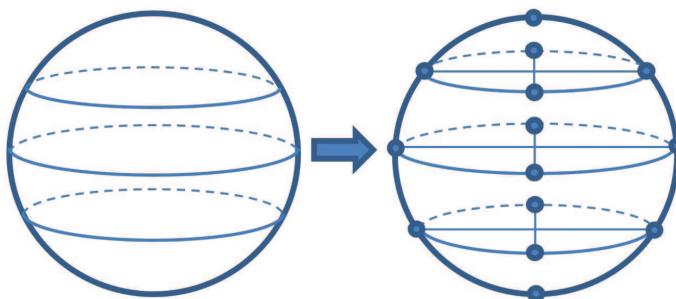


Figure 6.2
Building circle vertices.

3. Group the vertices into triangles. One approach is to step through the vertices, building two triangles at each step. For example, as we move along the row of the five colored vertices on the sphere in Figure 6.3,

for each of those five vertices we build the two triangles shown in the corresponding color (the steps are described in greater detail below).

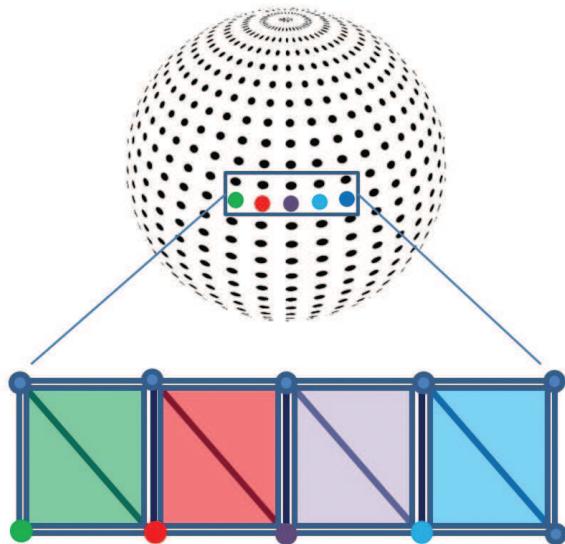


Figure 6.3
Grouping vertices into triangles.

4. Select texture coordinates depending on the nature of our texture images. In the case of a sphere, there exist many topographical texture images, such as the one shown below in Figure 6.4 [VE16] for planet Earth. If we assume this sort of texture image, then by imagining the image “wrapped” around the sphere as shown in Figure 6.5, we can assign texture coordinates to each vertex according to the resulting corresponding positions of the texels in the image.



Figure 6.4
Topographical texture image [VE16].

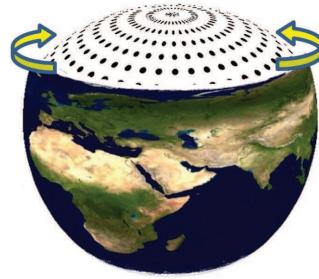


Figure 6.5
Sphere texture coordinates.

5. It is also often desirable to generate *normal vectors*—vectors that are *perpendicular to the model's surface*—for each vertex. We will use them soon, in Chapter 7, for *lighting*.

Determining normal vectors can be tricky, but in the case of a sphere, the vector pointing from the center of the sphere to a vertex happens to conveniently equal the normal vector for that vertex! Figure 6.6 illustrates this property (the center of the sphere is indicated with a “star”):

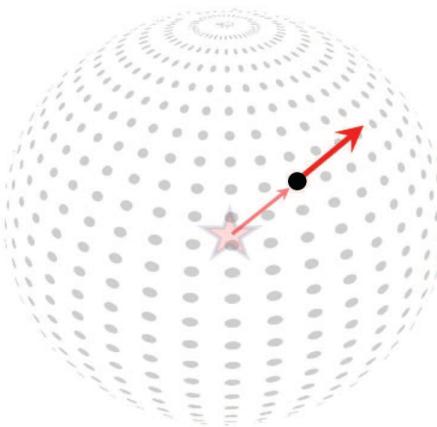
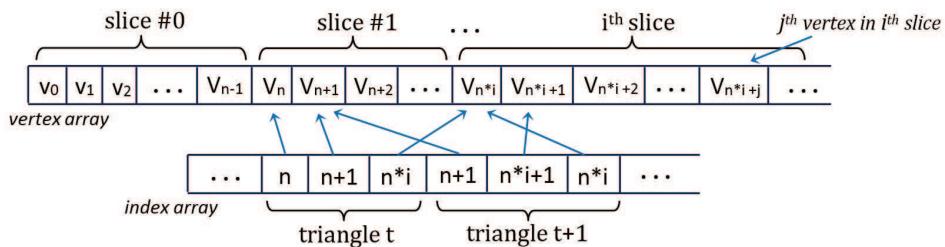


Figure 6.6
Sphere vertex normal vectors.

Some models define triangles using *indices*. Note in Figure 6.3 that each vertex appears in multiple triangles, which would lead to each vertex being specified multiple times. Rather than doing this, we instead store each vertex once, and then specify indices for each corner of a triangle, referencing the desired vertices. Since we will store a vertex's location, texture coordinates, and normal vector, this can facilitate memory savings for large models.

The vertices are stored in a one-dimensional array, starting with the vertices in the bottommost horizontal slice. When using indexing, the associated array of indices includes an entry for each triangle corner. The contents are integer references (specifically, subscripts) into the vertex array. Assuming that each slice contains n vertices, the vertex array would look as shown in Figure 6.7, along with an example portion of the corresponding index array.

**Figure 6.7**

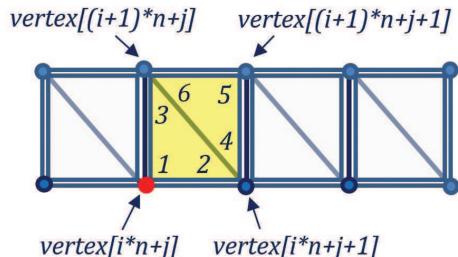
Vertex array and corresponding index array.

We can then traverse the vertices in a circular fashion around each horizontal slice, starting at the bottom of the sphere. As we visit each vertex, we build two triangles forming a square region above and to its right, as shown earlier in Figure 6.3. The processing is thus organized into nested loops, as follows:

```
for each horizontal slice i in the sphere (i ranges from 0 through all the slices in the sphere)
{   for each vertex j in slice i (j ranges from 0 through all the vertices in the slice)
    {   build two triangles out of the neighboring vertices above and to the right of vertex j
    } }
```

For example, consider the “red” vertex from Figure 6.3 (repeated in Figure 6.8). The vertex in question is at the lower left of the yellow triangles shown in Figure 6.8, and given the loops just described, would be indexed by i^*n+j , where i is the slice currently being processed (the outer loop), j is the vertex currently being processed within that slice (the inner loop), and n is the number of vertices per slice. Figure 6.8 shows this vertex (in red) along with its three relevant neighboring vertices, each with formulas showing how they would be indexed.

These four vertices are then used to build the two triangles (shown in yellow) generated for this (red) vertex. The six entries in the index table for these two triangles are indicated in the figure in the order shown by the numbers 1 through 6. Note that entries 3 and 6 both refer to

**Figure 6.8**

Vertices generated for the j^{th} vertex in the i^{th} slice ($n = \text{number of vertices per slice}$).

the same vertex, which is also the case for entries 2 and 4. The two triangles thus defined when we reach the vertex highlighted in red (i.e., `vertex[i*n+j]`) are built out of these six vertices—one with entries marked 1, 2, 3 referencing vertices `vertex[i*n+j]`, `vertex[i*n+j+1]`, and `vertex[(i+1)*n+j]`, and one with entries marked 4, 5, 6 referencing the three vertices `vertex[i*n+j+1]`, `vertex[(i+1)*n+j+1]`, and `vertex[(i+1)*n+j]`.

Program 6.1 shows the implementation of our sphere model as a class named `Sphere`. The center of the resulting sphere is at the origin. Code for using `Sphere` is also shown. Note that each vertex is stored as an instance of the `graphicslib3D` class `Vertex3D` (this is different from previous examples, where vertices were stored in float arrays). `Vertex3D` includes methods for obtaining the desired vertex components as float values, which are then put into float buffers as before.

Note the calculation of triangle indices in the `Sphere` class, as described earlier in Figure 6.8. The variable “`prec`” refers to the “precision,” which in this case is used both for the number of sphere slices and the number of vertices per slice. Because the texture map wraps completely around the sphere, we will need an extra coincident vertex at each of the points where the left and right edges of the texture map meet. Thus, the total number of vertices is $(\text{prec}+1) * (\text{prec}+1)$. Since six triangle indices are generated per vertex, the total number of indices is $\text{prec} * \text{prec} * 6$.

Program 6.1 Procedurally-Generated Sphere

Sphere class

```
import static java.lang.Math.*;

public class Sphere
{
    private int numVertices, numIndices, prec; // prec = precision
    private int[ ] indices;
    private Vertex3D[ ] vertices;

    public Sphere(int p)
    {
        prec = p;
        initSphere();
    }

    private void initSphere()
    {
        numVertices = (prec+1) * (prec+1);
        numIndices = prec * prec * 6;
        vertices = new Vertex3D[numVertices];
        indices = new int[numIndices];
    }
}
```

```

for (int i=0; i<numVertices; i++)
{
    vertices[i] = new Vertex3D();
}

// calculate triangle vertices
for (int i=0; i<=prec; i++)
{
    for (int j=0; j<=prec; j++)
    {
        float y = (float) cos(toRadians(180-i*180/prec));
        float x = -(float) cos(toRadians(j*360/prec)) * (float) abs(cos(asin(y)));
        float z = (float) sin(toRadians(j*360/prec)) * (float) abs(cos(asin(y)));
        vertices[i*(prec+1)+j].setLocation(new Point3D(x,y,z));
        vertices[i*(prec+1)+j].setS((float)j/prec);           // texture coordinates (s,t)
        vertices[i*(prec+1)+j].setT((float)i/prec);
        vertices[i*(prec+1)+j].setNormal(new Vector3D(vertices[i*(prec+1)+j].getLocation()));
    }
}

// calculate triangle indices
for(int i=0; i<prec; i++)
{
    for(int j=0; j<prec; j++)
    {
        indices[6*(i*prec+j)+0] = i*(prec+1)+j;
        indices[6*(i*prec+j)+1] = i*(prec+1)+j+1;
        indices[6*(i*prec+j)+2] = (i+1)*(prec+1)+j;
        indices[6*(i*prec+j)+3] = i*(prec+1)+j+1;
        indices[6*(i*prec+j)+4] = (i+1)*(prec+1)+j+1;
        indices[6*(i*prec+j)+5] = (i+1)*(prec+1)+j;
    }
}

public int[ ] getIndices() { return indices; }
public Vertex3D[ ] getVertices() { return vertices; }
}

```

Using the Sphere class

```

...
mySphere = new Sphere(48);                                // in init()
...

private void setupVertices()
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    Vertex3D[ ] vertices = mySphere.getVertices();
    int[ ] indices = mySphere.getIndices();

    float[ ] pvalues = new float[indices.length*3];          // vertex positions
    float[ ] tvalues = new float[indices.length*2];          // texture coordinates
    float[ ] nvalues = new float[indices.length*3];          // normal vectors

    for (int i=0; i<indices.length; i++)

```

```

{   pvalues[i*3] = (float) (vertices[indices[i]]).getX();
    pvalues[i*3+1] = (float) (vertices[indices[i]]).getY();
    pvalues[i*3+2] = (float) (vertices[indices[i]]).getZ();

    tvalues[i*2] = (float) (vertices[indices[i]]).getS();
    tvalues[i*2+1] = (float) (vertices[indices[i]]).getT();

    nvalues[i*3] = (float) (vertices[indices[i]]).getNormalX();
    nvalues[i*3+1] = (float)(vertices[indices[i]]).getNormalY();
    nvalues[i*3+2 ]=(float) (vertices[indices[i]]).getNormalZ();
}

gl glGenVertexArrays(vao.length, vao, 0);
gl glBindVertexArray(vao[0]);
gl glGenBuffers(3, vbo, 0);

// put the vertices into buffer #0
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
FloatBuffer vertBuf = Buffers.newDirectFloatBuffer(pvalues);
gl glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4, vertBuf, GL_STATIC_DRAW);

// put the texture coordinates into buffer #1
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
FloatBuffer texBuf = Buffers.newDirectFloatBuffer(tvalues);
gl glBufferData(GL_ARRAY_BUFFER, texBuf.limit()*4, texBuf, GL_STATIC_DRAW);

// put the normal coordinates into buffer #2
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
FloatBuffer norBuf = Buffers.newDirectFloatBuffer(nvalues);
gl glBufferData(GL_ARRAY_BUFFER, norBuf.limit()*4, norBuf, GL_STATIC_DRAW);
}

```

in display()

```

...
int numVerts = mySphere.getIndices().length;
gl glDrawArrays(GL_TRIANGLES, 0, numVerts);
...

```

When using the `Sphere` class, we will need three values for each vertex position and normal vector, but only two values for each texture coordinate. This is reflected in the declarations for the arrays (`pvalues`, `tvalues`, and `nvalues`) that are later populated with values obtained by calls to `Sphere` functions, and loaded into the buffers.

It is important to note that although indexing is used in the process of building the sphere, the ultimate sphere vertex data stored in the VBOs doesn't utilize

indexing. Rather, as `setupVertices()` loops through the sphere indices, it generates separate (often redundant) vertex entries in the VBO for each of the index entries. OpenGL does have a mechanism for indexing vertex data; for simplicity we didn't use it in this example, but we will use OpenGL's indexing in the next example.

Figure 6.9 shows the output of Program 6.1, with a precision of 48. The texture from Figure 6.5 has been loaded as described in Chapter 5.

Many other models can be created procedurally, from geometric shapes to real-world objects. One of the most well-known is the “*Utah teapot*” [CH16], which was developed in 1975 by Martin Newell, using a variety of Bézier curves and surfaces. The *OpenGL Utility Toolkit* (or “*GLUT*”) [GL16] even includes procedures for drawing teapots(!) (see Figure 6.10). We don’t cover GLUT in this book, but Bézier surfaces are covered in Chapter 11.



Figure 6.9
Textured sphere model.



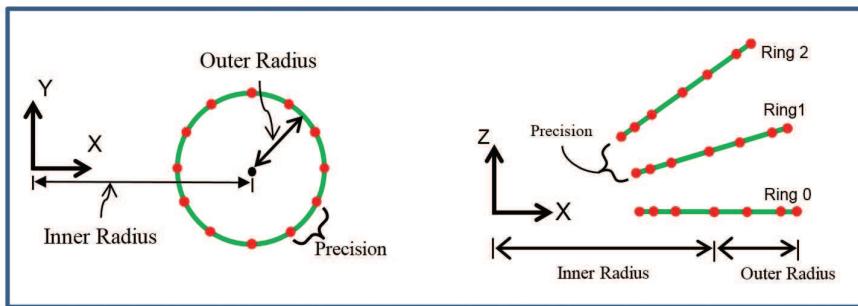
Figure 6.10
OpenGL GLUT teapot.

6.2 OPENGL INDEXING – BUILDING A TORUS

6.2.1 The Torus

Algorithms for producing a *torus* can be found on various websites. Paul Baker gives a step-by-step description for defining a circular slice, and then rotating the slice around a circle to form a donut [PP07]. Figure 6.11 shows two views; from the side, and from above.

The way that the torus vertex positions are generated is rather different from what was done to build the sphere. For the torus, the algorithm positions a vertex to the right of the origin, and then rotates that vertex in a circle on the XY plane using a rotation around the Z axis to form a “ring.” Texture coordinates and normal vectors are computed for each of these vertices as they are built. An additional

**Figure 6.11**

Building a torus.

vector tangent to the surface of the torus (called the *tangent vector*) is also generated for each vertex.

Vertices for additional torus rings are formed by rotating the original ring around the Y axis. Tangent and normal vectors for each resulting vertex are computed by also rotating the tangent and normal vectors of the original ring around the Y axis. After the vertices are created, they are traversed from ring to ring, and for each vertex two triangles are generated. The generation of six index table entries comprising the two triangles is done in a similar manner as we did for the sphere.

Our strategy for choosing texture coordinates will be to arrange them so that the S axis of the texture image wraps halfway around the horizontal perimeter of the torus, and then repeats for the other half. This can be done by using the *tiling* capability in OpenGL textures previously described in Chapter 5. As we rotate around the Y axis generating the rings, we specify a variable *ring* that starts at 1 and increases up to the specified precision (again dubbed “*prec*”). We set the value of the S texture coordinate value to $2.0 * \text{ring} / \text{prec}$, causing S to range between 0 and 2. Recalling that the default behavior for textures when a texture coordinate exceeds 1 is `GL_REPEAT`, the result is that the texture stretches halfway around the torus, and is then repeated on the second half. The motivation for this approach is to avoid having the texture image appear overly “stretched” horizontally.

Building a torus class in Java/JOGL could be done in a virtually identical manner as for the `Sphere` class. However, we have the opportunity to take advantage of the indices that we created while building the torus by using OpenGL’s support for vertex indexing (we could have also done this for the sphere, but we didn’t). For very large models with thousands of vertices, using OpenGL indexing can result in improved performance, and so we will describe how to do that next.

6.2.2 Indexing in OpenGL

In both our sphere and torus models, we generated an array of integer indexes referencing into the vertex array. In the case of the sphere, we used the list of indices to build a complete set of individual vertices and loaded them into a VBO just as we did for examples in earlier chapters. Instantiating the torus and loading its vertices, normals, etc. into buffers could be done in a similar manner as was done in Program 6.1, but instead we will use OpenGL's indexing.

When using OpenGL indexing, we also load the indices themselves into a VBO. We generate one additional VBO for holding the indices. Since each index value is simply an integer reference, we first copy the index array into a Java `IntBuffer`, and then use `glBufferData()` to load the `IntBuffer` into the added VBO, specifying that the VBO is of type `GL_ELEMENT_ARRAY_BUFFER` (this tells OpenGL that the VBO contains indices). The code that does this can be added to `setupVertices()`:

```
int[] indices = myTorus.getIndices(); // the torus index accessor returns the indices as an int array
...
gl glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]); // vbo #3 is the additional added vbo
IntBuffer idxBuf = Buffers.newDirectIntBuffer(indices);
gl glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, idxBuf.limit()*4, idxBuf, GL_STATIC_DRAW);
```

In the `display()` method, we replace the `glDrawArrays()` call with a call to `glDrawElements()`, which tells OpenGL to utilize the index VBO for looking up the vertices to be drawn. We also enable the VBO that contains the indices by using `glBindBuffer()`, specifying which VBO contains the indices, and that it is a `GL_ELEMENT_ARRAY_BUFFER`. The code is as follows:

```
int numIndices = myTorus.getIndices().length;
gl glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
gl DrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_INT, 0);
```

Interestingly, the shaders used for drawing the sphere continue to work, unchanged, for the torus, even with the changes that we made in the Java/JOGL application to implement indexing. OpenGL is able to recognize the presence of a `GL_ELEMENT_ARRAY_BUFFER` and utilize it to access the vertex attributes.

Program 6.2 shows a class named `Torus` based on Baker's implementation. The “inner” and “outer” variables refer to the corresponding inner and outer radius in Figure 6.11. The `prec` (“precision”) variable has a similar role as in the sphere, with

analogous computations for number of vertices and indices. By contrast, determining normal vectors is much more complex than it was for the sphere. We have used the strategy given in Baker's description, wherein two tangent vectors are computed (dubbed *tangent* and *bitangent*), and their cross-product forms the normal.

The torus is also available in `graphicslib3D`, by instantiating class `graphicslib3D.shape.Torus`.

Program 6.2 Procedurally-Generated Torus

Torus class

```
public class Torus
{
    private int numVertices, numIndices, prec;
    private int[ ] indices;
    private Vertex3D[ ] vertices;
    private float inner, outer;
    private Vector3D[ ] tTangent, bTangent;

    public Torus(float innerRadius, float outerRadius, int p)
    {   inner = innerRadius; outer=outerRadius; prec=p;
        initTorus();
    }

    private void initTorus()
    {   numVertices = (prec+1) * (prec+1);
        numIndices = prec * prec * 6;
        vertices = new Vertex3D[numVertices];
        indices = new int[numIndices];

        tTangent = new Vector3D[numVertices];
        bTangent = new Vector3D[numVertices];

        for (int i=0; i<numVertices; i++)
        {   vertices[i] = new Vertex3D(); }

        // calculate first ring.
        for (int i=0; i<prec+1; i++)
        {   // build the ring by rotating points around the origin, then moving them outward
            Point3D initOuterPos = new Point3D(outer, 0.0, 0.0);
            Point3D rotOuterPos = tRotateZ(initOuterPos, (i * 360.0f / prec));
            Point3D initInnerPos = new Point3D(inner,0.0, 0.0);
            Point3D ringPos = rotOuterPos.add(initInnerPos);
            vertices[i].setLocation (ringPos);
        }
    }
}
```

```

// compute texture coordinates for each vertex on the ring
vertices[i].setS(0.0f);
vertices[i].setT(((float)i)/((float)prec));

// compute normal vectors for each vertex in the ring
Vector3D negZ = new Vector3D(0.0f, 0.0f, -1.0f); // tangent vector of first outer ring = -Z axis
Vector3D negY = new Vector3D(0.0f, -1.0f, 0.0f); // initial bitangent vector = -Y axis
tTangent[i] = negZ; // the tangent vector is saved (and thus available to the application)
// the bitangent is then rotated around the Z axis, and also saved
bTangent[i] = new Vector3D(tRotateZ(new Point3D(negY),(i*360.0f/(prec))));
vertices[i].setNormal(tTangent[i].cross(bTangent[i])); // cross product produces the normal
}

// rotate the first ring about the Y axis to generate the other rings of the torus
for (int ring=1; ring<=prec; ring++)
{
    for (int vert=0; vert<=prec; vert++)
    {
        // rotate the vertex positions of the original ring around the Y axis
        float rotAmt = (float) ((float)ring*360.0f/prec);
        Vector3D vp = new Vector3D(vertices[vert].getLocation());
        Vector3D vpr = tRotateY(vp, rotAmt);
        vertices[ring*(prec+1)+vert].setLocation(new Point3D(vpr));

        // compute the texture coordinates for the vertices in the new rings
        vertices[ring*(prec+1)+vert].setS((float)ring / (float)prec);
        vertices[ring*(prec+1)+vert].setT(vertices[vert].getT());

        // rotate the tangent and bitangent vectors around the Y axis
        tTangent[ring*(prec+1)+vert] = tRotateY(tTangent[vert], rotAmt);
        bTangent[ring*(prec+1)+vert] = tRotateY(bTangent[vert], rotAmt);

        // rotate the normal vector around the Y axis
        Vector3D normalRotateY = tRotateY(vertices[vert].getNormal(), rotAmt);
        vertices[ring*(prec+1)+vert].setNormal(normalRotateY);
    }
}

// calculate triangle indices corresponding to the two triangles built per vertex
for(int ring=0; ring<prec; ring++)
{
    for(int vert=0; vert<prec; vert++)
    {
        indices[((ring*prec+vert)*2)*3+0] = ring*(prec+1)+vert;
        indices[((ring*prec+vert)*2)*3+1] = (ring+1)*(prec+1)+vert;
        indices[((ring*prec+vert)*2)*3+2] = ring*(prec+1)+vert+1;
        indices[((ring*prec+vert)*2+1)*3+0] = ring*(prec+1)+vert+1;
        indices[((ring*prec+vert)*2+1)*3+1] = (ring+1)*(prec+1)+vert;
        indices[((ring*prec+vert)*2+1)*3+2] = (ring+1)*(prec+1)+vert+1;
    }
}
}

```

```

// utility function for rotating a vector around the Y axis
private Vector3D tRotateY(Vector3D inVec, float amount)
{
    Matrix3D yMat = new Matrix3D();
    yMat.rotateY((double)amount);
    Vector3D result = inVec.mult(yMat);
    return result;
}

// utility function for rotating a point around the Z axis
private Point3D tRotateZ(Point3D inPt, float amount)
{
    Matrix3D zMat = new Matrix3D();
    zMat.rotateZ((double)amount);
    Point3D result = inPt.mult(zMat);
    return result;
}

// accessors for the torus indices and vertices
public int[ ] getIndices() { return indices; }

public Vertex3D[ ] getVertices() { return vertices; }
}

```

Using the Torus class (with OpenGL indexing)

```

...
myTorus = new Torus(0.5f, 0.2f, 48);                                // in init()
...

private void setupVertices()
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    Vertex3D[ ] vertices = myTorus.getVertices();
    int[ ] indices = myTorus.getIndices();

    float[ ] pvalues = new float[vertices.length*3];
    float[ ] tvalues = new float[vertices.length*2];
    float[ ] nvalues = new float[vertices.length*3];

    for (int i=0; i<vertices.length; i++)
    {
        pvalues[i*3] = (float) (vertices[i]).getX();                      // vertex position
        pvalues[i*3+1] = (float) (vertices[i]).getY();
        pvalues[i*3+2] = (float) (vertices[i]).getZ();
        tvalues[i*2] = (float) (vertices[i]).getS();                         // texture coordinates
        tvalues[i*2+1] = (float) (vertices[i]).getT();
        nvalues[i*3] = (float) (vertices[i]).getNormalX();                   // normal vector
    }
}

```

```

    nvalues[i*3+1] = (float)(vertices[i]).getNormalY();
    nvalues[i*3+2] = (float)(vertices[i]).getNormalZ();
}

gl glGenVertexArrays(vao.length, vao, 0);
gl glBindVertexArray(vao[0]);
gl glGenBuffers(vbo.length, vbo, 0);           // generate VBOs as before, plus one for indices

gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]); // vertex positions
FloatBuffer vertBuf = Buffers.newDirectFloatBuffer(pvalues);
glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4, vertBuf, GL_STATIC_DRAW);

gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]); // texture coordinates
FloatBuffer texBuf = Buffers.newDirectFloatBuffer(tvalues);
glBufferData(GL_ARRAY_BUFFER, texBuf.limit()*4, texBuf, GL_STATIC_DRAW);

gl glBindBuffer(GL_ARRAY_BUFFER, vbo[2]); // normal vectors
FloatBuffer norBuf = Buffers.newDirectFloatBuffer(nvalues);
glBufferData(GL_ARRAY_BUFFER, norBuf.limit()*4, norBuf, GL_STATIC_DRAW);

gl glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]); // indices
IntBuffer idxBuf = Buffers.newDirectIntBuffer(indices);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, idxBuf.limit()*4, idxBuf, GL_STATIC_DRAW);
}

```

in display()

```

...
int numIndices = myTorus.getIndices().length;
gl glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
gl drawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_INT, 0);
...

```

Note in the code that uses the Torus class that the loop in `setupVertices()` now stores the data associated with each vertex once, rather than once for each index entry (as was the case in the sphere example). This difference is also reflected in the declared array sizes for the data being entered into the VBOs. Also note that in the torus example, rather than using the index values when retrieving vertex data, they are simply loaded into VBO #3. Since that VBO is designated as a `GL_ELEMENT_ARRAY_BUFFER`, OpenGL knows that that VBO contains vertex indices.

Figure 6.12 shows the result of instantiating a torus, and texturing it with the brick texture.

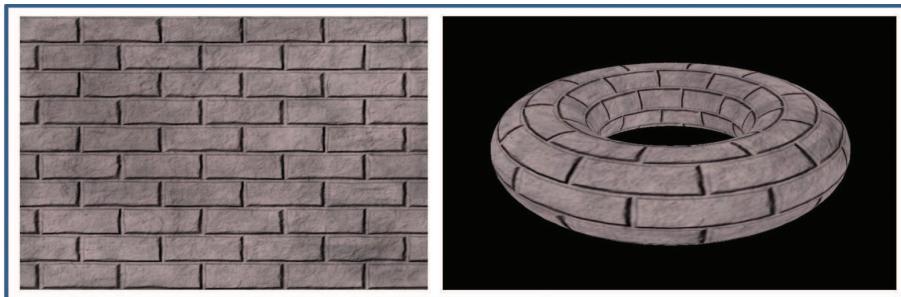


Figure 6.12
Procedurally-generated torus.

6.3 LOADING EXTERNALLY PRODUCED MODELS

Complex 3D models, such as characters found in videogames or computer-generated movies, are typically produced using modeling tools. Such “DCC” (Digital Content Creation) tools make it possible for people (such as artists) to build arbitrary shapes in 3D space, and automatically produce the vertices, texture coordinates, vertex normals, and so on. There are too many such tools to list, but some examples are Maya, Blender, Lightwave, Cinema4D, and many others. Blender is free and open source. Figure 6.13 shows an example Blender screen during the editing of a 3D model.



Figure 6.13
Example Blender model creation [BL16].

In order for us to use a DCC-created model in our OpenGL scenes, that model needs to be saved (exported) in a format that we can read (import) into our

program. There are several standard 3D model file formats; again, there are too many to list, but some examples are Wavefront (.obj), 3D Studio Max (.3ds), Stanford Scanning Repository (.ply), Ogre3D (.mesh), and many others. Arguably the simplest is Wavefront (usually dubbed OBJ), so we will examine that one.

OBJ files are simple enough that we can develop a basic importer relatively easily. In an OBJ file, lines of text specify vertex geometric data, texture coordinates, normals, and other information. It has some limitations—for example, OBJ files have no way of specifying model animation.

Lines in an OBJ file start with a character tag indicating what kind of data is on that line. Some common tags include:

- **v** – geometric (vertex location) data
- **vt** – texture coordinates
- **vn** – vertex normal
- **f** – face (typically vertices in a triangle)

Other tags make it possible to store the object name, materials it uses, curves, shadows, and many other details. We will limit our discussion to the four tags listed above, which are sufficient for importing a wide variety of complex models.

Suppose we use Blender to build a simple pyramid such as the one we developed for Program 4.3. Here is a screenshot of a similar pyramid being created in Blender:

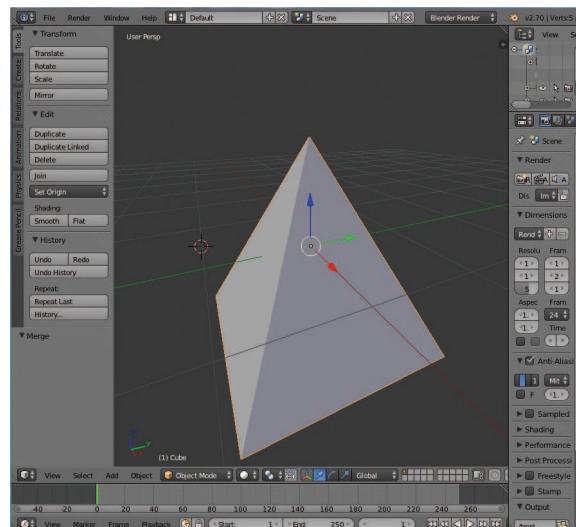


Figure 6.14
Pyramid built in Blender.

In Blender, if we now *export* our pyramid model, specify .obj format, and also set Blender to output texture coordinates and vertex normals, an OBJ file is created that includes all of this information. The resulting OBJ file is shown in Figure 6.15. (The actual values of the texture coordinates can vary depending on how the model is built.)

```
# Blender v2.70 (sub 0) OBJ File: ""
# www.blender.org
o Pyramid
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 0.000000 1.000000 0.000000
vt 0.515829 0.258220
vt 0.515829 0.750612
vt 0.023438 0.750612
vt 0.370823 0.790246
vt 0.820312 0.388210
vt 0.820312 0.991264
vt 0.566135 0.988689
vt 0.015625 0.742493
vt 0.566135 0.496298
vt 0.015625 0.250102
vt 0.566135 0.003906
vt 1.000000 0.000000
vt 1.000000 0.603054
vt 0.550510 0.402036
vt 0.023438 0.258220
vn 0.000000 -1.000000 0.000000
vn 0.894427 0.447214 0.000000
vn -0.000000 0.447214 0.894427
vn -0.894427 0.447214 -0.000000
vn 0.000000 0.447214 -0.894427
s off
f 2/1/1 3/2/1 4/3/1
f 1/4/2 5/5/2 2/6/2
f 2/7/3 5/8/3 3/9/3
f 3/9/4 5/10/4 4/11/4
f 5/12/5 1/13/5 4/14/5
f 1/15/1 2/1/1 4/3/1
```

Figure 6.15

Exported OBJ file for the pyramid.

We have color-coded the important sections of the OBJ file for reference. The lines at the top beginning with “#” are comments placed there by Blender, which our importer ignores. This is followed by a line beginning with “o” giving the name of the object. Our importer can ignore this line as well. Later, there is a line beginning with “s”, that specifies that the faces shouldn’t be smoothed. Our code will also ignore lines starting with “s”.

The first substantive set of lines in the OBJ file are those starting with “v”, colored **blue**. They specify the X, Y, and Z local spatial coordinates of the five vertices of our pyramid model relative to the origin, which in this case is at the center of the pyramid.

The values colored **red** (starting with “vt”) are the various texture coordinates. The reason that the list of texture coordinates is longer than the list of vertices is that some of the vertices participate in more than one triangle, and in those cases different texture coordinates might be used.

The values colored **green** (starting with “vn”) are the various normal vectors. This list too is often longer than the list of vertices (although not in this example), again because some of the vertices participate in more than one triangle, and in those cases different normal vectors might be used.

The values colored **purple** (starting with “f”), near the bottom of the file, specify the triangles (i.e., “faces”). In this example, each face (triangle) has three elements, each with three values separated by “/” (OBJ allows other formats as well). The values for each element are indices into the lists of vertices, texture coordinates, and normal vectors, respectively. For example, the third face is:

```
f 2/7/3 5/8/3 3/9/3
```

This indicates that the 2nd, 5th, and 3rd vertices from the list of vertices (in blue) comprise a triangle (note that OBJ indices start at 1). The corresponding texture coordinates are the 7th, 8th, and 9th from the list of texture coordinates in the section colored red. All three vertices have the same normal vector, the 3rd in the list of normals in the section colored green.

Models in OBJ format are not required to have normal vectors, or even texture coordinates. If a model does not have texture coordinates or normals, the face values would specify only the vertex indices:

```
f 2 5 3
```

If a model has texture coordinates, but not normal vectors, the format would be as follows:

```
f 2/7 5/8 3/9
```

And, if the model has normals but not texture coordinates, the format would be:

```
f 2//3 5//3 3//3
```

It is not unusual for a model to have tens of thousands of vertices. There are hundreds of such models available for download on the Internet for nearly every conceivable application, including models of animals, buildings, cars, planes, mythical creatures, people, and so on.

Programs of varying sophistication that can import an OBJ model are available on the Internet. Alternatively, it is relatively easy to write a very simple OBJ loader function that can handle the basic tags we have seen (`v`, `vt`, `vn`, and `f`). Program 6.3 shows one such loader, albeit a *very* limited one. It incorporates a class to hold an arbitrary imported model, which in turn calls the importer.

Before we describe the code in our simple OBJ importer, we must warn the reader of its limitations:

- It only supports models that include *all three* face attribute fields. That is, vertex positions, texture coordinates, and normals *must all* be present and in the form: `f #/#/# #/#/# #/#/#`
- The material tag is ignored—texturing must be done using the methods seen in Chapter 5.
- Only OBJ models comprised of a single triangle mesh are supported (the OBJ format supports composite models, but our simple importer does not).
- It assumes that elements on each line are separated by exactly one space.

If you have an OBJ model that doesn't satisfy all of the above criteria, and you wish to import it using the simple loader in Program 6.3, it *may* still be feasible to do so. It is often possible to load such a model into Blender, and then export it to another OBJ file that accommodates the loader's limitations. For instance, if the model doesn't include normal vectors, it is possible to have Blender produce normal vectors while it exports the revised OBJ file.

Another limitation of our OBJ loader has to do with indexing. Observe in the previous descriptions that the “face” tag allows for the possibility of mix-and-matching vertex positions, texture coordinates, and normal vectors. For example, two different “face” rows may include indices which point to the same v entry, but different vt entries. Unfortunately, OpenGL’s indexing mechanism does not support this level of flexibility—index entries in OpenGL can only point to a particular vertex *along with its attributes*. This complicates writing an OBJ model loader somewhat, as we cannot simply copy the references in the triangle face entries into an index array. Rather, using OpenGL indexing would require ensuring that entire *combinations* of v, vt, and vn values for a face entry each have their own references in the index array. A simpler, albeit less efficient, alternative is to create a new vertex for every triangle face entry. We opt for this simpler approach here in the interest of clarity, despite the space-saving advantage of using OpenGL indexing (especially when loading larger models).

The ModelImporter class includes a `parseOBJ()` function that reads in each line of an OBJ file one-by-one, handling separately the four cases v, vt, vn, and f. In each case, the subsequent numbers on the line are extracted, first by using `substring()` to skip the initial v, vt, vn, or f character(s), and then using the iterator returned by the `split()` function to extract each subsequent parameter value, storing them in an `ArrayList`. As the face (f) entries are processed, the vertices are built, with corresponding entries in parallel arrays for vertex positions, texture coordinates, and normal vectors.

The ModelImporter class is embedded in the ImportedModel class, which simplifies loading and accessing the vertices of an OBJ file by putting the imported vertices into an array of `Vertex3D` objects. Recall that `Vertex3D` is defined in `graphicslib3D`, and that a `Vertex3D` object includes fields for vertex position, texture coordinates, and normal vectors. The accessors in the ImportedModel class then make this array of `Vertex3D` objects available to the Java/Jogl application in much the same manner as was done in the Sphere and Torus classes.

Following the ModelImporter and ImportedModel classes is an example sequence of calls for loading an OBJ file, and then transferring the vertex information into a set of VBOs for subsequent rendering.

Figure 6.16 shows a rendered model of the space shuttle, downloaded as an OBJ file from the NASA website [NA16], imported using the code from Program 6.3, and textured using the code from Program 5.1 with the associated NASA texture image file, with anisotropic filtering. This texture image is an example of the

use of *UV-mapping*, where texture coordinates in the model are carefully mapped to particular regions of the texture image. (As mentioned earlier, the details of UV-mapping are outside the scope of this book.)

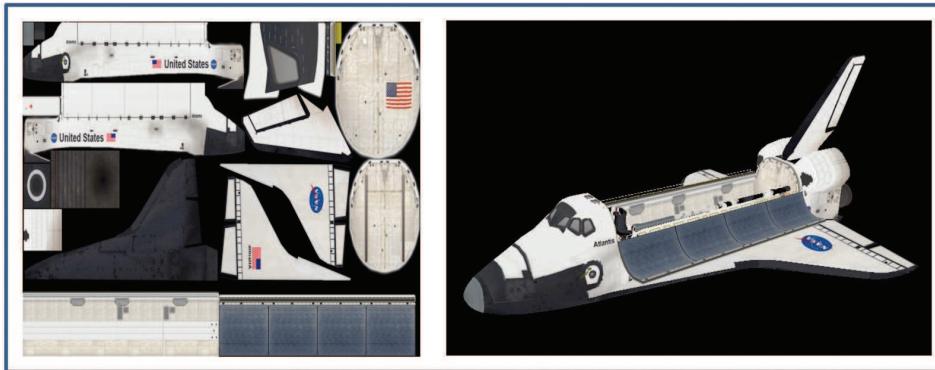


Figure 6.16

NASA space shuttle model with texture.

Program 6.3 Simple (Limited) *OBJ* Loader

ImportedModel class

```
public class ImportedModel
{
    private Vertex3D [ ] vertices;
    private int numVertices;

    public ImportedModel(String filename)
    {
        ModellImporter modellImporter = new ModellImporter();
        try
        {
            modellImporter.parseOBJ(filename);      // uses the modellImporter to get vertex information
            numVertices = modellImporter.getNumVertices();
            float[ ] verts = modellImporter.getVertices();
            float[ ] tcs = modellImporter.getTextureCoordinates();
            float[ ] normals = modellImporter.getNormals();

            vertices = new Vertex3D[numVertices];
            for(int i=0; i<vertices.length; i++)
            {
                vertices[i] = new Vertex3D();
                vertices[i].setLocation(verts[i*3], verts[i*3+1], verts[i*3+2]);
                vertices[i].setST(tcs[i*2], tcs[i*2+1]);
                vertices[i].setNormal(normals[i*3], normals[i*3+1], normals[i*3+2]);
            }
        }
    }
}
```

```

        } catch (IOException e)
        { e.printStackTrace();
    }

    public Vertex3D[ ] getVertices() { return vertices; }      // accessors application to obtain
    public int getNumVertices() { return numVertices; }
}

```

ModellImporter nested class

```

private class ModellImporter
{ // values as read in from OBJ file
    private ArrayList<Float> vertVals = new ArrayList<Float>();
    private ArrayList<Float> stVals = new ArrayList<Float>();
    private ArrayList<Float> normVals = new ArrayList<Float>();

    // values stored for later use as vertex attributes
    private ArrayList<Float> triangleVerts = new ArrayList<Float>();
    private ArrayList<Float> textureCoords = new ArrayList<Float>();
    private ArrayList<Float> normals = new ArrayList<Float>();

    public void parseOBJ(String filename) throws IOException
    { InputStream input = ModellImporter.class.getResourceAsStream(filename);
        BufferedReader br = new BufferedReader(new InputStreamReader(input));
        String line;
        while ((line = br.readLine()) != null)
        { if(line.startsWith("v "))
            { for(String s : (line.substring(2)).split(" "))
                { vertVals.add(Float.valueOf(s));
            }
            else if(line.startsWith("vt"))
            { for(String s : (line.substring(3)).split(" "))
                { stVals.add(Float.valueOf(s));
            }
            else if(line.startsWith("vn"))
            { for(String s : (line.substring(3)).split(" "))
                { normVals.add(Float.valueOf(s));
            }
            else if(line.startsWith("f"))
            { for(String s : (line.substring(2)).split(" "))
                { String v = s.split("/")[0];
                    String vt = s.split("/")[1];
                    String vn = s.split("/")[2];
                    int vertRef = (Integer.valueOf(v)-1)*3;

```

```

int tcRef = (Integer.valueOf(vt)-1)*2;
int normRef = (Integer.valueOf(vn)-1)*3;

triangleVerts.add(vertVals.get(vertRef));           // build array of vertices
triangleVerts.add(vertVals.get(vertRef+1));
triangleVerts.add(vertVals.get(vertRef+2));

textureCoords.add(stVals.get(tcRef));              // corresponding
textureCoords.add(stVals.get(tcRef+1));            // texture coordinates.

normals.add(normVals.get(normRef));                //... and normals
normals.add(normVals.get(normRef+1));
normals.add(normVals.get(normRef+2));

} } }
input.close();
}

// accessors for retrieving the number of vertices, the vertices themselves,
// and the corresponding texture coordinates and normal vectors (only called once per model)

public int getNumVertices() { return (triangleVerts.size()/3); }

public float[ ] getVertices()
{
    float[ ] p = new float[triangleVerts.size()];
    for(int i = 0; i < triangleVerts.size(); i++)
    {
        p[i] = triangleVerts.get(i);
    }
    return p;
}

// similar accessors for texture coordinates and normal vectors go here
}
}
}

```

Using the Model Importer

```

...
myObj = new ImportedModel("shuttle.obj");           // in init()
...

private void setupVertices(GL4 gl)
{
    Vertex3D[ ] vertices = myObj.getVertices();
    numObjVertices = myObj.getNumVertices();

    float[ ] pvalues = new float[numObjVertices*3];      // vertex positions
    float[ ] tvalues = new float[numObjVertices*2];      // texture coordinates

```

```

float[ ] nvalues = new float[numObjVertices*3];      // normal vectors

for (int i=0; i<numObjVertices; i++)
{
    pvalues[i*3] = (float) (vertices[i]).getX();
    pvalues[i*3+1] = (float) (vertices[i]).getY();
    pvalues[i*3+2] = (float) (vertices[i]).getZ();
    tvalues[i*2] = (float) (vertices[i]).getS();
    tvalues[i*2+1] = (float) (vertices[i]).getT();
    nvalues[i*3] = (float) (vertices[i]).getNormalX();
    nvalues[i*3+1] = (float) (vertices[i]).getNormalY();
    nvalues[i*3+2] = (float) (vertices[i]).getNormalZ();
}

gl glGenVertexArrays(vao.length, vao, 0);
gl glBindVertexArray(vao[0]);
gl glGenBuffers(vbo.length, vbo, 0);

// VBO for vertex locations
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
FloatBuffer vertBuf = Buffers.newDirectFloatBuffer(pvalues);
gl glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4, vertBuf, GL_STATIC_DRAW);

// VBO for texture coordinates
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
FloatBuffer texBuf = Buffers.newDirectFloatBuffer(tvalues);
gl glBufferData(GL_ARRAY_BUFFER, texBuf.limit()*4, texBuf, GL_STATIC_DRAW);

// VBO for normal vectors
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
FloatBuffer norBuf = Buffers.newDirectFloatBuffer(nvalues);
gl glBufferData(GL_ARRAY_BUFFER, norBuf.limit()*4,norBuf, GL_STATIC_DRAW);
}

```

in display():

```

...
int numVerts = myModel.getVertices().length;
gl glDrawArrays(GL_TRIANGLES, 0, numVerts);

```

SUPPLEMENTAL NOTES

Although we discussed the use of DCC tools for creating 3D models, we didn't discuss how to use such tools. There is a wealth of tutorial videos and documentation for all of the popular tools such as Blender and Maya, and such instruction is outside the scope of this text.

The topic of 3D modeling is itself a rich field of study. Our coverage in this chapter has been just a rudimentary introduction, with emphasis on its relationship to OpenGL. Many universities offer entire courses in 3D modeling, and readers interested in learning more about it are encouraged to consult some of the popular resources that offer greater detail (e.g., [BL16], [C13], [V12]).

We reiterate that the OBJ importer we presented in this chapter is limited, and can only handle a subset of the features supported by the OBJ format. Although sufficient for our needs, it will fail on some OBJ files. In those cases it would be necessary to first load the model into Blender (or Maya, etc.), and re-export it as an OBJ file that complies with the importer's limitations as described earlier in this chapter.

Exercises

- 6.1** Modify Program 4.4 so that the “sun,” “planet,” and “moon” are textured *spheres*, such as the ones shown in Figure 4.11.
- 6.2 (PROJECT)** Modify your program from Exercise 6.1 so that the imported NASA shuttle object from Figure 6.16 also orbits the “sun.” You’ll want to experiment with the scale and rotation applied to the shuttle to make it look realistic.
- 6.3 (RESEARCH & PROJECT)** Learn the basics of how to use Blender to create a 3D object of your own. To make full use of Blender with your OpenGL applications, you’ll want to learn how to use Blender’s UV-unwrapping tools to generate texture coordinates and an associated texture image. You can then export your object as an OBJ file and load it using the code from Program 6.3.

References

- [BL16] Blender, The Blender Foundation, accessed July 2016, <https://www.blender.org/>.
- [CH11] A. Chopine, *3D Art Essentials: The Fundamentals of 3D Modeling, Texturing, and Animation* (Focal Press, 2011).
- [CH16] Computer History Museum, Accessed July 2016, <http://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206>.
- [GL16] GLUT and OpenGL Utility Libraries, accessed July 2016, <https://www.opengl.org/resources/libraries/>.
- [NA16] NASA 3D Resources, accessed July 2016, <http://nasa3d.arc.nasa.gov/>.
- [PP07] P. Baker, *Paul's Projects*, 2007, accessed July 2016, www.paulsprojects.net.
- [VA12] V. Vaughan, *Digital Modeling* (New Riders, 2012).
- [VE16] Visible Earth, NASA Goddard Space Flight Center Image, Accessed July 2016, <http://visibleearth.nasa.gov>.

LIGHTING

7.1	<i>Lighting Models</i>	<i>157</i>
7.2	<i>Lights</i>	<i>159</i>
7.3	<i>Materials</i>	<i>163</i>
7.4	<i>ADS Lighting Computations</i>	<i>165</i>
7.5	<i>Implementing ADS Lighting</i>	<i>168</i>
7.6	<i>Combining Lighting and Textures</i>	<i>182</i>
	<i>Supplemental Notes</i>	<i>184</i>
	<i>Historical Note</i>	<i>185</i>



Light affects the appearance of our world in varied and sometimes dramatic ways. When a flashlight shines on an object, we expect it to appear brighter on the side facing the light. The earth on which we live is itself brightly lit where it faces the sun at noon, but as it turns, that daytime brightness gradually fades into evening, until becoming completely dark at midnight.

Objects also respond differently to light. Besides having different colors, objects can have different reflective characteristics. Consider two objects, both green, but where one is made of cloth versus another made of polished steel—the latter will appear more “shiny.”

7.1 LIGHTING MODELS

Light is the product of photons being emitted by high energy sources and subsequently bouncing around until some of the photons reach our eyes. Unfortunately, it is computationally infeasible to emulate this natural process, as it would require simulating and then tracking the movement of a huge number of photons, adding many objects (and matrices) to our scene. What we need is a *lighting model*.

Lighting models are sometimes called *shading models*, although in the presence of shader programming, that can be a bit confusing. Sometimes the term *reflection model* is used, complicating the terminology further. We will try and stick to whichever terminology is simple and most practical.

The most common lighting models today are called “ADS” models, because they are based on three types of reflection labeled A, D, and S:

- **Ambient** reflection simulates a low-level illumination that equally affects everything in the scene.
- **Diffuse** reflection brightens objects to various degrees depending on the light’s angle of incidence.
- **Specular** reflection conveys the shininess of an object by strategically placing a highlight of appropriate size on the object’s surface where light is reflected most directly towards our eyes.

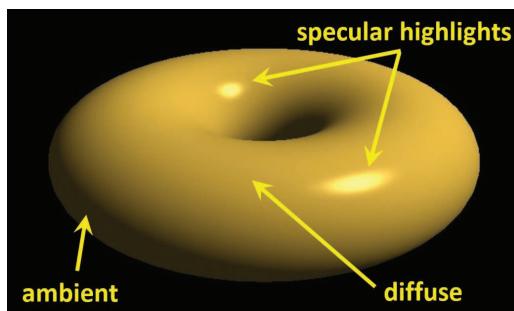


Figure 7.1
ADS Lighting contributions.

ADS models can be used to simulate different lighting effects, and a variety of materials.

Figure 7.1 illustrates the ambient, diffuse, and specular contributions of a positional light on a shiny gold torus.

Recall that a scene is ultimately drawn by having the fragment shader output a color for each pixel on the screen. Using an ADS lighting model

requires specifying contributions due to lighting on a pixel’s RGBA output value. Factors include:

- The type of light source, and its ambient, diffuse, and specular characteristics
- The object’s material’s ambient, diffuse, and specular characteristics
- The object’s material’s specified “shininess”
- The angle at which the light hits the object
- The angle from which the scene is being viewed

7.2 LIGHTS

There are many types of lights, each with different characteristics and requiring different steps to simulate their effects. Some types include:

- Global (usually called “global ambient” because it includes only an ambient component)
- Directional (or “distant”)
- Positional (or “point source”)
- Spotlight

Global ambient light is the simplest type of light to model. Global ambient light has no source position —the light is equal everywhere, at each pixel on every object in the scene, regardless of where the objects are. Global ambient lighting simulates the real-world phenomenon of light that has bounced around so many times that its source and direction are undeterminable. Global ambient light has only an ambient component, specified as an RGBA value; it has no diffuse or specular components. For example, global ambient light can be defined as follows:

```
float[ ] globalAmbient = new float[ ] { 0.7f, 0.7f, 0.7f, 1.0f }; // RGBA values range from 0 to 1
```

Alternatively, global ambient light can be specified using the `graphicslib3D` `AmbientLight` class, which is available as a singleton:

```
AmbientLight globalAmbient = getAmbientLight();
globalAmbient.setRed(0.7f);
globalAmbient.setGreen(0.7f);
globalAmbient.setBlue(0.7f);
globalAmbient.setAlpha(1.0f);
```

Directional, or **distant** light also doesn’t have a source location, but it does have a *direction*. It is useful for situations where the source of the light is so far away that the light rays are effectively parallel, such as light coming from the sun. In many such situations we may only be interested in modeling the light, and not the object that produces the light. The effect of directional light on an object depends on the light’s angle of impact; objects are brighter on the side facing a directional light than on a tangential or opposite side. Modeling directional light requires specifying its direction (as a vector), and its ambient, diffuse, and

specular characteristics (as RGBA values). A red directional light pointing down the negative Z axis might be specified thusly:

```
float[] dirLightAmbient = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] dirLightDiffuse = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] dirLightSpecular = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] dirLightDirection = new float[] { 0.0f, 0.0f, -1.0f };
```

Directional lights are included in `graphicslib3D` in the class `DistantLight`, which includes an accessor for direction, as well as accessors for ambient, diffuse, and specular fields. Instantiating a red directional light pointing down the negative Z axis in `graphicslib3D` can therefore be done thusly:

```
DistantLight dl1 = new DistantLight();
Vector3D directionVector = new Vector3D(0, 0, -1);
float[] amb = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] dif = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] spec = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
dl1.setAmbient(amb);
dl1.setDiffuse(dif);
dl1.setSpecular(spec);
dl1.setDirection(directionVector);
```

A ***Positional*** light has a specific location in the 3D scene. Light sources that are near the scene, such as lamps, candles, and so forth, are examples. Like directional lights, the effect of a positional light depends on angle of impact; however, its direction is not specified, as it is different for each vertex in our scene. Positional lights may also incorporate *attenuation factors*, in order to model how their intensity diminishes with distance. As with the other types of lights we have seen, positional lights have ambient, diffuse, and specular properties specified as RGB values. A red positional light at location (5, 2, -3) could for example be specified thusly:

```
float[] posLightAmbient = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] posLightDiffuse = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] posLightSpecular = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] posLightLocation = new float[] { 5.0f, 2.0f, -3.0f };
```

Positional lights are included in `graphicslib3D` in the class `PositionalLight`, which includes an accessor for position (specified as a `Point3D`), accessors for constant,

linear, and quadratic attenuation factors, as well as accessors for its ambient, diffuse, and specular fields. Instantiating a red positional light at position (5, 2, -3), along with modest attenuation factors, in graphicslib3D can therefore be done thusly (the attenuation factors are described below the code):

```
PositionalLight pl1 = new PositionalLight();
Point3D plocation = new Point3D(5, 2, -3);
float [] amb = new float[ ] { 1.0f, 0.0f, 0.0f, 1.0f };
float [] dif = new float[ ] { 1.0f, 0.0f, 0.0f, 1.0f };
float [] spec = new float[ ] { 1.0f, 0.0f, 0.0f, 1.0f };
pl1.setAmbient(amb);
pl1.setDiffuse(dif);
pl1.setSpecular(spec);
pl1.setPosition(plocation);
pl1.setConstantAtt(1.0f);
pl1.setLinearAtt(0.1f);
pl1.setQuadraticAtt(0.1f);
```

Attenuation factors can be modeled in a variety of ways. As shown above, we provide tunable non-negative parameters for *constant*, *linear*, and *quadratic* attenuation (k_c , k_l , and k_q , respectively). These parameters are then combined, taking into account the distance (d) from the light source:

$$\text{attenuationFactor} = \frac{1}{k_c + k_l d + k_q d^2}$$

Multiplying this factor by the light intensity causes the intensity to be decreased the greater the distance is to the light source. Note that k_c should always be set greater than or equal to 1, so that the attenuation factor will always be in the range (0...1) and approach 0 as the distance d increases.

Spotlights have both a position and a direction. The effect of the spotlight’s “cone” can be simulated by specifying a *cutoff angle* θ between 0° and 90° indicating the width of the light beam, and a *falloff exponent* to model the variation of intensity across the angle of the beam. As shown in Figure 7.2, we determine the angle ϕ between the spotlight’s direction and a vector from the spotlight to the pixel. We then compute an *intensity factor* by raising the cosine of ϕ to the falloff exponent when ϕ is less than θ (when ϕ is greater than θ the intensity factor is set to 0). The result is an intensity factor that ranges from 0 to 1. The falloff exponent adjusts the rate at which the intensity factor tends to 0 as the

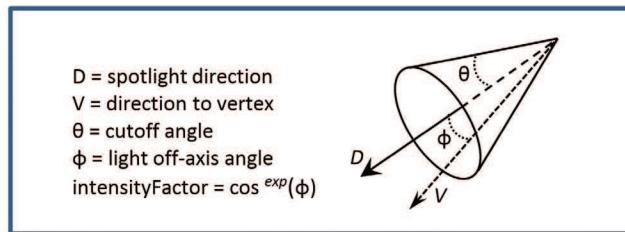


Figure 7.2
 Spotlight components.

angle ϕ increases. The intensity factor is then multiplied by the light's intensity to simulate the cone effect.

A red spotlight at location (5,2,-3) pointing down the negative Z axis could be specified thusly:

```
float[] spotLightAmbient = new float[] {1.0f, 0.0f, 0.0f, 1.0f};
float[] spotLightDiffuse = new float[] {1.0f, 0.0f, 0.0f, 1.0f };
float[] spotLightSpecular = new float[] {1.0f, 0.0f, 0.0f, 1.0f};
float[] spotLightLocation = new float[] {5.0f, 2.0f, -3.0f };
float[] spotLightDirection = new float[] {0.0f, 0.0f, -1.0f };
float[] spotLightCutoff = 45.0f;
float[] spotLightExponent = 2.0f;
```

Because spotlights share characteristics of both directional and positional lights, it is a bit tricky settling on a class hierarchy given Java's absence of multiple inheritance. In `graphicslib3D`, `SpotLight` is a subclass of `PositionalLight`. In addition to *direction*, it also includes accessors for *cutoff* and *exponent*. Instantiating the red spotlight described above can be done as follows:

```
SpotLight sl1 = new SpotLight();
Point3D plocation = new Point3D(5, 2, -3);
Vector3D directionVector = new Vector3D(0, 0, -1);
float[] amb = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] dif = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
float[] spec = new float[] { 1.0f, 0.0f, 0.0f, 1.0f };
sl1.setAmbient(amb);
sl1.setDiffuse(dif);
sl1.setSpecular(spec);
sl1.setPosition(plocation);
sl1.setDirection(directionVector);
```

```
sl1.setCutoffAngle(45.0f);
sl1.setFalloffExponent(2.0f);
```

Spotlights also can include attenuation factors in the same manner as positional lights. The `SpotLight` class in `graphicslib3D` inherits these attributes from `PositionalLight`, so they are available if desired.

Historically, spotlights have been iconic in computer graphics since Pixar's popular animated short "Luxo Jr." appeared in 1986 [DL16].

7.3 MATERIALS

The "look" of the objects in our scene has so far been handled exclusively by color and texture. The addition of lighting allows us to also consider the *reflectance* characteristics of the surfaces. By that, we mean how the object interacts with our ADS lighting model. This can be modeled by considering each object to be "made of" a certain *material*.

Materials can be simulated in an ADS lighting model by specifying four values, three of which we are already familiar with—*ambient*, *diffuse*, and *specular* RGB colors. The fourth is called *shininess*, which, as we will see, is used to build an appropriate specular highlight for the intended material. ADS and shininess values have been developed for many different types of common materials. For example, "pewter" can be specified thusly:

```
float[ ] pewterMatAmbient = new float[ ] { .11f, .06f, .11f, 1.0f };
float[ ] pewterMatDiffuse = new float[ ] { .43f, .47f, .54f, 1.0f };
float[ ] pewterMatSpecular = new float[ ] { .33f, .33f, .52f, 1.0f };
float pewterMatShininess = 9.85f;
```

ADS RGBA values for a few other materials are given in Figure 7.3 (from [BA16]).

Sometimes other properties are included in the material properties. *Transparency* is handled in the RGBA specifications in the fourth (or "alpha") channel, which specifies an opacity; a value of 1.0 represents completely opaque and 0.0 represents completely transparent. For most materials it is simply set to 1.0, although for certain materials a slight transparency plays a role. For example, in Figure 7.3, note that the materials "jade" and "pearl" include a small amount of transparency (values slightly less than 1.0) to add realism.

Emissiveness is also sometimes included in an ADS material specification. This is useful when simulating a material that emits its own light, such as phosphorescent materials.

<i>material</i>	<i>ambient RGBA</i> <i>diffuse RGBA</i> <i>specular RGBA</i>	<i>shininess</i>
Gold	0.24725, 0.1995, 0.0745, 1.0 0.75164, 0.60648, 0.22648, 1.0 0.62828, 0.5558, 0.36607, 1.0	51.2
Jade	0.135, 0.2225, 0.1575, 0.95 0.54, 0.89, 0.63, 0.95 0.3162, 0.3162, 0.3162, 0.95	12.8
Pearl	0.25, 0.20725, 0.20725, 0.922 1.00, 0.829, 0.829, 0.922 0.2966, 0.2966, 0.2966, 0.922	11.264
Silver	0.19225, 0.19225, 0.19225, 1.0 0.50754, 0.50754, 0.50754, 1.0 0.50827, 0.50827, 0.50827, 1.0	51.2

Figure 7.3
Material ADS coefficients.

In graphicslib3D, the class Material includes accessors for ambient, diffuse, and specular RGBA values (as float arrays), as well as for shininess and emission. Instantiating a “jade” material could therefore be done as follows:

```
Material jd1 = new Material();
float [] amb = new float[] { .135f, .2225f, .1575f, .95f };
float [] dif = new float[] { .54f, .89f, .63f, .95f };
float [] spec = new float[] { .3162f, .3162f, .3162f, .95f };
jd1.setAmbient(amb);
jd1.setDiffuse(dif);
jd1.setSpecular(spec);
jd1.setShininess(12.8f);
```

In addition, the Material class includes three predefined static materials: GOLD, SILVER, and BRONZE. For example, silver can be accessed with the following: graphicslib3D.Material.SILVER.

Note that the *graphicslib3D* light and materials classes described in these sections do not actually perform lighting. They merely provide an organized way to specify and store desired light and material properties for elements in a scene. We still need to actually compute the lighting ourselves. This is going to require some serious mathematical processing in our shader code. So let's now dive into the nuts-and-bolts of implementing ADS lighting in our Java/JOGL and GLSL graphics programs.

7.4 ADS LIGHTING COMPUTATIONS

As we draw our scene, recall that each vertex is transformed so as to simulate a 3D world on a 2D screen. Pixel colors are the result of rasterization, as well as texturing and interpolation. We must now incorporate the additional step of adjusting those rasterized pixel colors to effect the lighting and materials in our scene. The basic ADS computation that we need to perform is to determine the reflection intensity (I) for each pixel in our scene. This computation takes the following form:

$$I_{\text{observed}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

That is, we need to compute and sum the ambient, diffuse, and specular reflection contributions for each pixel, for each light source. This will of course depend on the type of light(s) in our scene, and the type of material associated with the rendered model.

Ambient contribution is the simplest. It is the product of the specified ambient light and the specified ambient coefficient of the material:

$$I_{\text{ambient}} = \text{Light}_{\text{ambient}} * \text{Material}_{\text{ambient}}$$

Keeping in mind that light and material intensities are specified via RGB values, the computation is more precisely:

$$I_{\text{ambient}}^{\text{red}} = \text{Light}_{\text{ambient}}^{\text{red}} * \text{Material}_{\text{ambient}}^{\text{red}}$$

$$I_{\text{ambient}}^{\text{green}} = \text{Light}_{\text{ambient}}^{\text{green}} * \text{Material}_{\text{ambient}}^{\text{green}}$$

$$I_{\text{ambient}}^{\text{blue}} = \text{Light}_{\text{ambient}}^{\text{blue}} * \text{Material}_{\text{ambient}}^{\text{blue}}$$

Diffuse contribution is more complex because it depends on the angle of incidence between the light and the surface. Lambert's Cosine Law (published in 1760) specifies that the amount of light that reflects from a surface is proportional to the cosine of the light's angle of incidence. This can be modeled as follows:

$$I_{\text{diffuse}} = \text{Light}_{\text{diffuse}} * \text{Material}_{\text{diffuse}} * \cos(\theta)$$

As before, the actual computations involve red, green, and blue components.

Determining the angle of incidence θ requires us to (a) find a vector from the pixel being drawn to the light source (or, similarly, a vector opposite the light direction), and (b) find a vector that is normal (perpendicular) to the surface of the object being rendered. Let's denote these vectors \vec{L} and \vec{N} , respectively, as shown below in Figure 7.4:

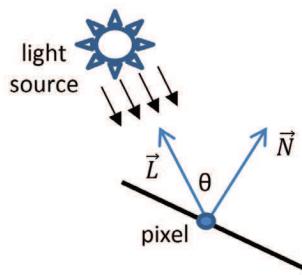


Figure 7.4
Angle of light incidence.

Depending on the nature of the lights in the scene, \vec{L} could be computed by negating the light direction vector, or by computing a vector from the pixel to the location of the light source. Determining vector \vec{N} may be trickier—normal vectors may be available for the vertices in the model being rendered, but if the model doesn't include normals, \vec{N} would need to be estimated geometrically based on the locations of neighboring vertices. For the rest of the chapter, we will assume that the model being rendered includes normal vectors for each vertex (this is common in models constructed in modeling tools such as Maya or Blender).

It turns out that in this case, it isn't necessary to compute θ itself. What we really desire is $\cos(\theta)$, and recall from Chapter 3 that this can be found using the *dot product*. Thus, the diffuse contribution can be computed as follows:

$$I_{\text{diffuse}} = \text{Light}_{\text{diffuse}} * \text{Material}_{\text{diffuse}} * (\hat{N} \bullet \hat{L})$$

The diffuse contribution is only relevant when the surface is exposed to the light, which occurs when $-90 < \theta < 90$; that is, when $\cos(\theta) > 0$. Thus we must replace the rightmost term above with:

$$\max((\hat{N} \bullet \hat{L}), 0)$$

Specular contribution determines whether the pixel being rendered should be brightened because it is part of a “specular highlight.” It involves not only the angle of incidence of the light source, but also the angle between the reflection of the light on the surface, and the viewing angle of the “eye” relative to the object’s surface:

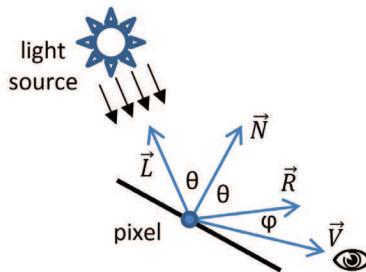


Figure 7.5
View angle incidence.

In Figure 7.5, \vec{R} represents the direction of reflection of the light, and \vec{V} (called the *view vector*) is a vector from the pixel to the eye. Note that \vec{V} is the negative of the vector from the eye to the pixel (in camera space, the eye is at the origin). The smaller the angle φ between \vec{R} and \vec{V} , the more the eye is on-axis, or “looking into” the reflection, and the more this pixel contributes to the specular highlight (and thus the brighter it should appear).

The manner in which φ is used to compute the specular contribution depends on the desired “shininess” of the object being rendered. Objects that are extremely shiny, such as a mirror, have very small specular highlights—that is, they reflect the incoming light to the eye exactly. Materials that are less shiny have specular highlights that are more “spread out,” and thus more pixels are a part of the highlight.

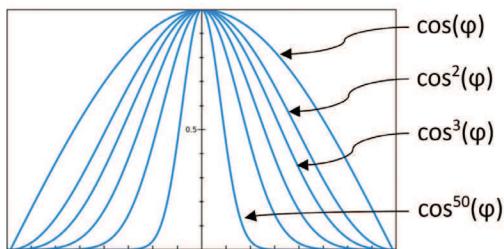


Figure 7.6
Shininess modeled as cosine exponent.

Shininess is generally modeled with a *falloff* function that expresses how quickly the specular contribution reduces to zero as the angle φ grows. We can use $\cos(\varphi)$ to model falloff, and increase or decrease the shininess by using powers of the cosine function, such as $\cos(\varphi)$, $\cos^2(\varphi)$, $\cos^3(\varphi)$, $\cos^{10}(\varphi)$, $\cos^{50}(\varphi)$, and so on, as illustrated in Figure 7.6.

Note that the higher the value of the exponent, the faster the falloff, and thus the smaller the specular contribution of pixels with light reflections that are off-axis from the viewing angle.

We call the exponent n , as used in the $\cos^n(\varphi)$ falloff function, the *shininess factor* for a specified material. Note back in Figure 7.3 that shininess factors for each of the materials listed is specified in the rightmost column.

We now can specify the full specular calculation:

$$I_{\text{spec}} = \text{Light}_{\text{spec}} * \text{Material}_{\text{spec}} * \max(0, (\hat{R} \bullet \hat{V})^n)$$

Note that we use the `max()` function in a similar manner as we did for the diffuse computation. In this case, we need to ensure that the specular contribution does not ever utilize negative values for $\cos(\varphi)$, which could produce strange artifacts such as “darkened” specular highlights.

And of course as before, the actual computations involve red, green, and blue components.

7.5 IMPLEMENTING ADS LIGHTING

The computations described in Section 7.4 have so far been mostly theoretical, as they have assumed that we can perform them for every pixel. This is complicated by the fact that normal (\vec{N}) vectors are typically available to us only for the vertices that define the models, not for each pixel. Thus we need to either compute normals for each pixel, which could be time-consuming, or find some way of estimating the values that we need to achieve a sufficient effect.

One approach is called “faceted shading” or “flat shading.” Here we assume that every pixel in each rendered primitive (i.e., polygon or triangle) has the same lighting value. Thus we need only do the lighting computations for one vertex in each polygon in the model, and then copy those lighting values across the nearby rendered pixels on a per-polygon or per-triangle basis.

Faceted shading is rarely used today, because the resulting images tend to not look very realistic, and because modern hardware makes more accurate computations feasible. An example of a faceted-shaded torus, in which each triangle behaves as a flat reflective surface, is shown in Figure 7.7.

Although faceted shading can be adequate in some circumstances, usually a better approach is “smooth shading,” in which the lighting intensity is computed for each pixel. Smooth shading is feasible because of the parallel processing done on modern graphics cards, and because of the interpolated rendering that takes place in the OpenGL graphics pipeline.

We will examine two popular methods for smooth shading: *Gouraud shading* and *Phong shading*.

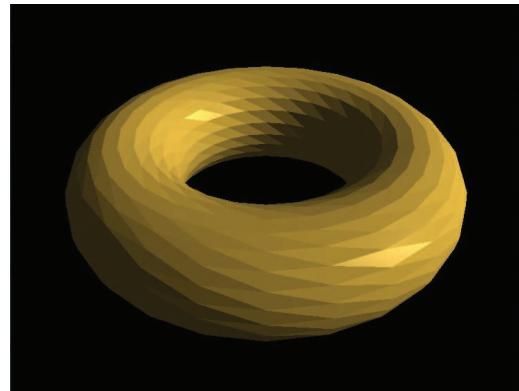


Figure 7.7
Torus with faceted shading.

7.5.1 Gouraud Shading

The French computer scientist Henri Gouraud published a smooth shading algorithm in 1971 that has come to be known as *Gouraud shading* [GO71]. It is particularly well suited to modern graphics cards, because it takes advantage of the automatic interpolated rendering that is available in 3D graphics pipelines such as in OpenGL. The process for Gouraud shading is as follows:

1. Determine the color of each *vertex*, incorporating the lighting computations.
2. Allow those colors to be interpolated across intervening pixels through the normal rasterization process (which will also in effect interpolate the lighting contributions).

In OpenGL/Jogl, this means that the lighting computations will be done in the vertex shader. The fragment shader will simply be a pass-through, so as to reveal the automatically interpolated lighted colors.

Figure 7.8 outlines the strategy we will use to implement our Gouraud shader in OpenGL, for a scene with a torus and one positional light. The strategy is then implemented in Program 7.1.

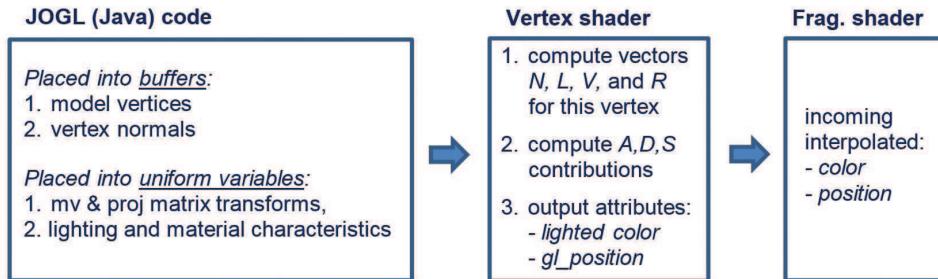


Figure 7.8

Implementing Gouraud shading.

Program 7.1 Torus with Positional Light and Gouraud Shading

```

// previous imports still apply – the following additional imports will be necessary:
import graphicslib3D.*;
import graphicslib3D.light.*;

public class Code extends JFrame implements GLEventListener
{
    private GLCanvas myCanvas;
    // creation of m, v, mv, and proj matrices as before.
    // declarations for building shaders and rendering program, as before.
    // declaration of one VAO, and two VBOs, as before.
    ...
    private Torus myTorus = new Torus(0.5f, 0.2f, 48);
    private int numTorusVertices;

    // light and material declarations
    private Material currentMaterial = Material.GOLD;
    private PositionalLight currentLight = new PositionalLight();
    private Point3D lightLoc = new Point3D(10.0f, 10.0f, 10.0f);
    private float [] globalAmbient = new float[] { 0.7f, 0.7f, 0.7f, 1.0f };
}

```

```
// location of torus and camera
private Point3D torusLoc = new Point3D(0,0,-1);
private Point3D cameraLoc = new Point3D(0,0,1.0f);

public Code()
{ // constructor same as before
}

public void init(GLAutoDrawable drawable)
{ GL4 gl = (GL4) drawable.getGL();
    rendering_program = createShaderProgram();
    setupVertices();
}

private void setupVertices()
{ // code to extract graphicslib3D torus indices, vertices and normals, as described before.
    ...
    // code to generate VAO, bind vertex array 0, and generate two buffers, as described before.
    ...
    numTorusVertices = indices.length;
    ...
    // put the Torus vertices into the first buffer
    gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    FloatBuffer vertBuf = Buffers.newDirectFloatBuffer(fvalues);
    gl.glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4, vertBuf, GL_STATIC_DRAW);

    // load the torus normal vectors into the second buffer
    gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    FloatBuffer norBuf = Buffers.newDirectFloatBuffer(nvalues);
    gl.glBufferData(GL_ARRAY_BUFFER, norBuf.limit()*4, norBuf, GL_STATIC_DRAW);
}

// main(), dispose(), reshape(), shader modules, and perspective computations as before.

public void display(GLAutoDrawable drawable)
{ // setup of GL4 object, clearing of depth buffer as in earlier examples.
    ...
    // setup of projection matrix as in earlier examples.
    ...
    gl.glUseProgram(rendering_program);

    // uniforms for model-view, projection, and normal matrices.
    mv_location = gl.glGetUniformLocation(rendering_program, "mv_matrix");
    proj_location = gl.glGetUniformLocation(rendering_program, "proj_matrix");
    n_location = gl.glGetUniformLocation(rendering_program, "norm_matrix");
```

```
// build the MODEL matrix based on the torus location
m_matrix.setTIdentity();
m_matrix.translate(torusLoc.getX(),torusLoc.getY(),torusLoc.getZ());
m_matrix.rotateX(35.0);

// build the VIEW matrix based on the camera location
v_matrix.setTIdentity();
v_matrix.translate(-cameraLoc.getX(),-cameraLoc.getY(),-cameraLoc.getZ());

// set up lights based on the current light's position
currentLight.setPosition(lightLoc);
installLights(v_matrix);

// build the MODEL-VIEW matrix by concatenating matrices v and m, as before
...
// put the MV, PROJ, and NORMAL matrices into the corresponding uniform variables
gl.glUniformMatrix4fv(mv_location, 1, false, mv_matrix.getFloatValues(), 0);
gl.glUniformMatrix4fv(proj_location, 1, false, proj_matrix.getFloatValues(), 0);
// NOTE -- the transformation matrix for the normal vector is the inverse transpose of MV.
gl.glUniformMatrix4fv(n_location, 1, false, (mv_matrix.inverse()).transpose().getFloatValues(), 0);

// bind the vertices buffer to vertex attribute #0 in the vertex shader
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

// bind the normals buffer to vertex attribute #1 in the vertex shader
gl glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
gl glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(1);

gl glClear(GL_DEPTH_BUFFER_BIT);
gl glEnable(GL_CULL_FACE);
gl glFrontFace(GL_CCW);
gl glEnable(GL_DEPTH_TEST);
gl glDepthFunc(GL_LEQUAL);

gl glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);
}

private void installLights(Matrix3D v_matrix, GL4)
{
    GL4 gl = (GL4) GLContext.getCurrentGL();

    // convert light's position to view space, and save it in a float array
```

```

Point3D lightP = currentLight.getPosition();
Point3D lightPv = lightP.mult(v_matrix);
float [ ] viewspaceLightPos =
    new float[ ] { (float) lightPv.getX(), (float) lightPv.getY(), (float) lightPv.getZ() };

// set the current globalAmbient settings
int globalAmbLoc = gl.glGetUniformLocation(rendering_program, "globalAmbient");
gl.glProgramUniform4fv(rendering_program, globalAmbLoc, 1, globalAmbient, 0);

// get the locations of the light and material fields in the shader
int ambLoc = gl.glGetUniformLocation(rendering_program, "light.ambient");
int diffLoc = gl.glGetUniformLocation(rendering_program, "light.diffuse");
int specLoc = gl.glGetUniformLocation(rendering_program, "light.specular");
int posLoc = gl.glGetUniformLocation(rendering_program, "light.position");
int MambLoc = gl.glGetUniformLocation(rendering_program, "material.ambient");
int MdiffLoc = gl.glGetUniformLocation(rendering_program, "material.diffuse");
int MspecLoc = gl.glGetUniformLocation(rendering_program, "material.specular");
int MshiLoc = gl.glGetUniformLocation(rendering_program, "material.shininess");

// set the uniform light and material values in the shader
gl.glProgramUniform4fv(rendering_program, ambLoc, 1, currentLight.getAmbient(), 0);
gl.glProgramUniform4fv(rendering_program, diffLoc, 1, currentLight.getDiffuse(), 0);
gl.glProgramUniform4fv(rendering_program, specLoc, 1, currentLight.getSpecular(), 0);
gl.glProgramUniform3fv(rendering_program, posLoc, 1, viewspaceLightPos, 0);
gl.glProgramUniform4fv(rendering_program, MambLoc, 1, currentMaterial.getAmbient(), 0);
gl.glProgramUniform4fv(rendering_program, MdiffLoc, 1, currentMaterial.getDiffuse(), 0);
gl.glProgramUniform4fv(rendering_program, MspecLoc, 1, currentMaterial.getSpecular(), 0);
gl.glProgramUniform1f(rendering_program, MshiLoc, currentMaterial.getShininess());
} }

```

Most of the elements of Program 7.1 should be familiar. The `graphicslib3D` classes for `Material`, `Torus`, and `PositionalLight` are instantiated and configured. `Torus` vertices and associated normals are loaded into buffers. The `display()` function is similar to that in previous programs, except that it also sends the light and torus locations to the vertex shader. The `installLights()` function loads the light and material ADS characteristics into corresponding uniform variables to make them available to the shader code.

One interesting detail is that the transformation matrix `MV`, used to move vertex positions into view space, doesn't always properly adjust *normal vectors* into view space. This is because simply applying the `MV` matrix to the normal doesn't guarantee that they will remain perpendicular to the object surface. The correct transformation matrix is equal to the *inverse transpose* of `MV`, as described earlier

in the supplemental notes to Chapter 3. In Program 7.1, this additional matrix, named “norm_matrix,” is sent to the shaders in a uniform variable.

The variable `lightPv` contains the light’s position in camera space. We only need to compute this once per frame, so we do it `installLights()` (called from `display()`) rather than in the shader.

The shaders are shown ahead. The vertex shader utilizes some notations that we haven’t yet seen. Note for example the vector addition done at the end of the vertex shader—vector addition was described in Chapter 3, and is available as shown here in GLSL. We will discuss some of the other notations after presenting the shaders.

(Program 7.1, continued)

Vertex Shader

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec4 varyingColor;

struct PositionalLight
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec3 position;
};

struct Material
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};

uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix; // for transforming normals

void main(void)
{
    vec4 color;

    // convert vertex position to view space
    vec4 P = mv_matrix * vec4(vertPos,1.0);
```

```

// convert normal to view space
vec3 N = normalize((norm_matrix * vec4(vertNormal,1.0)).xyz);

// calculate view space light vector (from vertex to light)
vec3 L = normalize(light.position - P.xyz);

// view vector is equivalent to the negative of view space vertex position
vec3 V = normalize(-P.xyz);

// R is reflection of -L with respect to surface normal N
vec3 R = reflect(-L,N);

// ambient, diffuse, and specular contributions
vec3 ambient = ((globalAmbient * material.ambient) + (light.ambient * material.ambient)).xyz;
vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(dot(N,L), 0.0);
vec3 specular =
    material.specular.xyz * light.specular.xyz * pow(max(dot(R,V), 0.0f), material.shininess);

// send the color output to the fragment shader
varyingColor = vec4((ambient + diffuse + specular), 1.0);

// send the position to the fragment shader, as before
gl_Position = proj_matrix * mv_matrix * vec4(vertPos,1.0);
}

```

Fragment Shader

```

#version 430
in vec4 varyingColor;
out vec4 fragColor;

// uniforms match those in the vertex shader,
// but aren't used directly in this fragment shader
struct PositionalLight
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec3 position;
};

struct Material
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};

uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;

```

```

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;

// interpolate lighted color (interpolation of gl_Position is automatic)
void main(void)
{
    fragColor = varyingColor;
}

```

The output of Program 7.1 is shown in Figure 7.9.

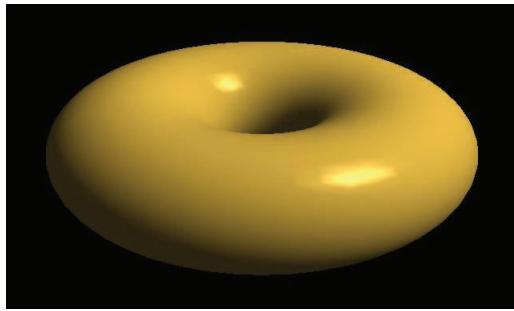


Figure 7.9
Torus with Gouraud Shading.

The vertex shader contains our first example of using the struct notation. A GLSL “struct” is like a datatype; it has a name, and a set of fields. When a variable is declared using the name of a struct, it then contains those fields, which are accessed using the “.” notation. For example, variable “light” is declared of type “PositionalLight”, so we can thereafter refer to its fields `light.ambient`, `light.diffuse`, and so forth.

Also note the field selector notation “`.xyz`”, used in several places in the vertex shader. This is a shortcut for converting a `vec4` to an equivalent `vec3` containing only its first three elements.

The vertex shader is where most of the lighting computations are performed. For each vertex, the appropriate matrix transforms are applied to the vertex position and associated normal vector, and vectors for light direction (\vec{L}) and reflection (\vec{R}) are computed. The ADS computations described in section 7.4 are then performed, resulting in a color for each vertex (called `varyingColor` in the code). The colors are interpolated as part of the normal rasterization process. The fragment shader is then a simple pass-through. The lengthy list of uniform variable declarations is also present in the fragment shader, but none of them are actually used there.

Note the use of the GLSL functions `normalize()`, which converts a vector to unit length and is necessary for proper application of the dot product, and `reflect()`, which computes the reflection of one vector about another.

Artifacts are evident in the output torus shown in Figure 7.9. Specular highlights have a blocky, faceted appearance. This artifact is more pronounced if the object is in motion (we can't illustrate that here).

Gouraud shading is susceptible to other artifacts. If the specular highlight is entirely contained within one of the model's triangles—that is, if it doesn't contain at least one of the model vertices—then it may disappear entirely. The specular component is calculated per vertex, so if a model vertex with a specular contribution does not exist, none of the rasterized pixels will include specular light either.

7.5.2 Phong Shading

Bui Tuong Phong developed a smooth shading algorithm while a graduate student at the University of Utah, and described it in his 1973 dissertation [PH73] and published in [PH75]. The structure of the algorithm is similar to the algorithm for Gouraud shading, except that the lighting computations are done *per-pixel* rather than per-vertex. Since the lighting computations require a normal vector \vec{N} and a light vector \vec{L} , which are only available in the model on a per-vertex basis, Phong shading is often implemented using a clever “trick,” whereby \vec{N} and \vec{L} are computed in the vertex shader and *interpolated* during rasterization. Figure 7.10 outlines the strategy:

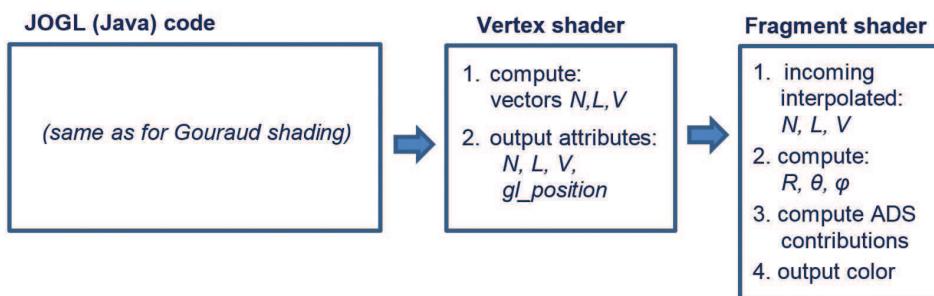


Figure 7.10

Implementing Phong shading.

The Java/JOGL code is completely unchanged. Some of the computations previously done in the vertex shader are now moved into the fragment shader. The effect of interpolating normal vectors is illustrated in Figure 7.11:

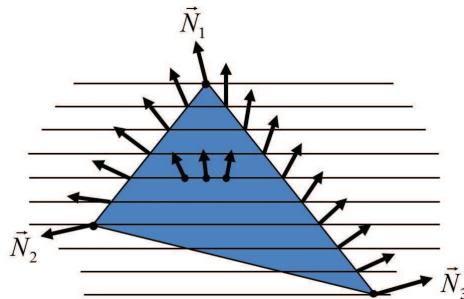


Figure 7.11
Interpolation of normal vectors.

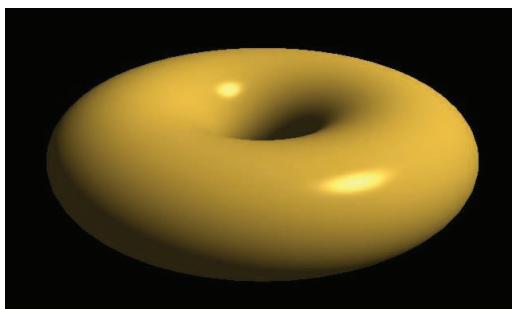


Figure 7.12
Torus with Phong shading.

We now are ready to implement our torus with positional lighting, using Phong shading. Most of the code is identical to that used for Gouraud shading. Since the Java/JOGL code is unchanged, we present only the revised vertex and fragment shaders, shown in Program 7.2. Examining the output of Program 7.2, as shown in Figure 7.12, Phong shading corrects the artifacts present in Gouraud shading.

Program 7.2 Torus with Phong Shading

Vertex Shader

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec3 varyingNormal;      // eye-space vertex normal
out vec3 varyingLightDir;    // vector pointing to the light
out vec3 varyingVertPos;     // vertex position in eye space
```

```
// structs and uniforms same as for Gouraud shading
...
void main(void)
{
    // output vertex position, light direction, and normal to the rasterizer for interpolation
    varyingVertPos=(mv_matrix * vec4(vertPos,1.0)).xyz;
    varyingLightDir = light.position - varyingVertPos;
    varyingNormal=(norm_matrix * vec4(vertNormal,1.0)).xyz;
    gl_Position=proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

Fragment Shader

```
#version 430
in vec3 varyingNormal;
in vec3 varyingLightDir;
in vec3 varyingVertPos;
out vec4 fragColor;
// structs and uniforms same as for Gouraud shading
...
void main(void)
{
    // normalize the light, normal, and view vectors:
    vec3 L = normalize(varyingLightDir);
    vec3 N = normalize(varyingNormal);
    vec3 V = normalize(-varyingVertPos);

    // compute light reflection vector with respect to N:
    vec3 R = normalize(reflect(-L, N));
    // get the angle between the light and surface normal:
    float cosTheta = dot(L,N);
    // angle between the view vector and reflected light:
    float cosPhi = dot(V,R);

    // compute ADS contributions (per pixel), and combine to build output color:
    vec3 ambient = ((globalAmbient * material.ambient) + (light.ambient * material.ambient)).xyz;
    vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(cosTheta,0.0);
    vec3 specular =
        light.specular.xyz * material.specular.xyz * pow(max(cosPhi,0.0), material.shininess);
    fragColor = vec4((ambient + diffuse + specular), 1.0);
}
```

Although Phong shading offers better realism than Gouraud shading, it does so while incurring a performance cost. One optimization to Phong shading was

proposed by James Blinn in 1977 [BL77], and is referred to as the *Blinn-Phong* reflection model. It is based on the observation that one of the most expensive computations in Phong shading is determining the reflection vector \vec{R} .

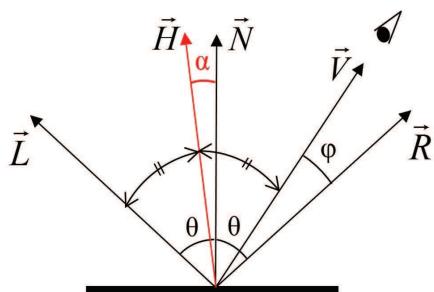


Figure 7.13
Blinn-Phong reflection.

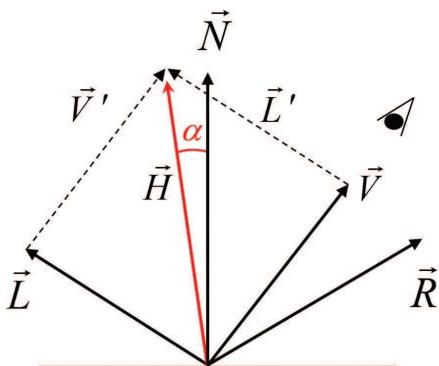


Figure 7.14
Blinn-Phong computation.

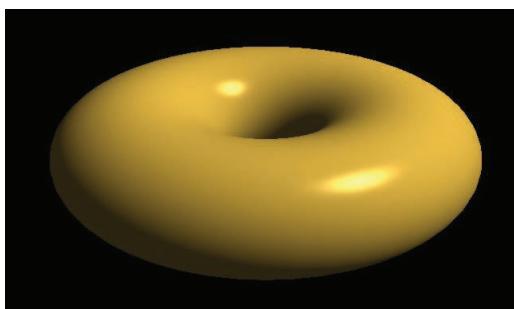


Figure 7.15
Torus with Blinn-Phong shading.

Blinn observed that the vector \vec{R} itself actually is not needed— \vec{R} is only produced as a means of determining the angle φ . It turns out that φ can be found without computing \vec{R} , by instead computing a vector \vec{H} that is halfway between \vec{L} and \vec{V} . As shown in Figure 7.13, the angle α between \vec{H} and \vec{N} is conveniently equal to $\frac{1}{2}(\varphi)$. Although α isn't identical to φ , Blinn showed that reasonable results can be obtained by using α instead of φ .

The “halfway” vector \vec{H} is most easily determined by finding $\vec{L} + \vec{V}$ (see Figure 7.14), after which $\cos(\alpha)$ can be found using the dot product $\hat{H} \bullet \hat{N}$.

The computations can be done in the fragment shader, or even in the vertex shader (with some tweaks) if necessary for performance. Figure 7.15 shows the torus rendered using Blinn-Phong shading; the quality is largely indistinguishable from Phong shading, with substantial performance cost savings.

Program 7.3 shows the revised vertex and fragment shaders for converting the Phong shading example shown in Program 7.2 to Blinn-Phong shading. As before, there is no change to the Java/JOGl code.

Program 7.3 Torus w/ Blinn-Phong Shading

Vertex Shader

```
...
// half-vector "H" is an additional output varying
out vec3 varyingHalfVector;
...
void main(void)
{
    // computations same as before, plus the following that computes L+V
    varyingHalfVector = (varyingLightDir + (-varyingVertPos)).xyz;
    // (the rest of the vertex shader is unchanged)
}
```

Fragment Shader

```
...
in vec3 varyingHalfVector;
...
void main(void)
{
    // note that it is no longer necessary to compute R in the fragment shader
    vec3 L = normalize(varyingLightDir);
    vec3 N = normalize(varyingNormal);
    vec3 V = normalize(-varyingVertPos);
    vec3 H = normalize(varyingHalfVector);
    ...
    // get angle between the normal and the halfway vector
    float cosPhi = dot(H,N);
    ...
    // halfway vector H was computed in the vertex shader, and then interpolated.
    vec3 ambient = ((globalAmbient * material.ambient) + (light.ambient * material.ambient)).xyz;
    vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(cosTheta,0.0);
    vec3 specular =
        light.specular.xyz * material.specular.xyz * pow(max(cosPhi,0.0), material.shininess*3.0);
        // the multiplication by 3.0 at the end is a "tweak" to improve the specular highlight.
    fragColor = vec4((ambient + diffuse + specular), 1.0);
}
```

Figure 7.16 shows two examples of the effect of Phong shading on more complex externally generated models. The top image shows a rendering of an OBJ model of a dolphin created by Jay Turberville at Studio 522 Productions [TU16]. The bottom image is a rendering of the well-known “Stanford Dragon,” the result of a 3D scan of an actual figurine, done in 1996 [ST96]. Both models were

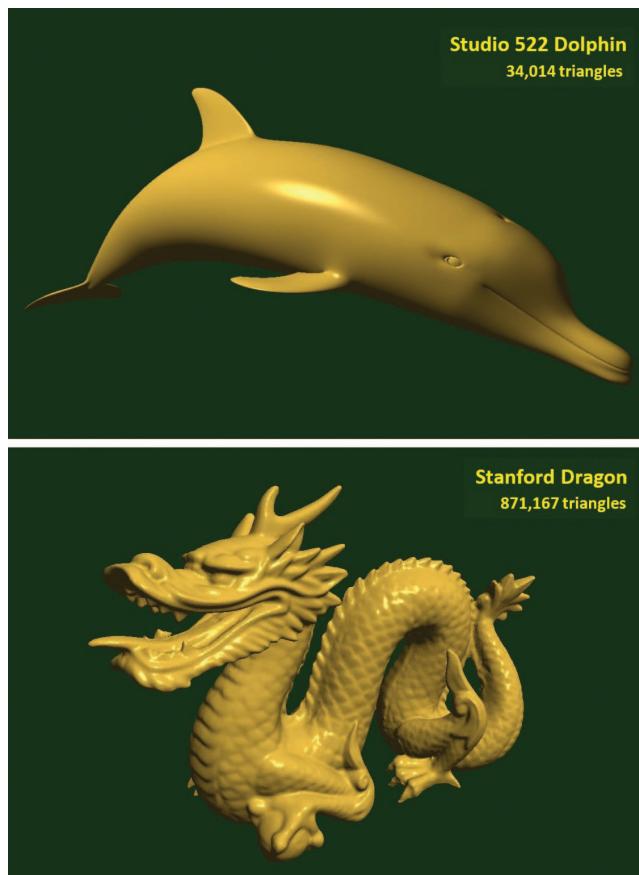


Figure 7.16
External models with Phong shading.

rendered using the “gold” material from `graphicslib3D`. The Stanford dragon is widely used for testing graphics algorithms and hardware because of its size—it contains over 800,000 triangles.

7.6 COMBINING LIGHTING AND TEXTURES

So far, our lighting model has assumed that we are using lights with specified ADS values to illuminate objects made of material that has also been defined with ADS values. However, as we saw in Chapter 5, some objects may instead have surfaces defined by texture images. Therefore, we need a way of

combining colors retrieved by sampling a texture, and colors produced from a lighting model.

The manner in which we combine lighting and textures depends on the nature of the object and the purpose of its texture. There are several scenarios, a few of which include:

- The texture image very closely reflects the actual appearance of the object’s surface.
- The object has both a material and a texture.
- The texture contains shadow or reflection information (covered in Chapters 8 and 10).
- There are multiple lights, and/or multiple textures involved.

Let’s consider the first case, where we have a simple textured object and we wish to add lighting to it. One simple way of accomplishing this in the fragment shader is to remove the material specification entirely, and to use the texel color returned from the texture sampler in place of the material ADS values. The following is one such strategy (expressed in pseudocode):

```
fragColor = textureColor * ( ambientLight + diffuseLight ) + specularLight
```

Here the texture color contributes to the ambient and diffuse computation, while the specular color is defined entirely by the light. It is common to set the specular contribution solely based on the light color, especially for metallic or “shiny” surfaces. However, some less shiny surfaces, such as cloth or unvarnished wood (and even a few metals, such as gold) have specular highlights that include the color of the object surface. In those cases, a suitable slightly modified strategy would be:

```
fragColor = textureColor * ( ambientLight + diffuseLight + specularLight)
```

There are also cases in which an object has an ADS material that is supplemented by a texture image, such as an object made of silver that has a texture that adds some tarnish to the surface. In those situations, the standard ADS model with both light and material, as described in previous sections, can be combined with the texture color using a weighted sum. For example:

```
textureColor = texture(sampler, texCoord)
lightColor = (ambLight * ambMaterial) + (diffLight * diffMaterial) + specLight
fragColor = 0.5 * textureColor + 0.5 * lightColor
```

This strategy for combining lighting, materials, and textures can be extended to scenes involving multiple lights and/or multiple textures. For example:

```
texture1Color = texture(sampler1, texCoord)
texture2Color = texture(sampler2, texCoord)
light1Color = (ambLight1 * ambMaterial) + (diffLight1 * diffMaterial) + specLight1
light2Color = (ambLight2 * ambMaterial) + (diffLight2 * diffMaterial) + specLight2
fragColor = 0.25 * texture1Color
    + 0.25 * texture2Color
    + 0.25 * light1Color
    + 0.25 * light2Color
```

Figure 7.17 shows the Studio 522 Dolphin with a UV-mapped texture image (produced by Jay Turberville [TU16]), and the NASA shuttle model we saw earlier in Chapter 5. Both textured models are enhanced with Blinn-Phong lighting, without the inclusion of materials, and with specular highlights that utilize light only. In both cases, the relevant output color computation in the fragment shader is:

```
vec4 texColor = texture(sampler, texCoord);
fragColor = texColor * (globalAmbient + lightAmb + lightDiff * max(dot(L,N),0.0))
    + lightSpec * pow(max(dot(H,N),0.0), matShininess*3.0);
```

Note that it is possible for the computation that determines `fragColor` to produce values greater than 1.0. When that happens, OpenGL clamps the computed value to 1.0.

SUPPLEMENTAL NOTES

The faceted-shaded torus shown in Figure 7.7 was created by adding the “`flat`” interpolation qualifier to the corresponding normal vector vertex attribute declarations in the vertex and fragment shaders. This instructs the rasterizer to *not* perform interpolation on the specified variable, and instead assign the same value for each fragment (by default it chooses the value associated with the first vertex in the triangle). In the Phong shading example, this could be done as follows:

`flat out vec3 varyingNormal;`
`flat in vec3 varyingNormal;`

in the vertex shader, and
in the fragment shader.

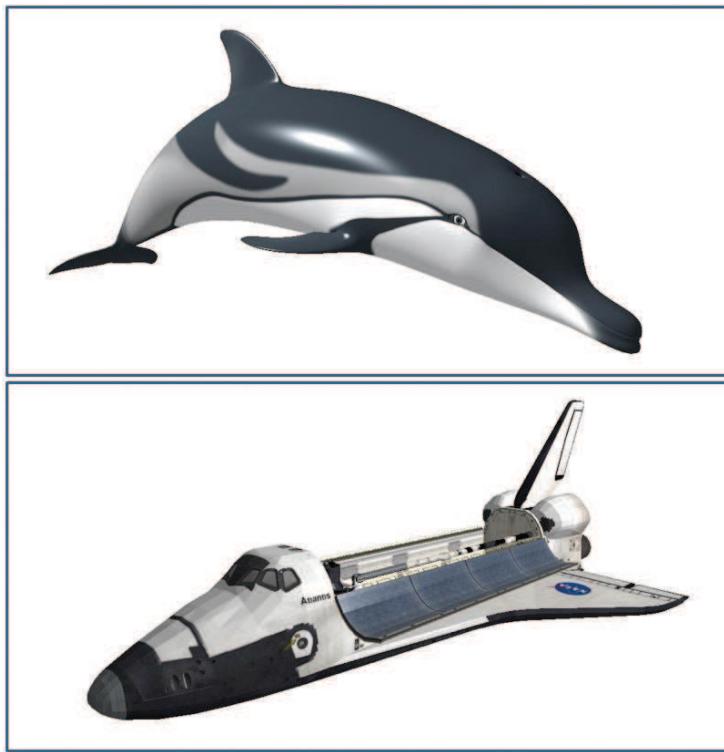


Figure 7.17
Combining lighting and textures.

An important kind of light source that we haven't discussed is a *distributed light* or *area light*, which is a light that is characterized by having a source that occupies an area rather than being a single point location. A real-world example would be a fluorescent tube-style light commonly found in an office or classroom. Interested readers can find more details about area lights in [MH02].

HISTORICAL NOTE

We took the liberty of over-simplifying some of the terminology in this chapter, with respect to the contributions of Gouraud and Phong. Gouraud is credited with *Gouraud shading*—the notion of generating a smoothly curved surface appearance by computing light intensities at vertices and allowing the rasterizer to interpolate these values (sometimes called “smooth shading”). Phong is credited

with *Phong shading*, another form of smooth shading that instead interpolates normals and computes lighting per-pixel. Phong is also credited with pioneering the successful incorporation of specular highlights into smooth shading. For this reason, the ADS lighting model, when applied to computer graphics, is often referred to as the *Phong Reflection Model*. So, our example of Gouraud shading is, more accurately, Gouraud shading with a Phong reflection model. Since Phong's reflection model has become so ubiquitous in 3D graphics programming, it is common to demonstrate Gouraud shading in the presence of Phong reflection, although it is a bit misleading because Gouraud's original 1971 work did not, for example, include any specular component.

Exercises

- 7.1 (*PROJECT*) Modify Program 7.1 so that the light can be positioned by moving the mouse. After doing this, move the mouse around and note the movement of the specular highlight, and the appearance of the Gouraud shading artifacts. You may find it convenient to render a point (or small object) at the location of the light source.
- 7.2 Repeat Exercise 7.1, but applied to Program 7.2. This should only require substituting the shaders for Phong shading into your solution to Exercise 7.1. The improvement from Gouraud to Phong shading should be even more apparent here, when the light is being moved around.
- 7.3 (*PROJECT*) Modify Program 7.2 so that it incorporates TWO positional lights, placed in different locations. The fragment shader will need to blend the diffuse and specular contributions of each of the lights. Try using a weighted sum, similar to the one shown in Section 7.6. You can also try simply adding them, and clamping the result so it doesn't exceed the maximum light value.
- 7.4 (*RESEARCH AND PROJECT*) Replace the positional light in Program 7.2 with a “spot” light, as described in Section 7.2.

References

- [BA16] N. Barradeu, accessed July 2016, <http://www.barradeau.com/nicoptere/dump/materials.html>.
- [BL77] J. Blinn, “Models of Light Reflection for Computer Synthesized Pictures,” *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, 1977.
- [DI16] Luxo Jr. (Pixar – copyright held by Disney), accessed July 2016, http://www.pixar.com/short_films/Theatrical-Shorts/Luxo-Jr.#.
- [GO71] H. Gouraud, “Continuous Shading of Curved Surfaces,” *IEEE Transactions on Computers* C-20, no. 6 (June 1971).
- [MH02] T. Akenine-Möller, and E. Haines, *Real-Time Rendering*, 2nd ed. (A. K. Peters, 2002).
- [PH73] B. Phong, “Illumination of Computer-Generated Images,” (PhD thesis, University of Utah, 1973).
- [PH75] B. Phong, “Illumination for Computer Generated Pictures,” *Communications of the ACM* 18, no. 6 (June 1975): 311–317.
- [ST96] Stanford Computer Graphics Laboratory, accessed July 2016, <http://graphics.stanford.edu/data/3Dscanrep/>.
- [TU16] J. Turberville, Studio 522 Productions, Scottsdale, AZ, www.studio522.com (dolphin model developed 2016).

SHADOWS

8.1	<i>The Importance of Shadows</i>	189
8.2	<i>Projective Shadows</i>	190
8.3	<i>Shadow Volumes</i>	191
8.4	<i>Shadow Mapping</i>	192
8.5	<i>A Shadow Mapping Example</i>	199
8.6	<i>Shadow Mapping Artifacts</i>	205
	<i>Supplemental Notes</i>	208



8.1 THE IMPORTANCE OF SHADOWS

In Chapter 7, we learned how to add lighting to our 3D scenes. However, we didn’t actually add light—instead, we simulated the effects of light on objects—using the ADS model—and modified how we drew those objects accordingly.

The limitations of this approach become apparent when we use it to light more than one object in the same scene. Consider the scene in Figure 8.1, which includes both our brick-textured torus and a ground plane (the ground plane is the top of a giant cube with a grass texture [LU16]).

At first glance our scene may appear reasonable. However, closer examination reveals that there is something very important missing. In particular, it is impossible to discern the distance between the torus and the large textured cube below it. Is the torus floating above the cube, or is it resting on top of the cube?



Figure 8.1
Scene without shadows.

The reason we cannot answer this question is due to the lack of *shadows* in the scene. We expect to see shadows, and our brain uses shadows to help build a more complete mental model of the objects we see and where they are located.

Consider the same scene, shown in Figure 8.2, with shadows incorporated. It is now obvious that the torus is resting on the ground plane in the left example, and floating above it in the right example.

8.2 PROJECTIVE SHADOWS

A variety of interesting methods have been devised for adding shadows to 3D scenes. One method that is well-suited to drawing shadows on a ground plane (such as our image in Figure 8.1), and relatively computationally inexpensive, is called *projective shadows*. Given a point light source position (X_L, Y_L, Z_L) , an object to render, and a plane on which the object's shadow is to be cast, it is possible to derive a transformation matrix that will convert points (X_w, Y_w, Z_w) on the object to corresponding shadow points $(X_s, 0, Z_s)$ on the plane. The resulting “shadow polygon” is then drawn, typically as a dark object blended with the texture on the ground plane, as illustrated in Figure 8.3.

The advantages of projective shadow casting are that it is efficient and simple to implement. However, it only works on a flat plane—the method can't be used to cast shadows on a curved surface or on other objects. It is still useful

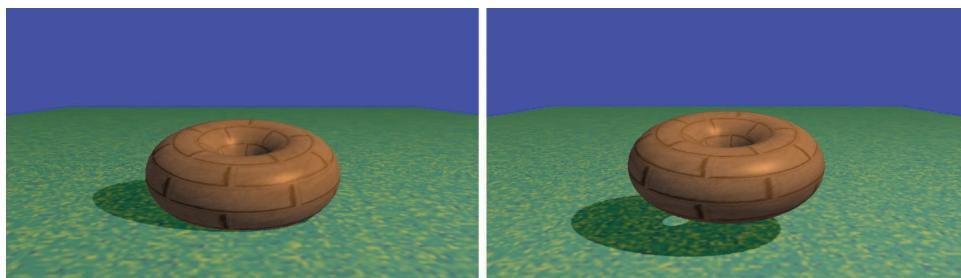


Figure 8.2
Lighting with shadows.

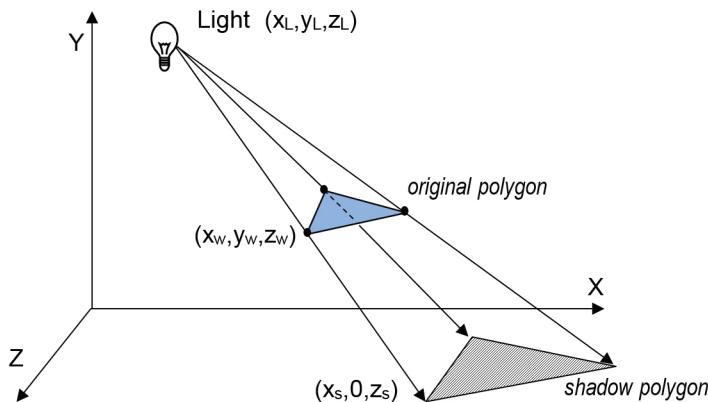


Figure 8.3
Projective shadow.

for performance-intensive applications involving outdoor scenes, such as in many video games.

Development of projective shadow transformation matrices is discussed in [BL88], [AS14], and [KS16].

8.3 SHADOW VOLUMES

Another important method, proposed by Crow in 1977, is to identify the spatial volume shadowed by an object, and reduce the color intensity of polygons inside the intersection of the shadow volume with the view volume [CR77]. Figure 8.4 shows a cube in a shadow volume, so the cube would be drawn darker.

Shadow volumes have the advantage of being highly accurate, with fewer artifacts than other methods. However, finding the shadow volume, and then computing whether each polygon is inside of it, is

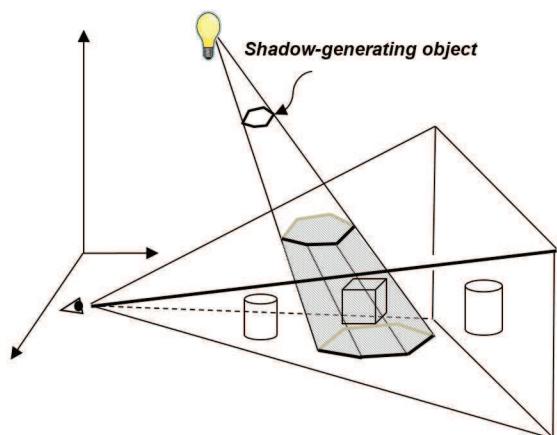


Figure 8.4
Shadow volume.

computationally expensive even on modern GPU hardware. Geometry shaders can be used to generate shadow volumes, and the *stencil buffer*¹ can be used to determine whether a pixel is within the volume. Some graphics cards include hardware support for optimizing certain shadow volume operations.

8.4 SHADOW MAPPING

One the most practical and popular methods for casting shadows is called *shadow mapping*. Although it is not always as accurate as shadow volumes (and is often accompanied by pesky artifacts), shadow mapping is easier to implement, can be used in a wide variety of situations, and enjoys powerful hardware support.

We would be remiss if we failed to clarify our use of the word “easier” in the previous paragraph. Although shadow mapping is simpler than shadow volumes (both conceptually and in practice), it is by no means “easy”! Students often find shadow mapping among the most difficult techniques to implement in a 3D graphics course. Shader programs are by nature difficult to debug, and shadow mapping requires the perfect coordination of several components and shader modules. *Be advised that successful implementation of shadow mapping will be greatly facilitated by liberal use of the debugging tools described earlier in Section 2.2.*

Shadow mapping is based on a very simple and clever idea: namely, anything that cannot be seen by the *light* is in shadow. That is, if object #1 blocks the light from reaching object #2, it is the same as the light not being able to “see” object #2.

The reason this idea is so powerful is that we already have a method for determining if something can be “seen”—the hidden surface removal algorithm (HSR) using the Z-buffer, as described in Section 2.1.7. So, a strategy for finding shadows is to temporarily move the camera to the location of the light, apply the Z-buffer HSR algorithm, and then use the resulting depth information to find shadows.

Rendering our scene will require two passes: one to render the scene from the point of view of the light (but not actually drawing it to the screen), and a second pass to render it from the point of view of the camera. The purpose of pass one is to generate a Z-buffer from the light’s point of view. After completing pass one, we need to retain the Z-buffer and use it to help us generate shadows in pass two. Pass two actually draws the scene.

¹ The *stencil buffer* is a third buffer—along with the color buffer and the z-buffer—accessible through OpenGL. The stencil buffer is not described in this textbook.

Our strategy is now becoming more refined:

- (“pass 1”) *Render the scene from the light’s position. The depth buffer then contains, for each pixel, the distance between the light and the nearest object to it.*
- *Copy the depth buffer to a separate “shadow buffer.”*
- (“pass 2”) *Render the scene normally. For each pixel, look up the corresponding position in the shadow buffer. If the distance to the point being rendered is greater than the value retrieved from the shadow buffer, then the object being drawn at this pixel is further from the light than the object nearest the light, and therefore this pixel is in shadow.*

When a pixel is found to be in shadow, we need to make it darker. One simple and effective way of doing this is to render only its ambient lighting, ignoring its diffuse and specular components.

The method described above is often called “shadow buffering.” The term “shadow mapping” arises when, in the second step, we instead copy the depth buffer into a *texture*. When a texture object is used in this way, we will refer to it as a *shadow texture*, and OpenGL has support for shadow textures in the form of a `sampler2DShadow` type (discussed below). This allows us to leverage the power of hardware support for texture units and sampler variables (i.e., “texture mapping”), in the fragment shader, to quickly perform the depth lookup in pass 2. Our revised strategy now is:

- (pass 1) as before.
- *Copy the depth buffer into a texture.*
- (pass 2) as before, except that the shadow buffer is now a shadow texture.

Let’s now implement these steps.

8.4.1 Shadow Mapping (PASS ONE) – “Draw” Objects from Light Position

In step one, we first move our camera to the light’s position, and then render the scene. Our goal here is *not* to actually draw the scene on the display, but to complete just enough of the rendering process that the depth buffer is properly filled. Thus it will not be necessary to generate colors for the pixels, and so our first pass will utilize a vertex shader, but will not require a fragment shader.

```
#version 430
layout (location=0) in vec3 vertPos;
uniform mat4 shadowMVP;
void main(void)
{ gl_Position = shadowMVP * vec4(vertPos,1.0);
}
```

Figure 8.5

Shadow mapping pass 1 vertex shader.

Of course, moving the camera involves constructing an appropriate *view* matrix. Depending on the contents of the scene, we will need to decide on an appropriate *direction* to view the scene from the light. Typically, we would want this direction to be towards the region that would ultimately be rendered in step 3. This is often application specific—in our scenes, we

will generally be pointing the camera from the light to the origin.

Here are some of the details handled in pass one:

- Configure the buffer and shadow texture.
- Disable color output.
- Build a look-at matrix from the light towards the objects in view.
- Enable the GLSL pass one shader program, containing only the simple vertex shader shown in Figure 8.5 that expects to receive an MVP matrix. In this case, the MVP matrix will include the object’s model matrix M, the look-at matrix computed in the previous step (serving as the view matrix V), and the perspective matrix P. We call this MVP matrix “shadowMVP” because it is based on the point of view of the light rather than the camera. Since the view from the light isn’t actually being displayed, the pass one shader program doesn’t include a fragment shader.
- For each object, create the shadowMVP matrix, and call `glDrawArrays()`. It is not necessary to include textures or lighting in pass one, because objects are not rendered to the screen.

8.4.2 Shadow Mapping (Intermediate Step) – Copying the Z-Buffer to a Texture

OpenGL offers two methods for putting Z-buffer depth data into a texture unit. The first method is to generate an empty shadow texture, and then use the command `glCopyTexImage2D()` to copy the active depth buffer into the shadow texture.

The second method is to build a “custom framebuffer” back in pass one (rather than use the default Z-buffer), and attach the shadow texture to it using the command `glFrameBufferTexture()`. This command was introduced into OpenGL in

version 3.0, to further support shadow mapping. When using this approach, it isn't necessary to "copy" the Z-buffer into a texture, because the buffer already has a texture attached to it, and so the depth information is put into the texture by OpenGL automatically. This is the method we will use in our implementation.

8.4.3 Shadow Mapping (PASS TWO) – Rendering the Scene with Shadows

Much of pass two will resemble what we saw in Chapter 7. Namely, it is here that we render our complete scene and all of the items in it, along with the lighting, materials, and any textures adorning the objects in the scene. We also need to add the necessary code to determine, for each pixel, whether or not it is in shadow.

An important feature of pass two is that it utilizes two MVP matrices. One is the standard MVP matrix that transforms object coordinates into screen coordinates (as seen in most of our previous examples). The other is the `shadowMVP` matrix that was generated in pass one for use in rendering from the light's point of view—this will now be used in pass two for looking up depth information from the shadow texture.

A complication arises in pass two when we try to look up pixels in a texture map. The OpenGL camera utilizes a $[-1,+1]$ coordinate space, whereas texture maps utilize a $[0,1]$ space. A common solution is to build an additional matrix transform, typically called B , that converts (or "biases," hence the name) from camera space to texture space. Deriving B is fairly simple—a scale by $\frac{1}{2}$ followed by a translate by $\frac{1}{2}$.

The B matrix is as follows:

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

B is then concatenated onto the `shadowMVP` matrix for use in pass two, thusly:

$$\text{shadowMVP2} = [B] [\text{shadowMVP}(\text{pass1})]$$

Assuming that we use the method whereby a shadow texture has been attached to our custom framebuffer, OpenGL provides some relatively simple tools

for determining whether each pixel is in shadow as we draw the objects. Here is a summary of the details handled in pass two:

- Build the “B” transform matrix for converting from light to texture space (actually this is more appropriately done in `init()`).
- Enable the shadow texture for look-up.
- Enable color output.
- Enable the GLSL pass two rendering program, containing both vertex and fragment shaders.
- Build MVP matrix for the object being drawn based on the camera position (as normal).
- Build the `shadowMVP2` matrix (incorporating the B matrix, as described earlier)—the shaders will need it to look up pixel coordinates in the shadow texture.
- Send the matrix transforms to shader uniform variables.
- Enable buffers containing vertices, normal vectors, and texture coordinates (if used), as usual.
- Call `glDrawArrays()`.

In addition to their rendering duties, the vertex and fragment shaders have additional tasks:

- The vertex shader converts vertex positions from camera space to light space, and sends the resulting coordinates to the fragment shader in a vertex attribute so that they will be interpolated. This makes it possible to retrieve the correct values from the shadow texture.
- The fragment shader calls the `textureProj()` function, which returns a 0 or 1 indicating whether or not the pixel is in shadow (this mechanism is explained later). If it is in shadow, the shader darkens the pixel by not including its diffuse and specular contributions.

Shadow mapping is such a common task that GLSL provides a special type of sampler variable called a `sampler2DShadow` (as previously mentioned) that can be attached to a shadow texture in the Java/JOGL application. The `textureProj()` function is used to look up values from a shadow texture, and is similar to `texture()` that we saw previously in Chapter 5, except that it uses a `vec3` to index the texture rather than the usual `vec2`. Since a pixel coordinate is a `vec4`, it is necessary to project that onto 2D texture space in order to look up the depth value in the shadow texture map. As we will see below, `textureProj()` does all of this for us.

The remainder of the vertex and fragment shader code implements Blinn-Phong shading. They are shown in Figure 8.6 and Figure 8.7, with the added code for shadow mapping highlighted.

Let's examine more closely how we use OpenGL to perform the depth comparison between the pixel being rendered and the value in the shadow texture. We start in the vertex shader with a vertex coordinate in eye space, which we multiply by `shadowMVP2` to produce the corresponding coordinate in “light space” (that is, from the light’s point of view). The interpolated (3D) light space coordinates (x,y,z) are used in the fragment shader as follows. The z component represents the distance from the light to the pixel. The (x,y) components are used to retrieve the depth information stored in the (2D) shadow texture. This retrieved value (the distance to the object nearest the light) is compared with z . This comparison produces

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;

out vec3 varyingNormal, varyingLightDir, varyingVertPos, varyingHalfVec;
out vec4 shadow_coord;

struct PositionalLight { vec4 ambient, diffuse, specular; vec3 position; };
struct Material { vec4 ambient, diffuse, specular; float shininess; };
uniform vec4 globalAmbient;
uniform PositionalLight light; // light's position is assumed to be in eye space
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
uniform mat4 shadowMVP2;
layout (binding=0) uniform sampler2DShadow shTex;

void main(void)
{   varyingVertPos = (mv_matrix * vec4(vertPos,1.0)).xyz;
    varyingLightDir = light.position - varyingVertPos;
    varyingNormal = (norm_matrix * vec4(vertNormal,1.0)).xyz;
    varyingHalfVec = (varyingLightDir - varyingVertPos).xyz;
    shadow_coord = shadowMVP2 * vec4(vertPos,1.0);
    gl_Position = proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

Figure 8.6

Shadow mapping pass 2 vertex shader.

```

#version 430
in vec3 varyingNormal, varyingLightDir, varyingVertPos, varyingHalfVec;
in vec4 shadow_coord;
out vec4 fragColor;
// same structs and uniforms as in the vertex shader

void main(void)
{
    vec3 L = normalize(varyingLightDir);
    vec3 N = normalize(varyingNormal);
    vec3 V = normalize(-varyingVertPos);
    vec3 H = normalize(varyingHalfVec);

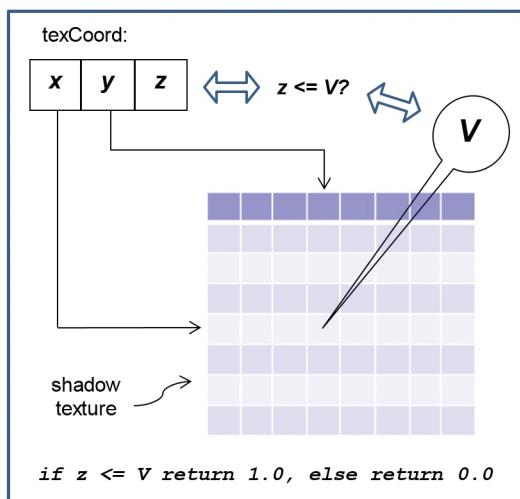
    float inShadow = textureProj(shTex, shadow_coord);

    fragColor = globalAmbient * material.ambient + light.ambient * material.ambient;
    If (inShadow != 0.0)
    {
        fragColor += light.diffuse * material.diffuse * max(dot(L,N),0.0)
                    + light.specular * material.specular
                    * pow(max(dot(H,N),0.0),material.shininess*3.0);
    }
}

```

Figure 8.7

Shadow mapping pass 2 fragment shader.

**Figure 8.8**

Automatic depth comparison.

a “binary” result that tells us whether the pixel we are rendering is further from the light than the object nearest the light (i.e., whether the pixel is in shadow).

If in OpenGL we use `glFrameBufferTexture()` as described earlier, and we enable *depth testing*, then using a `sampler2DShadow` and `textureProj()` as shown in the fragment shader (Figure 8.7) will do exactly what we need. That is, `textureProj()` will output either a 0 or a 1 depending on the depth comparison. Based on this value, we can then in the fragment shader omit the diffuse and specular contributions when the pixel is further from the light than the object nearest the light, effectively creating the shadow when appropriate. An overview is shown in Figure 8.8.

We are now ready to build our JOGL application to work with the previously described shaders.

8.5 A SHADOW MAPPING EXAMPLE

Consider the scene in Figure 8.9 that includes a torus and a pyramid. A positional light has been placed on the left (note the specular highlights—the pyramid *should* cast a shadow on the torus).

To clarify the development of the example, our first step will be to *render pass one to the screen* to make sure it is working properly. To do this, we will temporarily add a simple fragment shader (it will not be included in the final version) to pass one that just outputs a constant color (e.g., red); for example:

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

Let's assume that the origin of the scene above is situated at the center of the figure, in between the pyramid and the torus. In pass one we place the camera at the light's position (at the left in Figure 8.10) and point it towards (0,0,0). If we then draw the objects in red, it produces the output shown at the

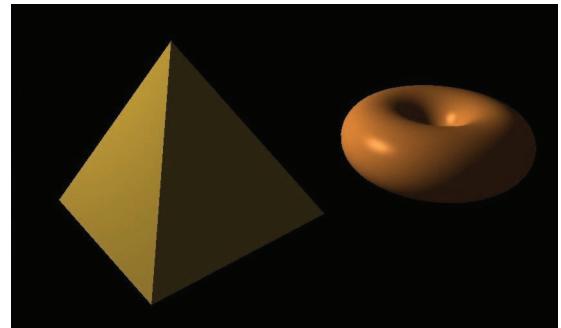


Figure 8.9
Lighted scene without shadows.

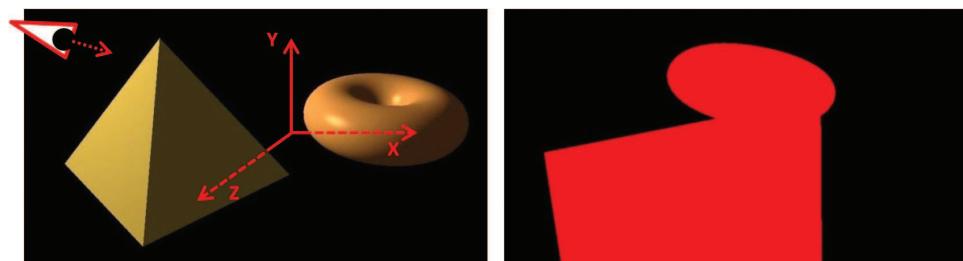


Figure 8.10

Pass one: Scene (left) from light's point of view (right).

right in Figure 8.10. Note the torus near the top—from this vantage point it is partially behind the pyramid.

The complete two-pass JOGL code with lighting and shadow mapping is shown in Program 8.1:

Program 8.1 Shadow Mapping

```
// Much is the same as we have seen before. New sections to support shadows are highlighted.
// The imports necessary for lighting, etc., would be included at the start, are the same as before,
// and are not shown here.

public class Code extends JFrame implements GLEventListener
{ // variables for lights, materials, objects, rendering programs, uniforms, buffers, shader sources,
  // etc., would go here.

  ...
  private Torus myTorus = new Torus(0.6f, 0.4f, 48);
  ...

  // set location of torus, pyramid, camera, and light
  private Point3D torusLoc = new Point3D(1.6, 0.0, -0.3);
  private Point3D pyrLoc = new Point3D(-1.0, 0.1, 0.3);
  private Point3D cameraLoc = new Point3D(0.0, 0.2, 6.0);
  private Point3D lightLoc = new Point3D(-3.8f, 2.2f, 1.1f);

  // light and camera view matrix transforms are all declared here (m_matrix, v_matrix, etc.) and
  // are of type Matrix3D.

  ...
  // shadow-related variables
  private int screenSizeX, screenSizeY;
  private int [] shadow_tex = new int [1];
  private int [] shadow_buffer = new int [1];
  private Matrix3D lightV_matrix = new Matrix3D();
  private Matrix3D lightP_matrix = new Matrix3D();
  private Matrix3D shadowMVP = new Matrix3D();
  private Matrix3D shadowMVP2 = new Matrix3D();
  private Matrix3D b = new Matrix3D();

  public Code()
  { // The constructor is unchanged from Program 7-3, and so it is not shown here.
    // This example assumes that FPSAnimator() is being used.
  }
```

```

// The init() routine performs the usual calls to compile shaders and initialize objects. It also calls
// setupShadowBuffers() to instantiate the buffers related to shadow-mapping.
// Lastly, it builds the B matrix for converting from light space to texture space.

public void init(GLAutoDrawable drawable)
{ GL4 gl = (GL4) GLContext.getCurrentGL();

    createShaderProgram();
    setupVertices();
    setupShadowBuffers();

    b.setElementAt(0,0,0.5); b.setElementAt(0,1,0.0);
    b.setElementAt(0,2,0.0); b.setElementAt(0,3,0.5f);
    b.setElementAt(1,0,0.0); b.setElementAt(1,1,0.5);
    b.setElementAt(1,2,0.0); b.setElementAt(1,3,0.5f);
    b.setElementAt(2,0,0.0); b.setElementAt(2,1,0.0);
    b.setElementAt(2,2,0.5); b.setElementAt(2,3,0.5f);
    b.setElementAt(3,0,0.0); b.setElementAt(3,1,0.0);
    b.setElementAt(3,2,0.0); b.setElementAt(3,3,1.0f);
}

public void setupShadowBuffers()
{ GL4 gl = (GL4) GLContext.getCurrentGL();
    screenSizeX = myCanvas.getWidth();
    screenSizeY = myCanvas.getHeight();

    // create the custom frame buffer
    gl glGenFramebuffers(1, shadow_buffer, 0);

    // create the shadow texture and configure it to hold depth information.
    // these steps are similar to those in Program 5.2
    gl glGenTextures(1, shadow_tex, 0);
    gl glBindTexture(GL_TEXTURE_2D, shadow_tex[0]);
    gl glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
                    screenSizeX, screenSizeY, 0, GL_DEPTH_COMPONENT, GL_FLOAT, null);
    gl glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    gl glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    gl glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                      GL_COMPARE_REF_TO_TEXTURE);
    gl glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
}

// display() manages the setup of the custom frame buffer, the shadow texture
// in preparation for pass 1 and pass 2 respectively. New shadow-related features are highlighted.

```

```

public void display(GLAutoDrawable drawable)
{   GL4 gl = (GL4) GLContext.getCurrentGL();

    // set up light and perspective matrix as before
    currentLight.setPosition(lightLoc);
    aspect = (float) myCanvas.getWidth() / (float) myCanvas.getHeight();
    proj_matrix = perspective(50.0f, aspect, 0.1f, 1000.0f);

    // clear the color buffer to the desired background color
    FloatBuffer bg = FloatBuffer.allocate(4);
    bg.put(0, 0.0f); bg.put(1, 0.0f); bg.put(2, 0.0f); bg.put(3, 1.0f);
    gl.glClearBufferfv(GL_COLOR, 0, bg);

    // make the custom frame buffer current, and associate it with the shadow texture
    gl glBindFramebuffer(GL_FRAMEBUFFER, shadow_buffer[0]);
    gl glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, shadow_tex[0], 0);

    // disable drawing colors, but enable the depth computation
    gl glDrawBuffer(GL_NONE);
    gl glEnable(GL_DEPTH_TEST);

    passOne();

    // restore the default display buffer, and re-enable drawing
    gl glBindFramebuffer(GL_FRAMEBUFFER, 0);
    gl glActiveTexture(GL_TEXTURE0);
    gl glBindTexture(GL_TEXTURE_2D, shadow_tex[0]);
    gl glDrawBuffer(GL_FRONT);    // drawing only front faces allows back face culling
    passTwo();
}

// What follows now are the modules for the first and second passes.
// They are largely identical to things we have seen before.
// Shadow-related additions are highlighted.

public void passOne()
{   GL4 gl = (GL4) GLContext.getCurrentGL();

    // rendering_program1 contains only the pass one vertex shader
    gl glUseProgram(rendering_program1);

    // build the light's P and V matrices to look-at the origin
    Point3D origin = new Point3D(0.0, 0.0, 0.0);
    Vector3D up = new Vector3D(0.0, 1.0, 0.0);
    lightV_matrix.setTolIdentity();
    lightP_matrix.setTolIdentity();
    lightV_matrix = lookAt(currentLight.getPosition(), origin, up);    // vector from light to origin
    lightP_matrix = perspective(50.0f, aspect, 0.1f, 1000.0f);
}

```

```

// the following blocks of code render the torus into the depth buffer
m_matrix.setTIdentity();
m_matrix.translate(torusLoc.getX(),torusLoc.getY(),torusLoc.getZ());
m_matrix.rotateX(25.0);

// we are drawing from the light's point of view, so we use the light's P and V matrices
shadowMVP.setTIdentity();
shadowMVP.concatenate(lightP_matrix);
shadowMVP.concatenate(lightV_matrix);
shadowMVP.concatenate(m_matrix);
int shadow_location = gl.glGetUniformLocation(rendering_program1, "shadowMVP");
gl.glUniformMatrix4fv(shadow_location, 1, false, shadowMVP.getFloatValues(), 0);

// we only need to set up torus vertices buffer – we don't need its textures or normals for pass one.
gl glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[0]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

gl glClear(GL_DEPTH_BUFFER_BIT);
gl glEnable(GL_CULL_FACE);
gl glFrontFace(GL_CCW);
gl glEnable(GL_DEPTH_TEST);
gl glDepthFunc(GL_LEQUAL);

gl glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);

// repeat for the pyramid (but don't clear the GL_DEPTH_BUFFER_BIT)
...
gl glDrawArrays(GL_TRIANGLES, 0, pyramid.getNumVertices());
}

public void passTwo()
{
    GL4 gl = (GL4) GLContext.getCurrentGL();

    // rendering_program2 includes the pass two vertex and fragment shaders
    gl glUseProgram(rendering_program2);

    // draw the torus – this time we need to include lighting, materials, normals, etc.
    // We also need to provide MVP tranforms for BOTH camera space and light space.
    thisMaterial = Material.BRONZE;
    installLights(rendering_program2, v_matrix);
    mv_location = gl.glGetUniformLocation(rendering_program2, "mv_matrix");
    proj_location = gl.glGetUniformLocation(rendering_program2, "proj_matrix");
    n_location = gl.glGetUniformLocation(rendering_program2, "norm_matrix");
    int shadow_location = gl.glGetUniformLocation(rendering_program2, "shadowMVP2");
}

```

```

// build the MV matrix from the camera's point of view
m_matrix.setTIdentity(); v_matrix.setTIdentity(); mv_matrix.setTIdentity();
m_matrix.translate(torusLoc.getX(),torusLoc.getY(),torusLoc.getZ());
v_matrix.translate(-cameraLoc.getX(),-cameraLoc.getY(),-cameraLoc.getZ());
mv_matrix.concatenate(v_matrix);
mv_matrix.concatenate(m_matrix);

// build the MVP matrix from the light's point of view
shadowMVP2.setTIdentity();
shadowMVP2.concatenate(b);
shadowMVP2.concatenate(lightP_matrix);
shadowMVP2.concatenate(lightV_matrix);
shadowMVP2.concatenate(m_matrix);

// put the MV and PROJ matrices into the corresponding uniforms
gl.glUniformMatrix4fv(mv_location, 1, false, mv_matrix.getFloatValues(), 0);
gl.glUniformMatrix4fv(proj_location, 1, false, proj_matrix.getFloatValues(), 0);
gl.glUniformMatrix4fv(n_location, 1, false, (mv_matrix.inverse()).transpose().getFloatValues(), 0);
gl.glUniformMatrix4fv(shadow_location, 1, false, shadowMVP2.getFloatValues(), 0);

// set up torus vertices and normal buffers (and texture coordinate buffer if used)
gl glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[0]); // torus vertices
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

gl glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[2]); // torus normals
gl glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(1);

gl glClear(GL_DEPTH_BUFFER_BIT);
gl glEnable(GL_CULL_FACE);
gl glFrontFace(GL_CCW);
gl glEnable(GL_DEPTH_TEST);
gl glDepthFunc(GL_LEQUAL);

gl glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);

// repeat for the pyramid (but don't clear the GL_DEPTH_BUFFER_BIT)
...
gl glDrawArrays(GL_TRIANGLES, 0, pyramid.getNumVertices());
}

```

Program 8.1 shows the relevant portions of the Java/JOGL application that interact with the pass one and pass two shaders previously detailed. Not shown are the usual modules for reading in and compiling the shaders, building the models

and their related buffers, installing the positional light's ADS characteristics, and performing the perspective and look-at matrix computations. Those are unchanged from previous examples.

8.6 SHADOW MAPPING ARTIFACTS

Although we have implemented all of the basic requirements for adding shadows to our scene, running Program 8.1 produces mixed results, as shown in Figure 8.11.

The good news is that our pyramid is now casting a shadow on the torus! Unfortunately, this success is accompanied by a severe artifact. There are wavy lines covering many of the surfaces in the scene. This is a common by-product of shadow mapping, and is called "*shadow acne*," or *erroneous self-shadowing*.

Shadow acne is caused by rounding errors during depth testing. The texture coordinates computed when looking up the depth information in a shadow texture often don't exactly match the actual coordinates. Thus, the lookup may return the depth for a neighboring pixel, rather than the one being rendered. If the distance to the neighboring pixel is further, then our pixel will appear to be in shadow even if it isn't.

Shadow acne can also be caused by differences in precision between the texture map and the depth computation. This too can lead to rounding errors and subsequent incorrect assessment of whether or not a pixel is in shadow.

Fortunately, fixing shadow acne is fairly easy. Since shadow acne typically occurs on surfaces that are *not* in shadow, a simple trick is to move every pixel slightly closer to the light during pass one, and then move them back to their normal positions for pass two. This is usually sufficient to compensate for either

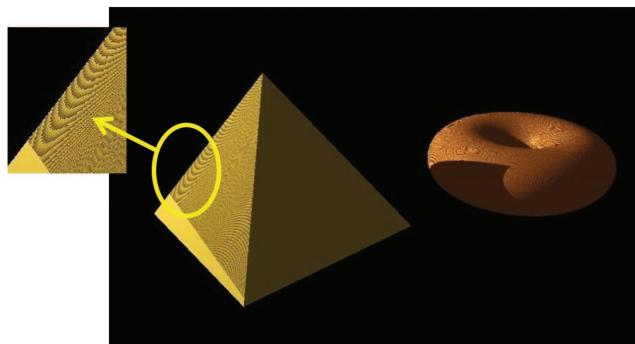


Figure 8.11
Shadow "acne."

type of rounding error. An easy way is to call `glPolygonOffset()` in the `display()` function, as shown in Figure 8.12 (highlighted):

```

public void display(GLAutoDrawable drawable)
{   GL4 gl = (GL4) GLContext.getCurrentGL();

    currentLight.setPosition(lightLoc);
    aspect = (float) myCanvas.getWidth() / (float) myCanvas.getHeight();
    proj_matrix = my_perspective(50.0f, aspect, 0.1f, 1000.0f);

    ...
    // clearing the depth and color buffers as before goes here
    ...

    gl glBindFramebuffer(GL_FRAMEBUFFER, shadow_buffer[0]);
    gl framebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, shadow_tex[0], 0);
    int buffs[] = { GL_COLOR_ATTACHMENT0 };
    gl drawBuffers(1, buffs, 0);
    gl drawBuffer(GL_NONE);
    gl glEnable(GL_DEPTH_TEST);
    gl glEnable(GL_POLYGON_OFFSET_FILL);
    gl glPolygonOffset(2.0f, 4.0f);

    firstPass();

    gl glDisable(GL_POLYGON_OFFSET_FILL);

    gl glBindFramebuffer(GL_FRAMEBUFFER, 0);
    gl setActiveTexture(GL_TEXTURE0);
    gl glBindTexture(GL_TEXTURE_2D, shadow_tex[0]);
    gl drawBuffer(GL_FRONT);

    secondPass();
}

```

Figure 8.12

Combating shadow acne.

Adding these few lines of code to our `display()` function improves the output of our program considerably, as shown in Figure 8.13. Note also that the inner circle of the torus now displays a small correctly cast shadow on itself.

Although fixing shadow acne is easy, sometimes the repair causes new artifacts. The “trick” of moving the object before pass one can sometimes cause a gap to appear inside an object’s shadow. An example of this is shown in Figure 8.14. This artifact is often called “peter panning,” because sometimes it causes the shadow

of a resting object to inappropriately separate from the object's base (thus making portions of the object's shadow detach from the rest of the shadow, reminiscent of J. M. Barrie's character Peter Pan [PP16]). Fixing this artifact requires adjusting the `glPolygonOffset()` parameters. If they are too small, shadow acne can appear; if too large, peter panning happens.

There are many other artifacts that can happen during shadow mapping. For example, shadows can *repeat* as a result of the region of the scene being rendered in pass one (into the shadow buffer) being different than the region of the scene rendered in pass two (they are from different vantage points). Because of this difference, those portions of the scene rendered in pass two that fall outside the region rendered in pass one will attempt to access the shadow texture using texture coordinates outside of the range (0..1). Recall that the default behavior in this case is `GL_REPEAT`, which can result in incorrectly duplicated shadows.

One possible solution is to add the following lines of code to `setupShadowBuffers()`, to set the texture wrap mode to “clamp to edge”:

```
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
gl.glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

This causes values outside of a texture edge to be clamped to the value at edge (instead of repeating). Note that this approach can introduce its own artifacts, namely, when a shadow exists at the edge of the shadow texture, clamping to the edge can produce a “shadow bar” extending to the edge of the scene.

Another common error is *jagged shadow edges*. This can happen when the shadow being cast is significantly

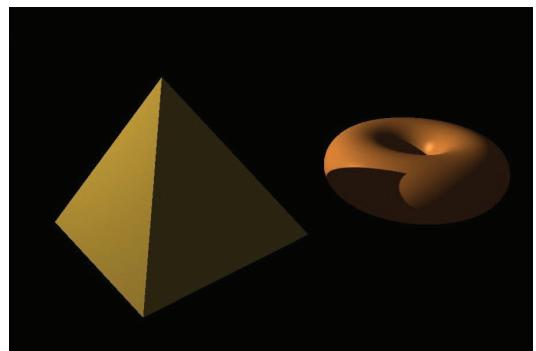


Figure 8.13

Rendered scene with shadows.

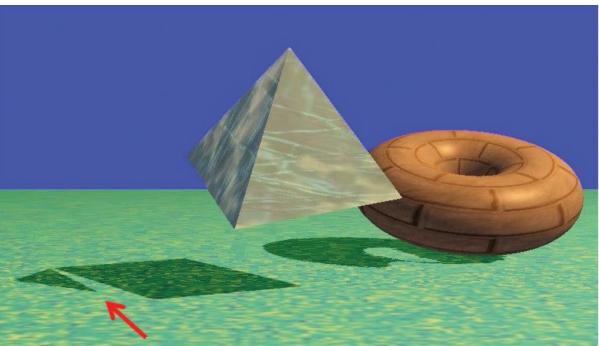


Figure 8.14

“Peter panning.”

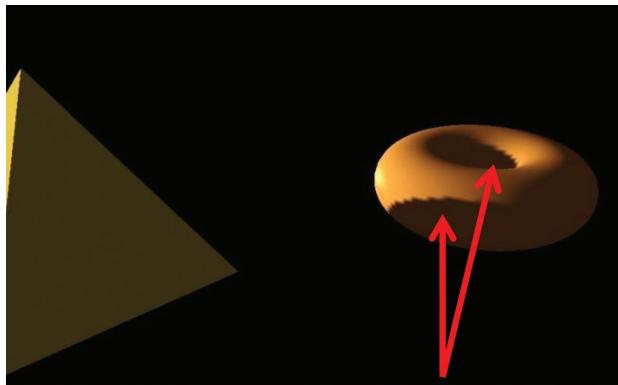


Figure 8.15
Jagged shadow edges.

of this textbook. See the supplemental chapter notes for references to advanced techniques for improving shadow mapping.

SUPPLEMENTAL NOTES

In this chapter we have only given the most basic of introductions to the world of shadows in 3D graphics. Even using the basic method of shadow mapping presented here will likely require further study if used in more complex scenes.

For example, when adding shadows to a scene in which some of the objects are textured, it is necessary to ensure that the fragment shader properly distinguishes between the shadow texture and other textures. A simple way of doing this is to bind them to different texture units, such as:

```
layout (binding = 0) uniform sampler2DShadow shTtex;
layout (binding = 1) uniform sampler2D otherTexture;
```

Then, the JOGL application can refer to the two samplers by their binding values.

When a scene utilizes multiple lights, multiple shadow textures are necessary—one for each light source. And, a *pass one* will need to be performed for each one, with the results blended in pass two.

larger than the shadow buffer can accurately represent. This usually depends on the location of the objects and light(s) in the scene. In particular, it commonly occurs when the light source is relatively distant from the objects involved. An example is shown in Figure 8.15.

Eliminating jagged shadow edges is not as simple as for the previous artifacts. There are a number of effective approaches that are available, but these are outside the scope



Figure 8.16
Soft shadow real-world example.

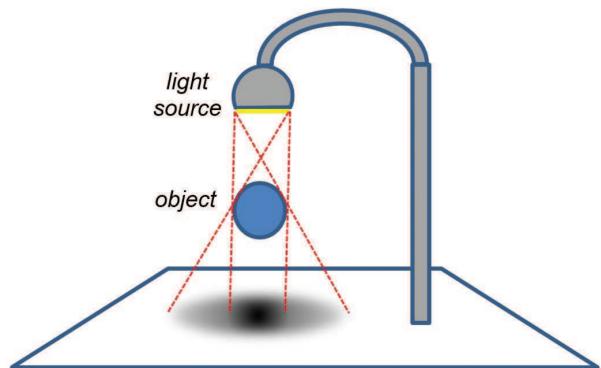


Figure 8.17
Soft shadow effect.

Although we have used perspective projection at each phase of shadow mapping, it is worth noting that *orthographic* projection is often preferred when the light source is distant and directional, rather than the positional light we utilized.

Finally, the methods presented here are limited to producing *hard shadows*. These are shadows with sharp edges. However, most shadows that occur in the real world are *soft shadows*. That is, their edges are blurred to various degrees, such as in the real-world example shown in Figure 8.16. There are many causes of soft shadows, the most common being that light sources are rarely points—more often they occupy some area. This gives rise to the penumbra effect shown in Figure 8.17.

There are many ways of simulating the penumbra effect in soft shadows. For example, the shadow buffer can be sampled multiple times with a bit of randomness incorporated. Such methods incur some amount of overhead that can impact performance.

Generating realistic shadows is a rich and complex area of computer graphics, and many of the available techniques are outside the scope of this text. Readers interested in more detail are encouraged to investigate more specialized resources such as [GP10] and [MI16].

Exercises

- 8.1** In Program 8.1, experiment with different settings for `glPolygonOffset()`, and observe the effects on shadow artifacts such as peter panning.
- 8.2 (PROJECT)** Modify Program 8.1 so that the *light* can be positioned by moving the mouse, similar to Exercise 7.1. You will probably notice that some lighting positions exhibit shadow artifacts, while others look fine.
- 8.3 (PROJECT)** Add animation to Program 8.1, such that either the objects, or the light (or both) move around on their own—such as one revolving around the other. The shadow effects will be more pronounced if you add a ground plane to the scene, such as the one illustrated in Figure 8.14.

References

- [AS14] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with WebGL*, 7th ed. (Pearson, 2014).
- [BL88] J. Blinn, “Me and My (Fake) Shadow,” *IEEE Computer Graphics and Applications* 8, no. 2 (1988).
- [CR77] F. Crow, “Shadow Algorithms for Computer Graphics,” *Proceedings of SIGGRAPH '77* 11, no. 2 (1977).
- [GP10] *GPU Pro* (series), ed. Wolfgang Engel (A. K. Peters, Ltd., 2010–2016).
- [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).
- [MI16] *Common Techniques to Improve Shadow Depth Maps* (Microsoft Corp., 2016), accessed July 2016, [https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx).
- [PP16] Peter Pan, Wikipedia page, accessed July 2016, https://en.wikipedia.org/wiki/Peter_Pan.

SKY AND BACKGROUNDS

9.1	<i>Skyboxes</i>	211
9.2	<i>Skydomes</i>	214
9.3	<i>Implementing a Skybox</i>	215
9.4	<i>Environment Mapping</i>	223
	<i>Supplemental Notes</i>	228



The realism in an outdoor 3D scene can often be improved by generating a realistic effect at the horizon. As we look beyond our nearby buildings and trees, we are accustomed to seeing large distant objects such as clouds, mountains, or the sun (or at night, the moon and stars). However, adding such objects to our scene as individual models may come at an unacceptable performance cost. A *skybox* or *skydome* provides a relatively simple way of efficiently generating a convincing horizon.

9.1 SKYBOXES

The concept of a *skybox* is a remarkably clever and simple one:

1. Instantiate a cube object.
2. Texture the cube with the desired environment.
3. Position the cube so it surrounds the camera.

We already know how to do all of these steps. There are a few additional details, however.

- *How do we make the texture for our horizon?*

A cube has six faces, and we will need to texture all of them. One way is to use six image files, and six texture units. Another common (and efficient) way is to use a single image that contains textures for all six faces, such as shown in Figure 9.1.

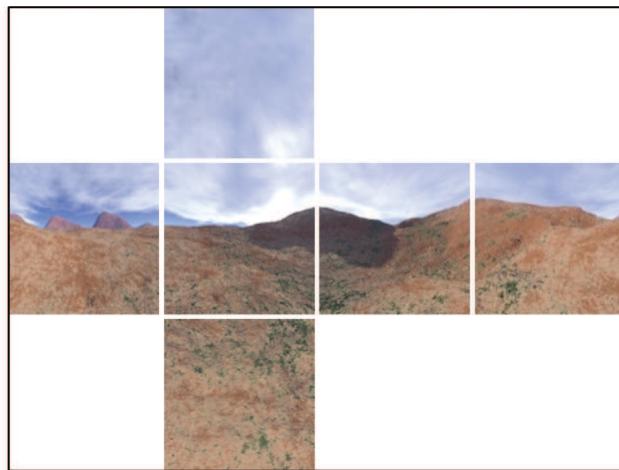


Figure 9.1
Six-faced skybox texture cube map.

An image that can texture all six faces of a cube with a single texture unit is an example of a *texture cube map*. The six portions of the cube map correspond to the top, bottom, front, back, and two sides of the cube. When “wrapped” around the cube, it acts as a horizon for a camera placed inside the cube, as shown below in Figure 9.2.

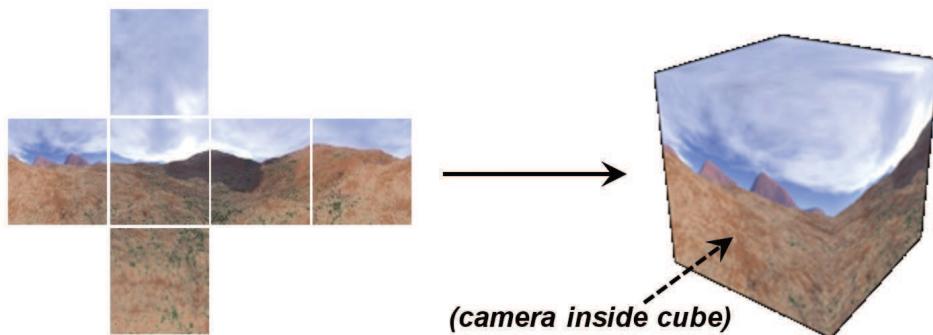


Figure 9.2
Cube map with camera placed inside.

Texturing the cube with a texture cube map requires specifying appropriate texture coordinates. Figure 9.3 shows the distribution of texture coordinates that are in turn assigned to each of the cube vertices.

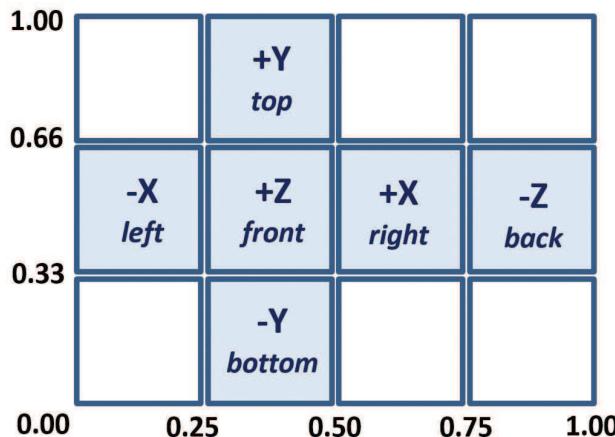


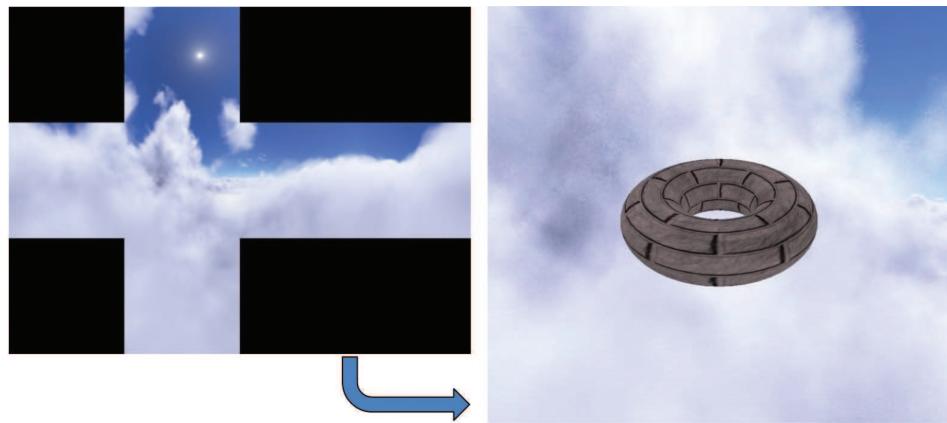
Figure 9.3
Cube map texture coordinates.

- *How do we make the skybox appear “distant”?*

Another important factor in building a skybox is ensuring that the texture appears as a distant horizon. At first, one might assume this would require making the skybox very large. However, it turns out that this isn’t desirable because it would stretch and distort the texture. Instead, it is possible to make the skybox *appear* very large (and thus distant), by using the following two-part trick:

- Disable depth testing, and render the skybox first (re-enabling depth testing when rendering the other objects in the scene).
- Move the skybox with the camera (if the camera moves).

By drawing the skybox first with depth testing disabled, the depth buffer will still be filled completely with 1.0’s (i.e., maximally far away). Thus all other objects in the scene will be fully rendered; that is, none of the other objects will be blocked by the skybox. This causes the walls of the skybox to appear further away than every other object, *regardless of the actual size of the skybox*. The actual skybox cube itself can be quite small, as long as it is moved along with the camera whenever the camera moves. Figure 9.4 shows viewing a simple scene (actually just a brick-textured torus) from inside a skybox.

**Figure 9.4**

Viewing a scene from inside a skybox.

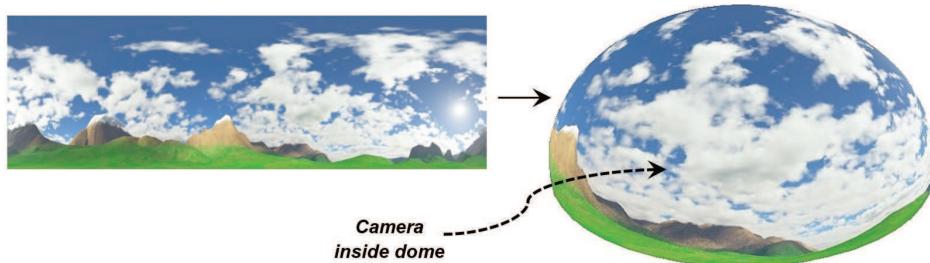
- *How do we construct the texture cube map?*

Building a texture cube map image, from artwork or photos, requires care to avoid “seams” at the cube face junctions, and to create proper perspective so that the skybox will appear realistic and undistorted. Many tools exist for assisting in this regard: Terragen, Autodesk 3ds Max, Blender, and Adobe Photoshop have tools for building or working with cube maps. There are also many websites offering a variety of off-the-shelf cube maps, some for a price, some for free.

9.2 SKYDOMES

Another way of building a horizon effect is to use a *skydome*. The basic idea is the same as for a skybox, except that instead of using a textured cube, we use a textured *sphere* (or half a sphere). As was done for the skybox, we render the skydome first (with depth testing disabled), and keep the camera positioned at the center of the skydome. (The skydome texture in Figure 9.5 was made using Terragen [TE16].)

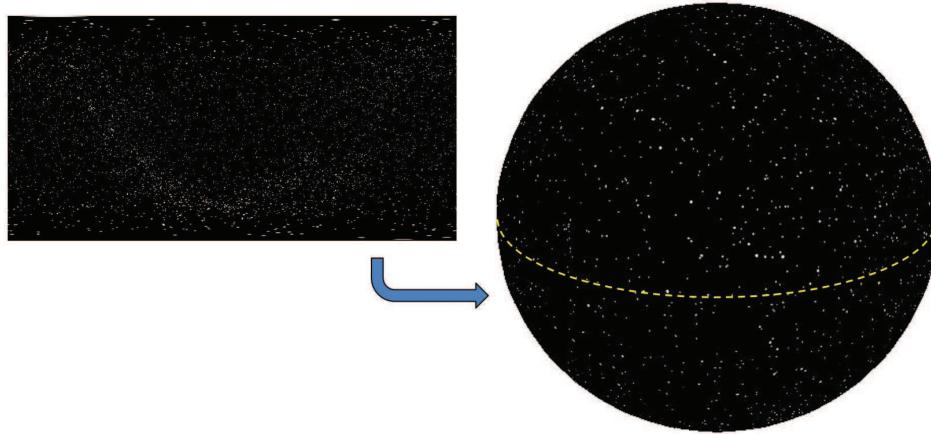
Skydomes have some advantages over skyboxes. For example, they are less susceptible to distortion and seams (although spherical distortion at the poles must be accounted for in the texture image). One disadvantage of a skydome is that a sphere or dome is a more complex model than a cube, with many more

**Figure 9.5**

Skydome with camera placed inside.

vertices and a potentially varying number of vertices depending on the desired precision.

When using a skydome to represent an outdoor scene, it is usually combined with a ground plane or some sort of terrain. When using a skydome to represent a scene in space, such as a starfield, it is often more practical to use a sphere, such as shown below in Figure 9.6 (a dashed line has been added for clarity in visualizing the sphere).

**Figure 9.6**

Skydome of stars using a sphere (starfield from [BO01]).

9.3 IMPLEMENTING A SKYBOX

Despite the advantages of a skydome, skyboxes are still more common. They also are better supported in OpenGL, which is advantageous when doing *environment*

mapping (covered later in this chapter). For these reasons, we will focus on skybox implementation.

There are two methods of implementing a skybox: building a simple one from scratch, and using the cube map facilities in OpenGL. Each has its advantages, so we will cover them both.

9.3.1 Building a Skybox from Scratch

We have already covered almost everything needed to build a simple skybox. A cube model was presented in Chapter 4; we can assign the texture coordinates as shown earlier in this chapter in Figure 9.3. We saw how to read in textures, and how to position objects in 3D space. We will see how to easily enable and disable depth testing (it's a single line of code).

Program 9.1 shows the code organization for our simple skybox, with a scene consisting of just a single textured torus. The texture coordinate assignments and calls that enable/disable the depth buffer are highlighted.

Program 9.1 Simple Skybox

```
// imports as before
...
public class Code extends JFrame implements GLEventListener,
{
    // all variable declarations, constructor, and init() same as before
    ...
    public void display(GLAutoDrawable drawable)
    {   // clear color and depth buffers, and create projection and camera view matrix as before
        ...
        gl.glUseProgram(rendering_program);
        // Prepare to draw the skybox first. The M matrix places the skybox at the camera location
        m_matrix.setTIdentity();
        m_matrix.translate(camera.getLocation().getX(), camera.getLocation().getY(),
                           camera.getLocation().getZ());
        ...
        // build the MODEL-VIEW matrix
        mv_matrix.setTIdentity();
        mv_matrix.concatenate(v_matrix);
        mv_matrix.concatenate(m_matrix);
        ...
        // put MV and PROJ matrices into uniforms, as before
        ...
    }
}
```

```

// set up buffer containing vertices
gl.glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[2]);
gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

// set up buffer containing texture coordinates
gl.glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[3]);
gl.glVertexAttribPointer(1, 2, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(1);

// activate the skybox texture
gl.glActiveTexture(GL_TEXTURE0);
gl glBindTexture(GL_TEXTURE_2D, textureID2);

gl glEnable(GL_CULL_FACE);
gl.glFrontFace(GL_CCW);           // cube has CW winding order, but we are viewing its interior

gl.glDisable(GL_DEPTH_TEST);
gl.glDrawArrays(GL_TRIANGLES, 0, 36);      // draw the skybox without depth testing
gl glEnable(GL_DEPTH_TEST);

// now draw desired scene objects as before
...
gl.glDrawArrays(...); // as before for scene objects
}

private void setupVertices()
{
    // cube_vertices defined same as before
    // cube texture coordinates for the skybox:
    float[] cube_texture_coord =
    {
        .25f, .66f, .25f, .33f, .50f, .33f,           // front triangles
        .50f, .33f, .50f, .66f, .25f, .66f,           //
        .50f, .33f, .75f, .33f, .50f, .66f,           // right triangles
        .75f, .33f, .75f, .66f, .50f, .66f,           //
        .75f, .33f, 1.0f, .33f, .75f, .66f,           // back triangles
        1.0f, .33f, 1.0f, .66f, .75f, .66f,           //
        0.0f, .33f, .25f, .33f, 0.0f, .66f,           // left triangles
        .25f, .33f, .25f, .66f, 0.0f, .66f,           //
        .25f, 0.0f, .50f, 0.0f, .50f, .33f,           // bottom triangles
        .50f, .33f, .25f, .33f, .25f, 0.0f,           //
        .25f, .66f, .50f, .66f, .50f, 1.0f,           // top triangles
        .50f, 1.0f, .25f, 1.0f, .25f, .66f           //
    };
    // set up buffers for cube and scene objects as usual
}
// modules for loading shaders, textures, etc. as before
}

```

Standard texturing shaders are now used for all objects in the scene, including the cube map:

Vertex Shader

```
#version 430
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 tex_coord;
out vec2 tc;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform sampler2D s;

void main(void)
{
    tc = tex_coord;
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

Fragment Shader

```
#version 430
in vec2 tc;
out vec4 fragColor;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform sampler2D s;

void main(void)
{
    fragColor = texture(s,tc);
}
```

The output of Program 9.1 is shown in Figure 9.7, for each of two different cube map textures.

As mentioned earlier, skyboxes are susceptible to image distortion and *seams*. Seams are lines that are sometimes visible where two texture images meet, such as along the edges of the cube. Figure 9.8 shows an example of a seam in the upper part of the image that is an artifact of running the previous Program 9.1. Avoiding seams requires careful construction of the cube map image, and assignment of precise texture coordinates. There exist tools for reducing seams along image edges (such as [Gl16]); this topic is outside the scope of this book.

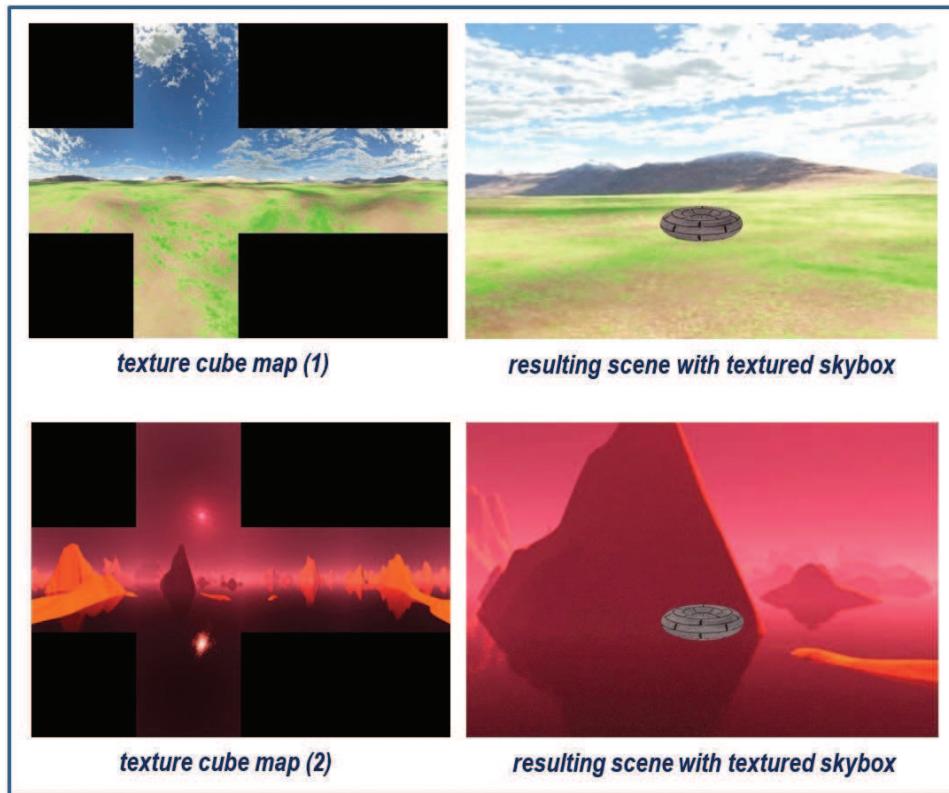


Figure 9.7
Simple skybox results.

9.3.2 Using OpenGL Cube Maps

Another way to build a skybox is to use an OpenGL *texture cube map*. OpenGL cube maps are a bit more complex than the simple approach we saw in the previous section. There are advantages, however, to using OpenGL cube maps, such as seam reduction and support for environment mapping.



Figure 9.8
Skybox “seam” artifact.

Texture cube maps are similar to *3D textures* that we will study later, in that they are accessed using three texture coordinates—often labeled (s,t,r) —rather than two. Another unique characteristic of OpenGL texture cube maps is that the images in them are oriented with texture coordinate $(0,0,0)$ at the upper left (rather than the usual lower left) of the texture image; this is often a source of confusion.

Whereas the method shown in Program 9.1 reads in a single image for texturing the cube map, the `loadCubeMap()` function shown in Program 9.2 reads in six separate cube face image files. The cube map is generated first as a JOGL Texture of type `GL_TEXTURE_CUBE_MAP`. It then reads the image files by using the `newTextureData()` function in the JOGL `TextureIO` class, which results in six objects of type `TextureData`. Since the texture is specified as being a cube map, that means that it has six *sub-images*. Each sub-image is associated with one of the six `TextureData` objects containing the image file data, using the `updateImage()` function in the JOGL `Texture` class. The OpenGL texture cube map object is then extracted from the JOGL texture object using `getTextureObject()`. The completed OpenGL texture cube map is referenced by an int identifier. As was the case for shadow-mapping, artifacts along a border can be reduced by setting the texture wrap mode to “clamp to edge.” In this case, it can help reduce seams. Note that this is set for all three texture coordinates: s , t , and r .

The `init()` function now includes a call to enable `GL_TEXTURE_CUBE_MAP_SEAMLESS`, which tells OpenGL to attempt to blend adjoining edges of the cube to reduce or eliminate seams. In `display()`, the cube’s vertices are sent down the pipeline as before, but this time it is unnecessary to send the cube’s texture coordinates. As we will see, this is because an OpenGL texture cube map usually uses the cube’s vertex positions as its texture coordinates. The view matrix is “flipped” on the Y and Z axis (with a scale function) to account for the reversed orientation described above. After disabling depth testing, the cube is drawn. Depth testing is then re-enabled for the rest of the scene.

The texture is accessed in the fragment shader with a special type of sampler called a `samplerCube`. In a texture cube map, the value returned from the sampler is the texel “seen” from the origin as viewed along the direction vector (s,t,r) . As a result, we can usually simply use the incoming interpolated vertex positions as the texture coordinates. In the vertex shader, we assign the cube vertex positions into the outgoing texture coordinate attribute, so that they will be interpolated when they reach the fragment shader. Note also in the vertex shader that we convert the incoming view matrix to 3×3 , and then back to 4×4 . This “trick” effectively

removes the translation component, while retaining the rotation (recall that translation values are found in the 4th column of a transformation matrix). This fixes the cube map at the default camera location, while still allowing the synthetic camera to “look around.”

Program 9.2 OpenGL Cube Map Skybox

Java/Jogl application

```
public void init(GLAutoDrawable drawable)
{ // same as before, plus the following:
    ...
    textureID2 = loadCubeMap();
    gl.glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
}

public void display(GLAutoDrawable drawable)
{ // clear color and depth buffers, projection and camera view matrix as before.
    ...
    // draw cube map first - note that it requires a different rendering program
    gl.glUseProgram(rendering_program_cube_map);

    // put the V matrix into the corresponding uniform
    cubeV_matrix = (Matrix3D) v_matrix.clone();
    cubeV_matrix.scale(1.0, -1.0, -1.0); // flip view matrix for reversed axes in OpenGL cube map
    v_location = gl.glGetUniformLocation(rendering_program_cube_map, "v_matrix");
    gl glUniformMatrix4fv(v_location, 1, false, cubeV_matrix.getFloatValues(), 0);

    // put the P matrix into the corresponding uniform
    int ploc = gl.glGetUniformLocation(rendering_program_cube_map, "p_matrix");
    gl glUniformMatrix4fv(ploc, 1, false, proj_matrix.getFloatValues(), 0);

    // set up vertices buffer for cube (buffer for texture coordinates not necessary)
    gl glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[2]);
    gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    gl glEnableVertexAttribArray(0);

    // make the cube map the active texture
    gl glActiveTexture(GL_TEXTURE0);
    gl glBindTexture(GL_TEXTURE_CUBE_MAP, textureID2);

    // disable depth testing, and then draw the cube map
    gl glEnable(GL_CULL_FACE);
    gl glFrontFace(GL_CCW);
    gl glDisable(GL_DEPTH_TEST);
```

```

gl.glDrawArrays(GL_TRIANGLES, 0, 36);
gl.glEnable(GL_DEPTH_TEST);

// draw remainder of the scene
...
}

private int loadCubeMap()
{
    GL4 gl = (GL4) GLContext.getCurrentGL();
    GLProfile glp = gl.getGLProfile();
    Texture tex = new Texture(GL_TEXTURE_CUBE_MAP);

    try
    { // get images from image files
        TextureData topFile = TextureIO.newTextureData(glp, new File("code/top.jpg"), false, "jpg");
        TextureData leftFile = TextureIO.newTextureData(glp, new File("code/left.jpg"), false, "jpg");
        TextureData fntFile = TextureIO.newTextureData(glp, new File("code/center.jpg"), false, "jpg");
        TextureData rightFile = TextureIO.newTextureData(glp, new File("code/right.jpg"), false, "jpg");
        TextureData bkFile = TextureIO.newTextureData(glp, new File("code/back.jpg"), false, "jpg");
        TextureData botFile = TextureIO.newTextureData(glp, new File("code/bottom.jpg"), false, "jpg");

        // attach textures to each face of the active texture
        tex.updateImage(gl, rightFile, GL_TEXTURE_CUBE_MAP_POSITIVE_X);
        tex.updateImage(gl, leftFile, GL_TEXTURE_CUBE_MAP_NEGATIVE_X);
        tex.updateImage(gl, botFile, GL_TEXTURE_CUBE_MAP_POSITIVE_Y);
        tex.updateImage(gl, topFile, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y);
        tex.updateImage(gl, fntFile, GL_TEXTURE_CUBE_MAP_POSITIVE_Z);
        tex.updateImage(gl, bkFile, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z);

    } catch (IOException|GLEException e) { e.printStackTrace(); }

    int[] textureIDs = new int[1];
    gl glGenTextures(1, textureIDs, 0);
    int textureID = tex.getTextureObject();

    // any texture parameter settings go here, such as...
    gl.glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    gl.glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    gl.glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    return textureID;
}

```

Vertex shader

```

#version 430
layout (location = 0) in vec3 position;
out vec3 tc;

```

```

uniform mat4 v_matrix;
uniform mat4 p_matrix;
layout (binding = 0) uniform samplerCube samp;

void main(void)
{
    tc = position;                                // texture coordinates are simply the vertex coordinates
    mat4 vrot_matrix = mat4(mat3(v_matrix));        // removes translation from view matrix
    gl_Position = p_matrix * vrot_matrix * vec4(position, 1.0);
}

```

Fragment shader

```

#version 430
in vec3 tc;
out vec4 fragColor;

uniform mat4 v_matrix;
uniform mat4 p_matrix;
layout (binding = 0) uniform samplerCube samp;

void main(void)
{
    fragColor = texture(samp,tc);
}

```

9.4 ENVIRONMENT MAPPING

When we looked at lighting and materials, we considered the “shininess” of objects. However, we never modeled *very shiny* objects, such as a mirror or something made out of chrome. Such objects don’t just have small specular highlights; they actually reflect their surroundings. When we look at them, we see things in the room, or sometimes even our own reflection. The ADS lighting model doesn’t provide a way of simulating this effect.

Texture cube maps, however, offer a relatively simple way to simulate reflective surfaces—at least partially. The trick is to *use the cube map to texture the reflective object itself*¹. Doing this so that it appears realistic requires finding texture coordinates that correspond to which part of the surrounding environment we should see reflected in the object from our vantage point.

¹ This can also be done with a skydome.

Figure 9.9 illustrates the strategy of using a combination of the view vector and the normal vector to calculate a *reflection vector* which is then used to look up a texel from the cube map. The reflection vector can thus be used to access the texture cube map directly. When the cube map performs this function, it is referred to as an *environment map*.

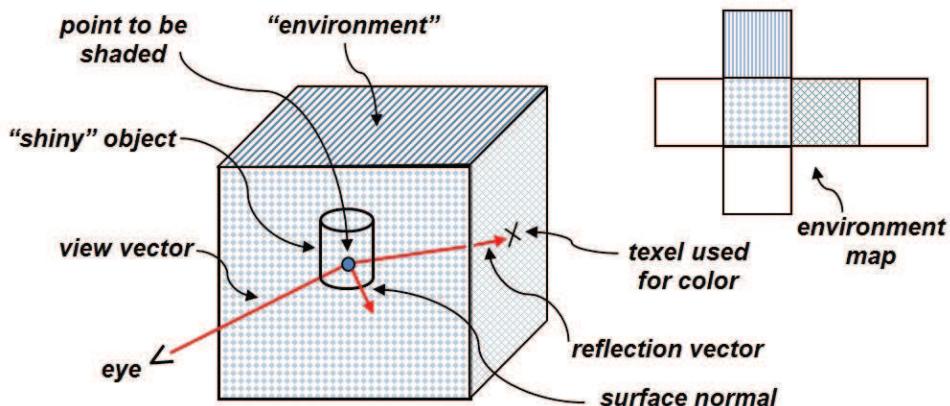


Figure 9.9
Environment mapping overview.

We computed reflection vectors earlier when we studied Blinn-Phong lighting. The concept here is similar, except that now we are using the reflection vector to look up a value from a texture map. This technique is called *environment mapping*, or *reflection mapping*. If the cube map is implemented using the second method we described (in Section 9.3.2; that is, as an OpenGL `GL_TEXTURE_CUBE_MAP`), then OpenGL can perform the environment mapping lookup in the same manner as was done for texturing the cube map itself. We use the view vector and the surface normal to compute a reflection of the view vector off the object's surface. The reflection vector can then be used to sample the texture cube map image directly. The lookup is facilitated by the OpenGL `samplerCube`; recall from the previous section that the `samplerCube` is indexed by a view direction vector. The reflection vector is thus well-suited for looking up the desired texel.

The implementation requires a relatively small amount of additional code. Program 9.3 shows the changes that would be made to the `display()` and `init()` functions and the relevant shaders for rendering a “reflective” torus using environment mapping. The changes are highlighted. It is worth noting that if Blinn-Phong

lighting is present, many of these additions would likely already be present. The only truly new section of code is in the fragment shader (in the `main()` method).

In fact, it might at first appear that the highlighted code in Program 9.3 (i.e., the yellow sections) aren't really new at all. Indeed, we have seen nearly identical code before, when we studied lighting. However, in this case, the normal and reflection vectors are used for an entirely different purpose. Previously they were used to implement the ADS lighting model. Here they are instead used to compute texture coordinates for environment mapping. We highlighted these lines of code so that the reader can more easily track the use of normals and reflection computations for this new purpose.

The result, showing an environment-mapped “chrome” torus within a cube map of storm clouds, is shown in Figure 9.10.

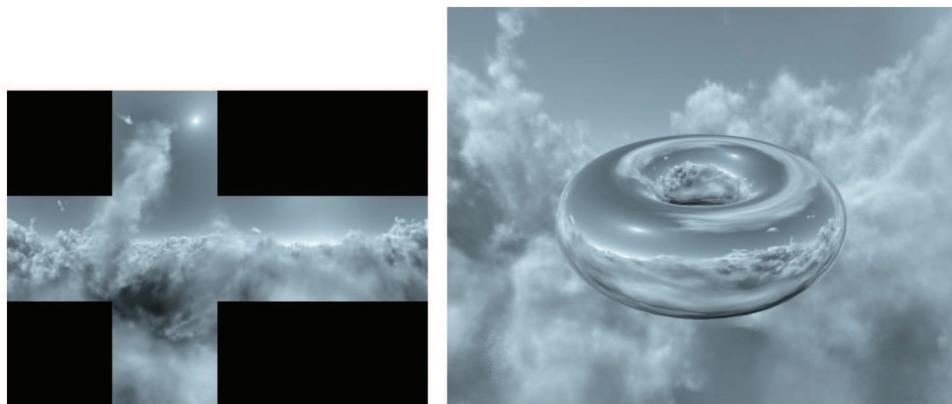


Figure 9.10

Example of environment mapping to create a reflective torus.

Program 9.3 Environment Mapping

```
public void display(GLAutoDrawable drawable)
{ // the code for drawing the cube map is unchanged.
    ...
    // the changes are all in drawing the torus:
    gl.glUseProgram(rendering_program);

    // uniform locations for matrix transforms, including the transform for normals
    mv_location = gl.glGetUniformLocation(rendering_program, "mv_matrix");
```

```

proj_location = gl.glGetUniformLocation(rendering_program, "proj_matrix");
n_location = gl.glGetUniformLocation(rendering_program, "norm_matrix");

// build the MODEL matrix, as before
m_matrix.setTIdentity();
m_matrix.translate(torusLocX, torusLocY, torusLocZ);

// build the MODEL-VIEW matrix, as before
mv_matrix.setTIdentity();
mv_matrix.concatenate(v_matrix);
mv_matrix.concatenate(m_matrix);

// the normals transform is now included in the uniforms:
gl glUniformMatrix4fv(mv_location, 1, false, mv_matrix.getFloatValues(), 0);
gl glUniformMatrix4fv(proj_location, 1, false, proj_matrix.getFloatValues(), 0);
gl glUniformMatrix4fv(n_location, 1, false, ((mv_matrix.inverse()).transpose()).getFloatValues(), 0);

// activate the torus vertices buffer, as before
gl glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[0]);
gl glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(0);

// we need to activate the torus normals buffer:
gl glBindBuffer(GL_ARRAY_BUFFER, bufferIDs[2]);
gl glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
gl glEnableVertexAttribArray(1);

// the torus texture is now the cube map
gl setActiveTexture(GL_TEXTURE0);
gl glBindTexture(GL_TEXTURE_CUBE_MAP, textureID2);

// drawing the torus is otherwise unchanged
gl glClear(GL_DEPTH_BUFFER_BIT);
gl glEnable(GL_CULL_FACE);
gl glFrontFace(GL_CCW);
gl glDepthFunc(GL_EQUAL);
gl glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);
}

```

Vertex shader

```

#version 430
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
out vec3 varyingNormal;
out vec3 varyingVertPos;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

```

```

uniform mat4 norm_matrix;
layout (binding = 0) uniform samplerCube tex_map;

void main(void)
{   varyingVertPos = (mv_matrix * vec4(position,1.0)).xyz;
    varyingNormal = (norm_matrix * vec4(normal,1.0)).xyz;
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}

```

Fragment shader

```

#version 430
in vec3 varyingNormal;
in vec3 varyingVertPos;
out vec4 fragColor;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
layout (binding = 0) uniform samplerCube tex_map;

void main(void)
{   vec3 r = reflect(normalize(-varyingVertPos), normalize(varyingNormal));
    fragColor = texture(tex_map, r);
}

```

Although two sets of shaders are required for this scene—one set for the cube map and another set for the torus—only the shaders used to draw the torus are shown in Program 9.3. This is because the shaders used for rendering the cube map are unchanged from Program 9.2. The changes made to Program 9.2, resulting in Program 9.3, are summarized as follows:

in init():

- A buffer of normals for the torus is created (actually done in `setupVertices()`, called by `init()`).
- The buffer of texture coordinates for the torus is no longer needed.

in display():

- The matrix for transforming normals (dubbed “`norm_matrix`” in Chapter 7) is created and linked to the associated uniform variable.
- The torus normal buffer is activated.
- The texture cube map is activated as the texture for the torus (rather than the “brick” texture).

in the vertex shader:

- The normal vectors and `norm_matrix` are added.
- The transformed vertex and normal vector are output in preparation for computing the reflection vector, similar to what was done for lighting and shadows.

in the fragment shader:

- The reflection vector is computed in a similar way to what was done for lighting.
- The output color is retrieved from the texture (now the cube map), with the lookup texture coordinate now being the reflection vector.

The resulting rendering shown in Figure 9.10 is an excellent example of how a simple trick can achieve a powerful illusion. By simply painting the background on an object, we have made the object look “metallic,” when no such ADS material modeling has been done at all. It has also given the appearance that light is reflecting off of the object (there even seem to be specular highlights), even though no ADS lighting whatsoever has been incorporated into the scene!

SUPPLEMENTAL NOTES

A major limitation of environment mapping, as presented in this chapter, is that it is only capable of constructing objects that reflect the cube map. Other objects rendered in the scene are not reflected in the reflection-mapped object. Depending on the nature of the scene, this might or might not be acceptable. If other objects are present that must be reflected in a mirror or chrome object, other methods must be used. A common approach utilizes the stencil buffer (mentioned earlier in Chapter 8), and is described in various web tutorials ([OV12], [NE14], and [GR16], for example), but is outside the scope of this text.

We didn’t include an implementation of skydomes, although they are in some ways arguably simpler than skyboxes and they can be less susceptible to distortion. Even environment mapping is simpler—at least the math—but the OpenGL support for cube maps often makes skyboxes more practical.

Of the topics covered in the later sections of this textbook, skyboxes and skydomes are arguably among the simplest conceptually. However, getting them to look convincing can consume a lot of time. We have dealt only briefly with

some of the issues that can arise (such as seams), but depending on the texture image files used, other issues can occur, requiring additional repair. This is especially true when the scene is animated, or when the camera can be moved interactively.

We also glossed over the generation of usable and convincing texture cube map images. There are excellent tools for doing this, one of the most popular being Terragen [TE16]. Several of the cube maps in this chapter were made using Terragen, including those in Figure 9.7, made by the authors.

Exercises

- 9.1 (PROJECT)** In Program 9.2, add the ability to move the camera around with the mouse. To do this, you will need to utilize the code you developed earlier in Exercise 4.2 for constructing a view matrix. You'll also need to assign mouse or keyboard actions to functions that move the camera forward and backward, and functions that rotate the camera on one or more of its axes (you'll need to write these functions too). After doing this, you should be able to "fly around" in your scene, noting that the skybox always appears to remain at the distant horizon.
- 9.2 (PROJECT)** Add animation to Program 9.3 so that one (or more) environment-mapped object(s) in the scene rotate or tumble. The simulated reflectivity of the object should be apparent as the skybox texture moves on the object's surface.
- 9.3 (PROJECT)** Modify Program 9.3 so that the object in the scene blends environment-mapping with a texture. Use a weighted sum in the fragment shader, as described in Chapter 7.
- 9.4 (RESEARCH & PROJECT)** Learn the basics of how to use Terragen [TE16] to create a simple cube map. This generally entails making a "world" with the desired terrain and atmospheric patterns (in Terragen), and then positioning Terragen's synthetic camera to save six images representing the views front, back, right, left, top, and bottom. Use your images in Program 9.2 and Program 9.3 to see their appearance as cube maps, and with environment mapping. The free version of Terragen is quite sufficient for this exercise.

References

- [BO01] P. Bourke, “Representing Star Fields,” June 2001, accessed July 2016, <http://paulbourke.net/miscellaneous/starfield/>.
- [GI16] GNU Image Manipulation Program, accessed July 2016, <http://www.gimp.org>.
- [GR16] OpenGL Resources, “Planar Reflections and Refractions Using the Stencil Buffer,” accessed July 2016, <https://www.opengl.org/archives/resources/code/samples/advanced/advanced97/notes/node90.html>.
- [NE14] NeHeProductions, “Clipping and Reflections Using the Stencil Buffer,” 2014, accessed July 2016, http://nehe.gamedev.net/tutorial/clipping_reflections_using_the_stencil_buffer/17004/.
- [OV12] A. Overvoorde, “Depth and Stencils,” 2012, accessed July 2016, <https://open.gl/depthstencils>.
- [TE16] Terragen, Planetside Software, LLC, accessed July 2016, <http://planetside.co.uk/>.

ENHANCING SURFACE DETAIL

10.1	<i>Bump Mapping</i>	232
10.2	<i>Normal Mapping</i>	233
10.3	<i>Height Mapping</i>	242
	<i>Supplemental Notes</i>	246



Suppose we want to model an object with an irregular surface—like the bumpy surface of an orange, the wrinkled surface of a raisin, or the cratered surface of the moon. How would we do it? So far, we have learned two potential methods: (a) we could model the entire irregular surface, which would often be impractical (a highly cratered surface would require a huge number of vertices), or (b) we could apply a texture-map image of the irregular surface to a smooth version of the object. The second option is often effective. However, if the scene includes lights, and the lights (or camera angle) move, it becomes quickly obvious that the object is statically textured (and smooth) because the light and dark areas on the texture wouldn't change, as they would if the object was actually bumpy.

In this chapter we are going to explore several related methods for using lighting effects to make objects *appear* to have realistic surface texture, even if the underlying object model is smooth. We will start by examining *bump mapping* and *normal mapping*, which can add considerable realism to the objects in our scenes when it would be too computationally expensive to include tiny surface details in the object models. We will also look at ways of actually perturbing the vertices in a smooth surface through *height mapping*, which is useful for generating *terrain* (and other uses).

10.1 BUMP MAPPING

In Chapter 7, we saw how surface normals are critical to creating convincing lighting effects. Light intensity at a pixel is determined largely by the reflection angle, taking into account the light source location, camera location, and the normal vector at the pixel. Thus, we can avoid generating detailed vertices corresponding to a bumpy or wrinkled surface if we can find a way of generating the corresponding normals.

Figure 10.1 illustrates the concept of modified normals corresponding to a single “bump.”

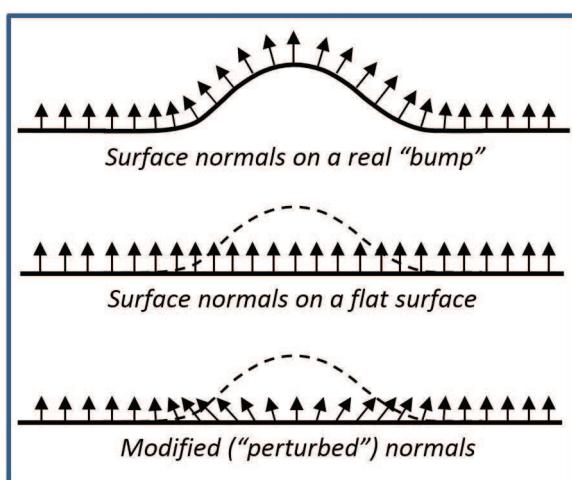


Figure 10.1

Perturbed normal vectors for bump mapping.

Thus, if we want to make an object look as though it has bumps (or wrinkles, craters, etc.), one way is to compute the *normals* that would exist on such a surface. Then when the scene is lit, the lighting would produce the desired illusion. This was first proposed by Blinn in 1978 [BL78], and became practical with the advent of the capability of performing per-pixel lighting computations in a fragment shader.

An example is illustrated in the vertex and fragment shaders shown in Program 10.1, which produces a torus with a “golf-ball” surface as shown

in the Figure 10.2 that follows. The code is almost identical to Program 7.2 that we saw previously in Chapter 7. The only significant change is in the fragment shader—the incoming interpolated normal vectors (named “varyingNormal” in the original program) are altered with bumps calculated using a sine wave function in the X, Y, and Z axes applied to the original (untransformed) vertices of the torus model. Note that the vertex shader therefore now needs to pass these untransformed vertices down the pipeline.

Altering the normals in this manner, with a mathematical function computed at runtime, is called *procedural bump mapping*.

Program 10.1 Procedural Bump Mapping

Vertex Shader

```
#version 430
// same as Phong shading, but add this output vertex attribute:
out vec3 originalVertex;
...
void main(void)
{ // include this pass-through of original vertex for interpolation:
    originalVertex = vertPos;
...
}
```

Fragment Shader

```
#version 430
// same as Phong shading, but add this input vertex attribute:
in vec3 originalVertex;
...
void main(void)
{
    // add the following to perturb the incoming normal vector:
    float a = 0.25; // a controls height of bumps
    float b = 100.0; // b controls width of bumps
    float x = originalVertex.x;
    float y = originalVertex.y;
    float z = originalVertex.z;
    N.x = varyingNormal.x + a*sin(b*x); // perturb incoming normal using sine function
    N.y = varyingNormal.y + a*sin(b*y);
    N.z = varyingNormal.z + a*sin(b*z);
    N = normalize(N);
    // lighting computations and output fragColor (unchanged) now utilize the perturbed normal N
    ...
}
```

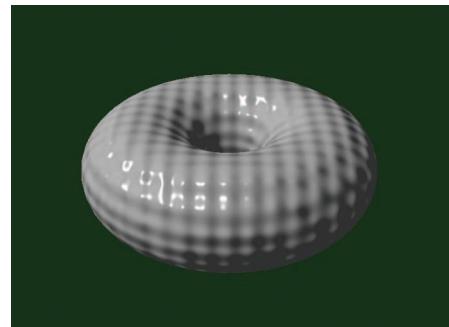


Figure 10.2
Procedural bump mapping example.

10.2 NORMAL MAPPING

An alternative to bump mapping is to replace the normals using a lookup table. This allows us to construct bumps for which there is no mathematical function, such as the bumps corresponding to the craters on the moon. A common way of doing this is called *normal mapping*.

To understand how this works, we start by noting that a vector can be stored to reasonable precision in three bytes, one for each of the X, Y, and Z components. This makes it possible to store normals in a color image file, with the R, G, and B components corresponding to X, Y, and Z. RGB values in an image are stored in bytes and are usually interpreted as values in the range [0...1], whereas vectors can have positive or negative component values. If we restrict normal vector components to the range [-1...+1], a simple conversion to enable storing a normal vector \mathbf{N} as a pixel in an image file is:

$$\begin{aligned} R &= (N_x + 1)/2 \\ G &= (N_y + 1)/2 \\ B &= (N_z + 1)/2 \end{aligned}$$

Normal mapping utilizes an image file (called a *normal map*) that contains normals corresponding to a desired surface appearance in the presence of lighting. In a normal map, the vectors are represented relative to an arbitrary plane X-Y, with their X and Y components representing deviations from “vertical,” and their Z component set to 1. A vector strictly perpendicular to the X-Y plane (i.e., with no deviation) would be represented [0,0,1], whereas non-perpendicular vectors would have non-zero X and/or Y components. We use the above formulae to convert to RGB space; for example, [0,0,1] would be stored as [.5,.5,1], since actual offsets range [-1..+1], but RGB values range [0..1].

We can make use of such a normal map through yet another clever application of texture units: instead of storing colors in the texture unit, we store the desired normal vectors. We can then use a sampler to look up the value in the normal map for a given fragment, and then rather than applying the returned value to the output pixel color (as we did in texture mapping), we instead use it as the normal vector.

One example of such a normal map image file is shown in Figure 10.3. It was generated by applying the GIMP normal mapping plugin [GI16] to a texture from Luna [LU16]. Normal-mapping image files are not intended for viewing; we show this one to point out that such images end up being largely blue. This is because every entry in the image file has a B value of 1 (maximum blue), making the image appear “bluish” if viewed.

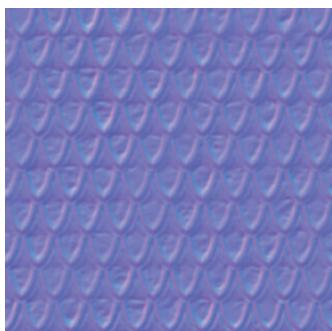


Figure 10.3

Normal mapping image file example [ME11].

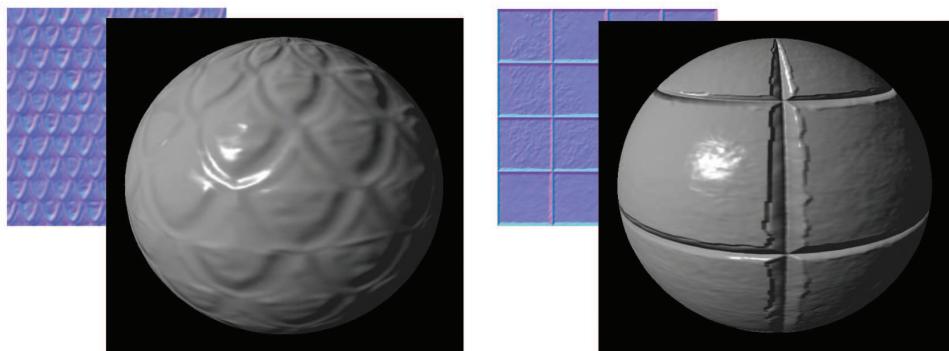


Figure 10.4
Normal mapping examples.

Figure 10.4 shows two different normal map image files (both are built out of textures from Luna [LU16]), and the result of applying them to a sphere in the presence of Blinn-Phong lighting.

Normal vectors retrieved from a normal map cannot be utilized directly, because they are defined relative to an arbitrary X-Y plane as described above, without taking into account their position on the object and their orientation in camera space. Our strategy for addressing this will be to build a transformation matrix for converting the normals into camera space, as follows.

At each vertex on an object, we consider a plane that is tangent to the object. The object normal at that vertex is perpendicular to this plane. We define two mutually perpendicular vectors in that plane, also perpendicular to the normal, called the *tangent* and *bitangent* (sometimes called the *binormal*). Constructing our desired transformation matrix requires that our models include a tangent vector for each vertex (the bitangent can be built by computing the cross product of the tangent and the normal). If the model does not already have tangent vectors defined, they could be computed. In the case of a sphere they can be computed exactly, as shown in the following modifications to Program 6.1 from Chapter 6:

```
...
for (int i=0; i<=prec; i++)
{
    for (int j=0; j<=prec; j++)
    {
        float y = (float)cos(toRadians(180-i*180/(prec+1)));
        float x = (float)cos(toRadians(j*360/(prec+1))) * (float)abs(cos(asin(y)));
```

```

float z = (float)sin(toRadians(j*360/(prec+1))) * (float)abs(cos(asin(y)));
vertices[i*(prec+1)+j].setLocation(new Point3D(x,y,z));

// calculate tangent vector
if (((x==0) && (y==1) && (z==0)) || ((x==0) && (y==-1) && (z==0))) // if north or south pole,
{   tangent = new Vector3D(0,0,-1);                                     // set tangent to -Z axis
}
else
{   tangent = (new Vector3D(0,1,0)).cross(new Vector3D(x,y,z)); // otherwise, calculate tangent
}
vertices[i*(prec+1)+j].setTangent(tangent); // store the tangent in vertex object
... // remaining computations are unchanged
} }

```

For models that don't lend themselves to exact analytic derivation of surface tangents, the tangents can be approximated, for example by drawing vectors from each vertex to the next as they are constructed (or loaded). Note that such an approximation can lead to tangent vectors that are not strictly perpendicular to the corresponding vertex normals. Implementing normal mapping that works across a variety of models therefore needs to take this possibility into account (our solution will).

The tangent vectors are sent from a buffer (VBO) to a vertex attribute in the vertex shader, as is done for the vertices, texture coordinates, and normals. The vertex shader then processes them the same as is done for normal vectors, by applying the inverse-transpose of the MV matrix, and forwarding the result down the pipeline for interpolation by the rasterizer and ultimately into the fragment shader. This last step converts the normal and tangent vectors into camera space, after which we construct the bitangent using the cross product.

Once we have the normal, tangent, and bitangent vectors in camera space, we can use them to construct a matrix (called the “TBN” matrix, after its components) which transform the normals retrieved from the normal map into their corresponding orientation in camera space relative to the surface of the object.

In the fragment shader, the computing of the new normal is done in the calcNewNormal() function. The computation in the third line of the function (the one containing `dot(tangent, normal)`) ensures that the tangent vector is perpendicular to the normal vector. A cross product between the new tangent and the normal produces the bitangent.

We then create TBN as a 3×3 `mat3` matrix. The `mat3` constructor takes three vectors and generates a matrix containing the first vector in the top row, the second vector in the middle row, and the third in the bottom row (similar to building a view matrix from camera space—see Figure 3.13).

The shader uses the fragment’s texture coordinates to extract the normal map entry corresponding to the current fragment. The sampler variable “`normMap`” is used for this, and in this case is bound to texture unit 0 (note: the Java/JOGL application must therefore have attached the normal map image to texture unit 0). To convert the color component from the stored range $[0\dots 1]$ to its original range $[-1\dots +1]$ we multiply by 2.0 and subtract 1.0.

The TBN matrix is then applied to the resulting normal to produce the final normal for the current pixel. The rest of the shader is identical to the fragment shader used for Phong lighting. The fragment shader is shown in Program 10.2, and is based on a version by Etay Meiri [ME11].

Program 10.2 also contains our first example of a GLSL *function* (aside from the “`main`”). As in the C language, functions must be defined before (or “above”) where they are called, or else a forward declaration must be provided. In this example a forward declaration isn’t required because the function has been defined above the call to it.

A variety of tools exist for developing normal map images. Some image editing tools, such as GIMP [GI16] and Photoshop [PH16], have such capabilities. Such tools analyze the edges in an image, inferring peaks and valleys, and producing a corresponding normal map.

Figure 10.5 shows a texture map of the surface of the moon created by Hastings-Trew [HT16] based on NASA satellite data. The corresponding normal map was generated by applying the GIMP normal map plugin [GP16] to a black and white reduction also created by Hastings-Trew.

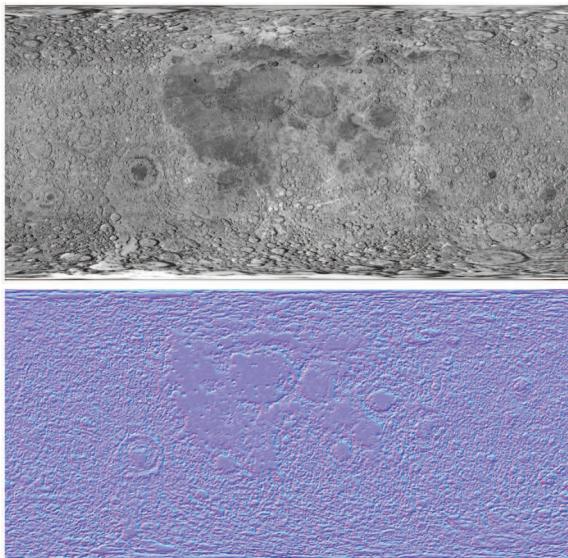


Figure 10.5
Moon, texture and normal map.

Program 10.2 Normal Mapping Fragment Shader

```

#version 430
in vec3 varyingLightDir;
in vec3 varyingVertPos;
in vec3 varyingNormal;
in vec3 varyingTangent;
in vec3 originalVertex;
in vec2 tc;
out vec4 fragColor;

layout (binding=0) uniform sampler2D normMap;
// remaining uniforms same as before
...
vec3 calcNewNormal()
{
    vec3 normal = normalize(varyingNormal);
    vec3 tangent = normalize(varyingTangent);
    tangent = normalize(tangent - dot(tangent, normal) * normal); // tangent is perpendicular to normal
    vec3 bitangent = cross(tangent, normal);
    mat3 tbn = mat3(tangent, bitangent, normal); // TBN matrix to convert to camera space
    vec3 retrievedNormal = texture(normMap,tc).xyz;
    retrievedNormal = retrievedNormal * 2.0 - 1.0;           // convert from RGB space
    vec3 newNormal = tbn * retrievedNormal;
    newNormal = normalize(newNormal);
    return newNormal;
}

void main(void)
{
    // normalize the light, normal, and view vectors:
    vec3 L = normalize(varyingLightDir);
    vec3 V = normalize(-varyingVertPos);
    vec3 N = calcNewNormal();

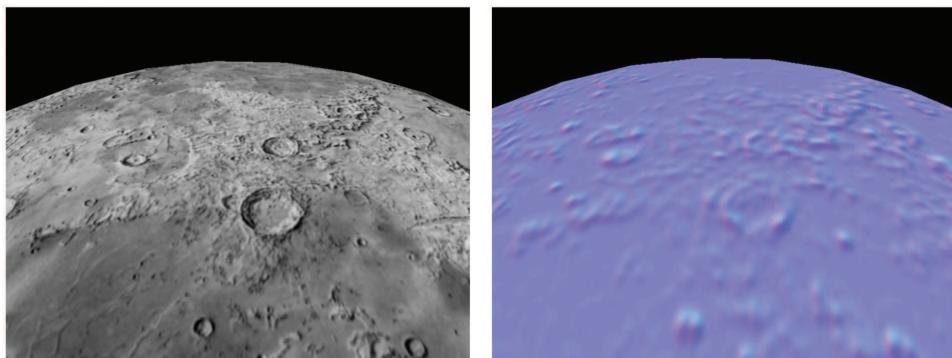
    // get the angle between the light and surface normal:
    float cosTheta = dot(L,N);

    // compute half vector for Blinn optimization:
    vec3 H = varyingHalfVector;

    // angle between the view vector and reflected light:
    float cosPhi = dot(H,N);

    // compute ADS contributions (per pixel):
    fragColor = globalAmbient * material.ambient
        + light.ambient * material.ambient
        + light.diffuse * material.diffuse * max(cosTheta,0.0)
        + light.specular * material.specular * pow(max(cosPhi,0.0), material.shininess);
}

```

**Figure 10.6**

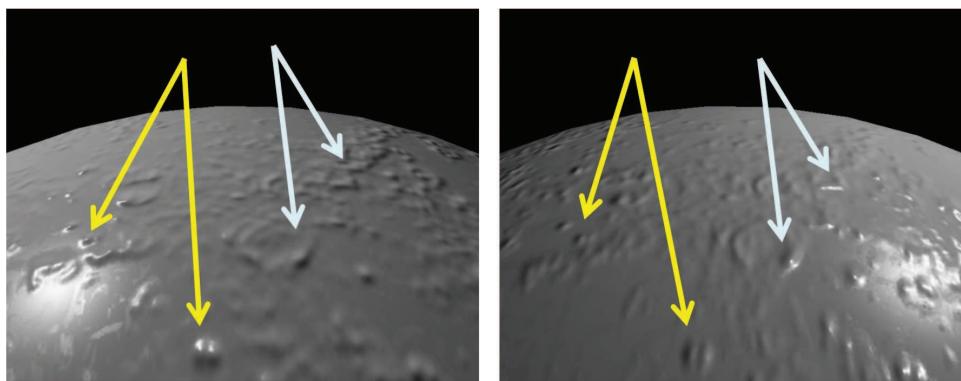
Sphere textured with moon texture (left) and normal map (right).

Figure 10.6 shows a sphere with the moon surface rendered in two different ways. On the left, simply textured with the original texture map; on the right, textured with the image normal map (for reference). Normal mapping has not been applied in either case. As realistic as the textured “moon” is, close examination reveals that the texture image was apparently taken when the moon was being lit from the left, because ridge shadows are cast to the right (most clearly evident in the crater at the bottom center). If we were to add lighting to this scene with Phong shading, and then animate the scene by moving the moon, the camera, or the light, those shadows will not change as we would expect them to.

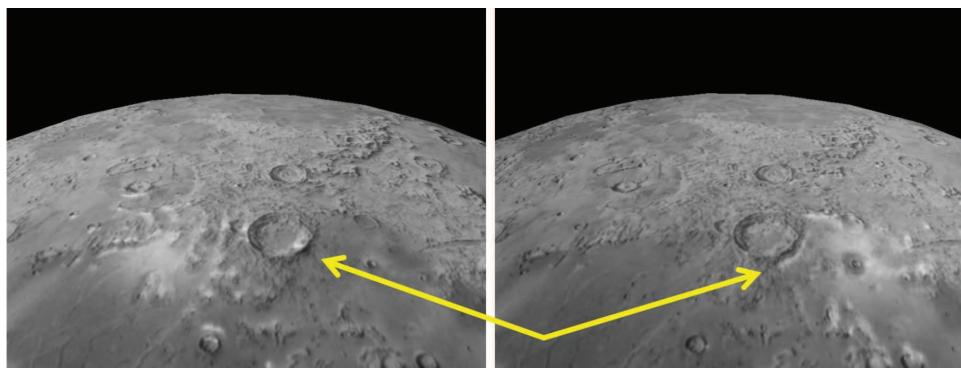
Furthermore, as the light source moves (or as the camera moves), we would expect many specular highlights to appear on the ridges. But a plain textured sphere such as at the left of Figure 10.6 would produce only one specular highlight, corresponding to what would appear on a smooth sphere, which would look very unrealistic. Incorporation of the normal map can improve the realism of lighting on objects such as this considerably.

If we use normal mapping on the sphere (rather than applying the texture), we obtain the results shown in Figure 10.7. Although not as realistic (yet) as standard texturing, it now *does* respond to lighting changes. The first image is lit from the left, and the second is lit from the right. Note the blue and yellow arrows showing the change in diffuse lighting around ridges and the movement of specular highlights.

Figure 10.8 shows the effect of combining normal mapping with standard texturing, in the presence of Phong lighting. The image of the moon is enhanced with diffuse-lit regions and specular highlights that respond to the movement of the light source (or camera or object movement). Lighting in the two images is from the left and right sides, respectively.

**Figure 10.7**

Normal map lighting effects on moon.

**Figure 10.8**

Texturing plus normal mapping, with lighting from the left and right.

Our program now requires two textures—one for the texture and one for the normal map—and thus two samplers. The fragment shader blends the texture color with the color produced by the lighting computation as shown in Program 10.3, with adjustable weights highlighted.

Program 10.3 Texturing Plus Normal Map

```
// variables and structs as in previous fragment shader, plus:  
layout (binding=0) uniform sampler2D s0;           // normal map  
layout (binding=1) uniform sampler2D s1;           // texture
```

```

void main(void)
{
    // computations same as before, until:

    vec3 N = calcNewNormal();
    vec4 texel = texture(s1,tc); // standard texture
    ...

    // reflection computations as before, then blend results:
    fragColor = 0.6 * texel
        + 0.4 * (global.ambient * material.ambient
        + light.ambient * material.ambient
        + light.diffuse * material.diffuse * max(cosTheta,0.0)
        + light.specular * material.specular * pow(max(cosPhi,0.0), material.shininess));
}

```

Interestingly, normal mapping can benefit from *mipmapping*, because the same “aliasing” artifacts that we saw in Chapter 5 for texturing also occur when using a texture image for normal mapping. Figure 10.9 shows a normal-mapped moon, with and without mipmapping. Although not easily shown in a still image, the sphere at the left (*not* mipmapped) has shimmering artifacts around its perimeter.

Anisotropic filtering (AF) works even better, reducing sparkling artifacts while preserving detail, as illustrated in Figure 10.10. A version combining equal parts texture and lighting with normal mapping and AF is shown alongside, in Figure 10.11.

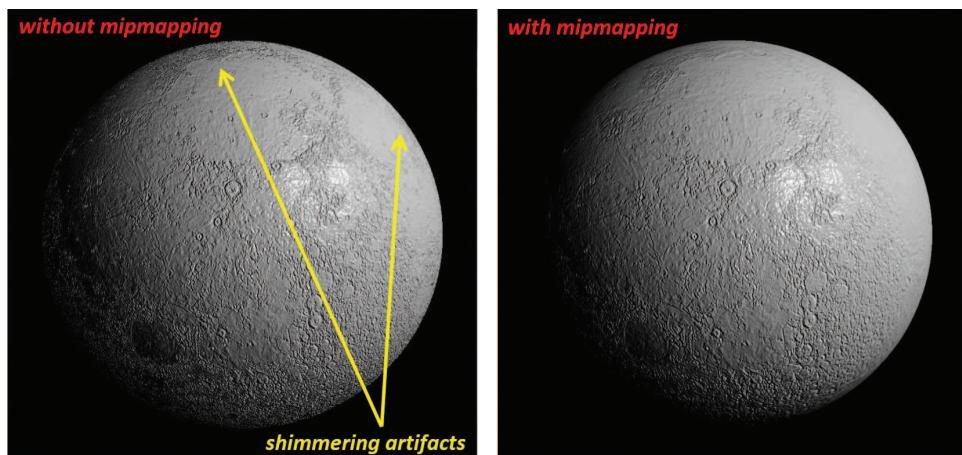


Figure 10.9

Normal mapping artifacts, corrected with mipmapping.

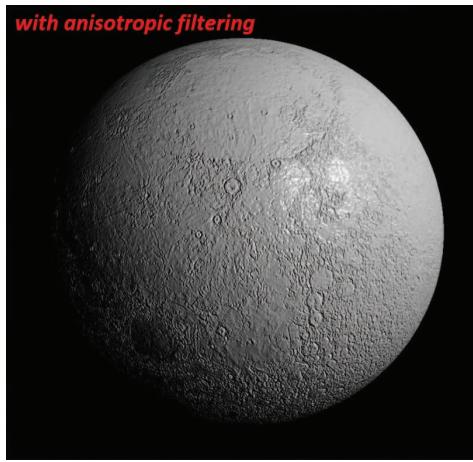


Figure 10.10
Normal mapping with AF.



Figure 10.11
Texturing + normal mapping w/AF.

The results are imperfect. Shadows appearing in the original texture image will still show on the rendered result, regardless of lighting. Also, while normal mapping can affect diffuse and specular effects, it cannot cast shadows. Therefore, this method is best used when the surface features are small.

10.3 HEIGHT MAPPING

We now extend the concept of normal mapping—where a texture image is used to perturb normals—to instead *perturb the vertex locations themselves*. Actually modifying an object’s geometry in this way has certain advantages, such as making the surface features visible along the object’s edge, and enabling the features to respond to shadow-mapping. It can also facilitate building *terrain*, as we will see.

A practical approach is to use a texture image to store *height* values, which can then be used to raise (or lower) vertex locations. An image that contains height information is called a *height map*, and using a height map to alter an object’s vertices is called *height mapping*. Height maps usually encode height information as grayscale colors: (0,0,0) (black) = *low height*, and (1,1,1) (white) = *high height*. This makes it easy to create height maps algorithmically, or by using a “paint” program. The higher the image contrast, the greater the variation in height expressed by the map. These concepts are illustrated in Figure 10.12 (showing a randomly generated map), and Figure 10.13 (showing a map with an organized pattern).

The usefulness of altering vertex locations depends on the model being altered. Vertex manipulation is easily done in the vertex shader, and when there is a high level of detail in the model vertices (such as in a sphere with sufficiently high precision), this approach can work well. However, when the underlying number of vertices is small (such as the corners of a cube), rendering the object's surface relies on vertex interpolation in the rasterizer to fill in the detail. When there are very few vertices available in the vertex shader to perturb, the heights of many pixels would be interpolated rather than retrieved from the height map, leading to poor surface detail. Vertex manipulation in the fragment shader is, of course, impossible because by then the vertices have been rasterized into pixel locations.

Program 10.4 shows a vertex shader that moves the vertices “outwards” (i.e., in the direction of the surface normal), by multiplying the vertex normal by the value retrieved from the height map and then adding that product to the vertex position.

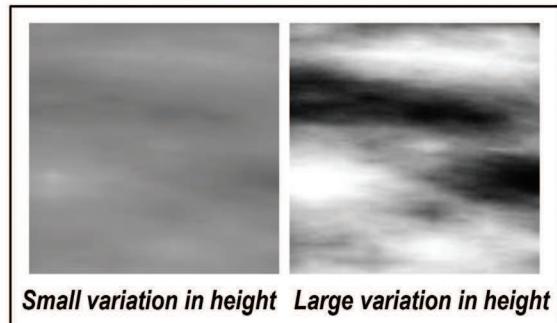


Figure 10.12
Height map examples.

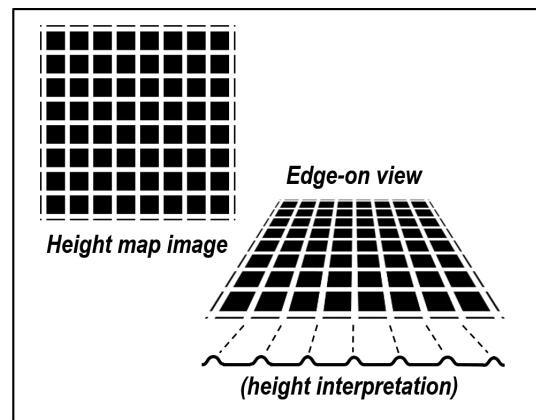


Figure 10.13
Height map interpretation.

Program 10.4 Height Mapping in Vertex Shader

```
#version 430
```

```
layout (location=0) in vec3 vertPos;
layout (location=0) in vec3 vertNormal;
layout (location=0) in vec2 texCoord;

out vec2 tc;
```

```

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

layout(binding=0) uniform sampler2D t;    // for texture
layout(binding=1) uniform sampler2D h;    // for heightmap

void main(void)
{ // "p" is the vertex position altered by the height map.
  // Since the height map is grayscale, any of the color components can be
  // used (we use "r"). Dividing by 5.0 is to adjust the height.
  vec4 p = vec4(vertPos,1.0) + vec4( (vertNormal * ((texture(h, texCoord).r) / 5.0f)),1.0f );
  tc = tex_coord;
  gl_Position = proj_matrix * mv_matrix * p;
}

```

Figure 10.14 shows a simple height map (top left) created by scribbling in a paint program. A white square is also drawn in the height map image. A green-tinted version of the height map (bottom left) is used as a texture. When the height map is applied to a rectangular 100x100 grid model using the shader shown in Program 10.4, it produces a sort of “terrain” (shown on the right). Note how the white square results in the precipice at the right.

Figure 10.15 shows another example of doing height mapping in a vertex shader. This time the height map is an outline of the countries of the world [HT16]. It is applied to a sphere textured with a blue-tinted version of the height

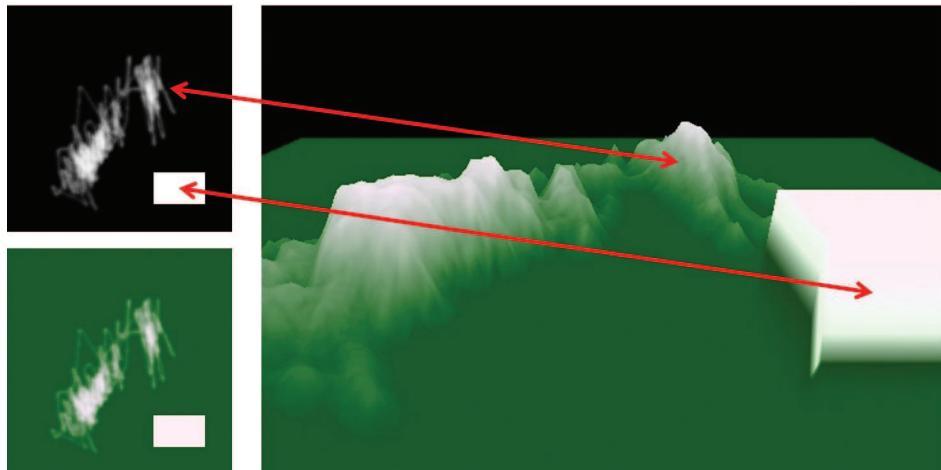
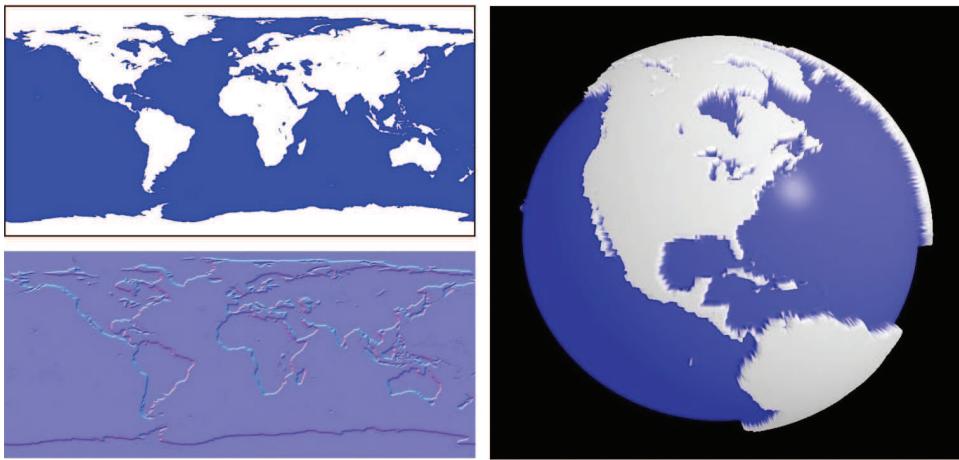


Figure 10.14

Terrain, height-mapped in the vertex shader.

**Figure 10.15**

Vertex shader-based height mapping, applied to a sphere.

map (see top left—note the original black and white version is not shown), and lit with Blinn-Phong shading using a normal map (shown at the lower left) built using the tool *SS_Bump_Generator* [SS16]. The sphere precision was increased to 500 to ensure enough vertices to render the detail. Note how the raised vertices affect not only the lighting, but also the silhouette edges.

The rendered examples shown in Figure 10.14 and Figure 10.15 work acceptably because the two models (grid and sphere) have a sufficient number of vertices to sample the height map values. That is, they each have a fairly large number of vertices, and the height map is relatively coarse and adequately sampled at a low resolution. However, close inspection still reveals the presence of resolution artifacts, such as along the bottom left edge of the raised box at the right of the terrain in Figure 10.14. The reason that the sides of the raised box don't appear perfectly square, and include gradations in color, is because the 100x100 resolution of the underlying grid cannot adequately align perfectly with the white box in the height map, and the resulting rasterization of texture coordinates produces artifacts along the sides.

The limitations of doing height mapping in the vertex shader are further exposed when trying to apply it with a more demanding height map. Consider the moon image shown back in Figure 10.5. Normal mapping did an excellent job of capturing the detail in the image (as shown previously in Figure 10.9 and Figure 10.11),

and since it is grayscale, it would seem natural to try applying it as a height map. However, vertex-shader-based height mapping would be inadequate for this task, because the number of vertices sampled in the vertex shader (even for a sphere with precision=500) is small compared to the fine level of detail in the image. By contrast, normal mapping *was* able to capture the detail impressively, because the normal map is sampled in the fragment shader, at the pixel level.

We will revisit height mapping later in Chapter 12 when we discuss methods for generating a greater number of vertices in a *tessellation shader*.

SUPPLEMENTAL NOTES

One of the fundamental limitations of bump or normal mapping is that, while they are capable of providing the appearance of surface detail in the interior of a rendered object, the silhouette (outer boundary) doesn't show any such detail (it remains smooth). Height mapping, if used to actually modify vertex locations, fixes this deficiency, but has its own limitations. As we will see later in this book, sometimes a geometry or tessellation shader can be used to increase the number of vertices, making height mapping more practical and more effective.

We have taken the liberty of simplifying some of the bump and normal mapping computations. More accurate and/or more efficient solutions are available for critical applications [BN12].

Exercises

- 10.1** Experiment with Program 10.1 by modifying the settings and/or computations in the fragment shader, and observing the results.
- 10.2** Using a paint program, generate your own height map and use it in Program 10.4. See if you can identify locations where detail is missing as the result of the vertex shader being unable to adequately sample the height map. You will probably find it useful to also texture the terrain with your height map image file as shown in Figure 10.14 (or with some sort of pattern that exposes the surface structure, such as a grid), so that you can see the hills and valleys of the resulting terrain.
- 10.3 (PROJECT)** Add lighting to Program 10.4, so that surface structure of the height-mapped terrain is further exposed.

10.4 (PROJECT) Add shadow-mapping to your code from Exercise 10.3 so that your height-mapped terrain casts shadows.

References

- [BL78] J. Blinn, “Simulation of Wrinkled Surfaces,” *Computer Graphics* 12, no. 3 (1978): 286–292.
- [BN12] E. Bruneton and F. Neyret, “A Survey of Non-Linear Pre-Filtering Methods for Efficient and Accurate Surface Shading,” *IEEE Transactions on Visualization and Computer Graphics* 18, no. 2 (2012).
- [GI16] GNU Image Manipulation Program, accessed July 2016, <http://www.gimp.org>.
- [GP16] GIMP Plugin Registry, *normalmap plugin*, accessed July 2016, <http://registry.gimp.org/node/69>.
- [HT16] J. Hastings-Trew, *JHT’s Planetary Pixel Emporium*, accessed July 2016, <http://planetpixelemporium.com/>.
- [LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).
- [ME11] E. Meiri, *OGLdev tutorial 26*, 2011, accessed July 2016, <http://ogldev.atspace.co.uk/index.html>.
- [PH16] Adobe Photoshop, accessed July 2016, <http://www.photoshop.com>.
- [SS16] SS Bump Generator, accessed July 2016, <http://ssbump-generator.yolasite.com/>.

PARAMETRIC SURFACES

11.1	<i>Quadratic Bézier Curves</i>	249
11.2	<i>Cubic Bézier Curves</i>	251
11.3	<i>Quadratic Bézier Surfaces</i>	254
11.4	<i>Cubic Bézier Surfaces</i>	256
	<i>Supplemental Notes</i>	258



While working at the Renault corporation in the 1950s and 1960s, Pierre Bézier developed software systems for designing automobile bodies. His programs utilized mathematical systems of equations developed earlier by Paul de Casteljau, who was working for the competing Citroën automobile manufacturer [BE72, DC63]. The de Casteljau equations describe curves using just a few scalar parameters, and are accompanied by a clever recursive algorithm dubbed “de Casteljau’s algorithm” for generating the curves to arbitrary precision. Now known as “Bézier curves” and “Bézier surfaces,” these methods are commonly used to efficiently model many kinds of curved 3D objects.

11.1 QUADRATIC BÉZIER CURVES

A *Quadratic Bézier curve* is defined by a set of parametric equations that specify a particular curved shape using three *control points*, each of which is a point in 2D space.¹ Consider, for example, the set of three points $[p_0, p_1, p_2]$ shown below in Figure 11.1:

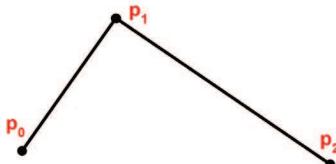


Figure 11.1
Control points for a Bézier curve.

¹ Of course, a curve can exist in 3D space. However, a quadratic curve lies entirely within a 2D plane.

By introducing a parameter t , we can build a system of parametric equations that define a curve. The t represents a fraction of the distance along the

line segment connecting one control point to the next control point. Values for t are within the range [0..1] for points along the segment. Figure 11.2 shows one such value, $t = 0.75$, applied to the lines connecting p_0-p_1 and p_1-p_2 , respectively. Doing this defines two new points $p_{01}(t)$ and $p_{12}(t)$ along the two original lines. We repeat this process for the line segment connecting the two new points $p_{01}(t)$ and $p_{12}(t)$ producing point $P(t)$ where $t = 0.75$ along the line $p_{01}(t)-p_{12}(t)$. $P(t)$ is one of the points on the resulting curve, and for this reason is denoted with a capital P.

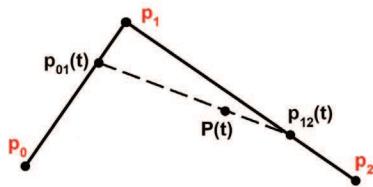


Figure 11.2
Points at parametric position $t = 0.75$.

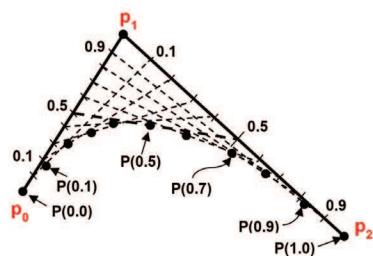


Figure 11.3
Building a quadratic Bézier curve.

Collecting many points $P(t)$ for various values of t generates a curve, as shown in Figure 11.3. The more parameter values for t that are sampled, the more points $P(t)$ are generated, and the smoother the resulting curve.

The analytic definition for a quadratic Bézier curve can now be derived. First, we note that an arbitrary point p on the line segment p_a-p_b connecting two points p_a and p_b can be represented in terms of the parameter t as follows:

$$p(t) = tp_a + (1-t)p_b$$

Using this, we find the points p_{01} and p_{12} (points on p_0-p_1 and p_1-p_2 respectively) as follows:

$$p_{01}(t) = tp_1 + (1-t)p_0$$

$$p_{12}(t) = tp_2 + (1-t)p_1$$

Similarly, a point on the connecting line segment between these points would be:

$$P(t) = tp_{12}(t) + (1-t)p_{01}(t)$$

Substituting the definitions of p_{12} and p_{01} gives:

$$P(t) = t[tp_2 + (1-t)p_1] + (1-t)[tp_1 + (1-t)p_0]$$

Factoring and combining terms then gives:

$$P(t) = (1-t)^2 p_0 + (-2t^2 + 2t)p_1 + t^2 p_2$$

or,

$$P(t) = \sum_{i=0}^2 p_i B_i(t)$$

where:

$$B_0(t) = (1-t)^2$$

$$B_1(t) = -2t^2 + 2t$$

$$B_2(t) = t^2$$

Thus, we find any point on the curve by a weighted sum of the control points. The weighting function B is often called a “blending function” (although the name “ B ” actually derives from Sergei Bernstein [BE16] who first characterized this family of polynomials). Note that the blending functions are all quadratic in form, which is why the resulting curve is called a quadratic Bézier curve.

11.2 CUBIC BÉZIER CURVES

We now extend our model to *four* control points, resulting in a *cubic* Bézier curve as shown in Figure 11.4. Cubic Bézier curves are capable of defining a much richer set of shapes than are quadratic curves, which are limited to concave shapes.

As for the quadratic case, we can derive an analytic definition for cubic Bézier curves:

$$p_{01}(t) = tp_1 + (1-t)p_0$$

$$p_{12}(t) = tp_2 + (1-t)p_1$$

$$p_{23}(t) = tp_3 + (1-t)p_2$$

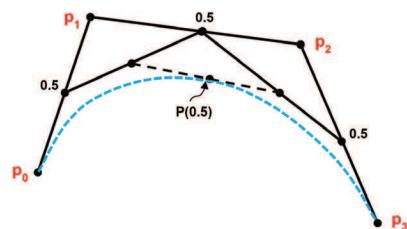


Figure 11.4
Building a cubic Bézier curve.

$$\begin{aligned}\mathbf{p}_{01-12}(t) &= t\mathbf{p}_{12}(t) + (1-t)\mathbf{p}_{01}(t) \\ \mathbf{p}_{12-23}(t) &= t\mathbf{p}_{23}(t) + (1-t)\mathbf{p}_{12}(t)\end{aligned}$$

A point on the curve would then be:

$$P(t) = t\mathbf{p}_{12-23}(t) + (1-t)\mathbf{p}_{01-12}(t)$$

Substituting the definitions of \mathbf{p}_{12-23} and \mathbf{p}_{01-12} and collecting terms yields:

$$P(t) = \sum_{i=0}^3 \mathbf{p}_i B_i(t)$$

where:

$$\begin{aligned}B_0(t) &= (1-t)^3 \\ B_1(t) &= 3t^3 - 6t^2 + 3t \\ B_2(t) &= -3t^3 + 3t^2 \\ B_3(t) &= t^3\end{aligned}$$

There are many different techniques for rendering Bézier curves. One approach is to iterate through successive values of t , starting at 0.0 and ending at 1.0, using a fixed increment. For instance, if the increment is 0.1, then we could use a loop with t values 0.0, 0.1, 0.2, 0.3, and so on. For each value of t , the corresponding point on the Bézier curve would be computed, and a series of line segments connecting the successive points would be drawn, as described in the algorithm in Figure 11.5:

```
void drawBezierCurve (controlPointVector C)
{
    currentPoint = C[0];           // curve starts at first control point
    t = 0.0;
    while (t <= 1.0)
    {
        // compute next point as the sum of the Control Points,
        // weighted by the blending function evaluated at 't'.
        nextPoint = (0,0);
        for (int i=0; i<=3; i++)
            nextPoint = nextPoint + (blending(i,t) * C[i]);
        drawLine (currentPoint,nextPoint);
        currentPoint = nextPoint;
        t = t + increment;
    }
}
```

```

double blending(int i, double t)
{
    switch (i)
    {
        case 0: return ((1-t)*(1-t)*(1-t)); // (1-t)3
        case 1: return (3*t*(1-t)*(1-t)); // 3t(1-t)2
        case 2: return (3*t*t*(1-t)); // 3t2(1-t)
        case 3: return (t*t*t); // t3
    }
}

```

Figure 11.5

Iterative algorithm for rendering Bézier curves.

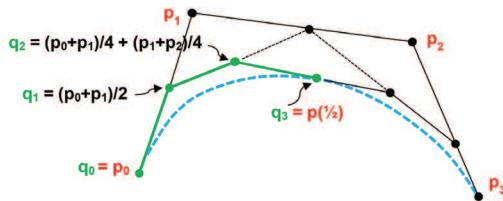
Another approach is to use de Casteljau's algorithm to *recursively subdivide* the curve in half, where $t=1/2$ at each recursive step. Figure 11.6 shows the left side subdivision into new cubic control points (q_0, q_1, q_2, q_3) shown in green, as derived by de Casteljau (a full derivation can be found in [AS14]).

The algorithm is shown in Figure 11.7. It subdivides the curve segments in half repeatedly, until each curve segment is sufficiently straight enough that further subdivision produces no tangible benefit. In the limiting case (as the control points are generated closer and closer together), the curve segment itself is effectively the same as a straight line between the first and last control points (p_0 and p_3). Determining whether a curve segment is “straight enough” can therefore be done by comparing the distance from the first control point to the last control point, versus the sum of the lengths of the three lines connecting the four control points:

$$D_1 = |p_0 - p_1| + |p_1 - p_2| + |p_2 - p_3|$$

$$D_2 = |p_0 - p_3|$$

Then, if $D_1 - D_2$ is less than a sufficiently small tolerance, there is no point in further subdivision.

**Figure 11.6**

Subdividing a cubic Bézier curve.

```

drawBezierCurve(ControlPointVector C)
{  if (C is "straight enough")
   draw line from first to last control point
else
{  subdivide(C, LeftC, RightC)
   drawBezierCurve(LeftC)
   drawBezierCurve(RightC)
}
subdivide(ControlPointVector p, q, r)
{ // compute left subdivision control points
  q(0) = p(0)
  q(1) = (p(0)+p(1)) / 2
  q(2) = (p(0)+p(1)) / 4 + (p(1)+p(2)) / 4
// compute right subdivision control points
  r(1) = (p(1)+p(2)) / 4 + (p(2)+p(3)) / 4
  r(2) = (p(2)+p(3)) / 2
  r(3) = p(3)
// compute "shared" center point at t=0.5
  q(3) = r(0) = (q(2)+r(1)) / 2
}
}

```

Figure 11.7

Recursive subdivision algorithm for Bézier curves.

An interesting property of the de Casteljau algorithm is that it is possible to generate all of the points on the curve without actually using the previously described blending functions. Also, note that the center point at $p(\frac{1}{2})$ is “shared”; that is, it is both the rightmost control point in the left subdivision, and the leftmost control point in the right subdivision. It can be computed either using the blending functions at $t=\frac{1}{2}$, or by using the formula $(q_2 + r_1)/2$, as derived by de Casteljau.

As a side note, we point out that the `subdivide()` function shown in Figure 11.7 assumes that the incoming parameters `p`, `q`, and `r` are “reference” parameters (such as Java objects), and hence the computations in the function modify the actual parameters in the calls from the `drawBezierCurve()` function listed above it.

11.3 QUADRATIC BÉZIER SURFACES

Whereas Bézier curves define curved *lines* (in 2D or 3D space), Bézier *surfaces* define *curved surfaces* in *3D space*. Extending the concepts we saw in

curves to surfaces requires extending our system of parametric equations from one parameter to two parameters. For Bézier curves, we called that parameter t . For Bézier surfaces, we will refer to the parameters as u and v . Whereas our curves were comprised of points $P(t)$, our surfaces will be comprised of points $P(u,v)$, as shown in Figure 11.8.

For quadratic Bézier surfaces, there are three control points on each axis u and v , for a total of *nine* control points. Figure 11.9 shows an example of a set of nine control points (typically called a *control point “mesh”*) in blue, and the associated corresponding curved surface (in red).

The nine control points in the mesh are labeled p_{ij} , where i and j represent the indices in the u and v directions respectively. Each set of three adjacent control points, such as (p_{00}, p_{01}, p_{02}) , defines a Bézier curve. Points $P(u,v)$ on the surface are then defined as a sum of two blending functions, one in the u direction and one in the v direction. The form of the two blending functions for building Bézier surfaces then follows from the methodology given previously for Bézier curves:

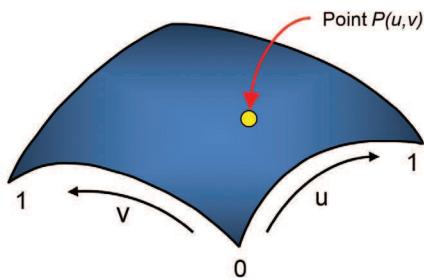


Figure 11.8
Parametric surface.

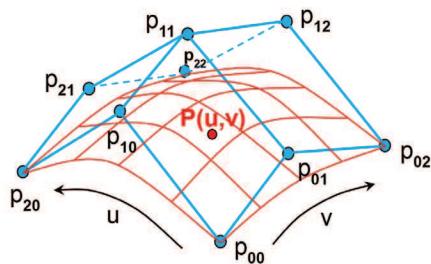


Figure 11.9
Quadratic Bézier control mesh and corresponding surface.

$$B_0(u) = (1-u)^2$$

$$B_1(u) = -2u^2 + 2u$$

$$B_2(u) = u^2$$

$$B_0(v) = (1-v)^2$$

$$B_1(v) = -2v^2 + 2v$$

$$B_2(v) = v^2$$

The points $P(u,v)$ comprising the Bézier surface are then generated by summing the product of each control point p_{ij} and the i^{th} and j^{th} blending functions evaluated at parametric values u and v respectively:

$$P(u,v) = \sum_{i=0}^2 \sum_{j=0}^2 p_{ij} * B_i(u) * B_j(v)$$

The set of generated points that comprise a Bézier surface is sometimes called a *patch*. The term “patch” can sometimes be confusing, as we will see later when we study tessellation shaders (useful for actually implementing Bézier surfaces). There, it is the grid of control points that is typically called a “patch.”

11.4 CUBIC BÉZIER SURFACES

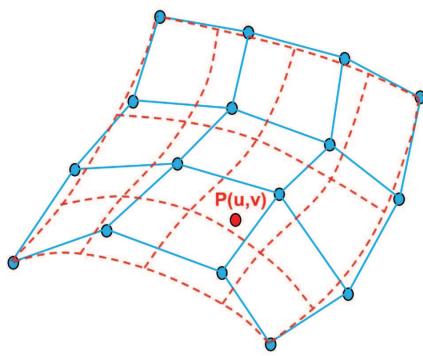


Figure 11.10

Cubic Bézier control mesh and corresponding surface.

Moving from quadratic to cubic surfaces requires utilizing a larger mesh—4x4 rather than 3x3. Figure 11.10 shows an example of a 16-control-point mesh (in blue), and the corresponding curved surface (in red).

As before, we can derive the formula for points $P(u,v)$ on the surface by combining the associated blending functions for cubic Bézier curves:

$$P(u,v) = \sum_{i=0}^3 \sum_{j=0}^3 p_{ij} * B_i(u) * B_j(v)$$

where:

$$B_0(u) = (1-u)^3$$

$$B_0(v) = (1-v)^3$$

$$B_1(u) = 3u^3 - 6u^2 + 3u$$

$$B_1(v) = 3v^3 - 6v^2 + 3v$$

$$B_2(u) = -3u^3 + 3u^2$$

$$B_2(v) = -3v^3 + 3v^2$$

$$B_3(u) = u^3$$

$$B_3(v) = v^3$$

Rendering Bézier surfaces can also be done with recursive subdivision, by alternately splitting the surface in half along each dimension, as shown in Figure 11.11.

Each subdivision produces four new control point meshes, each containing 16 points which define one quadrant of the surface.

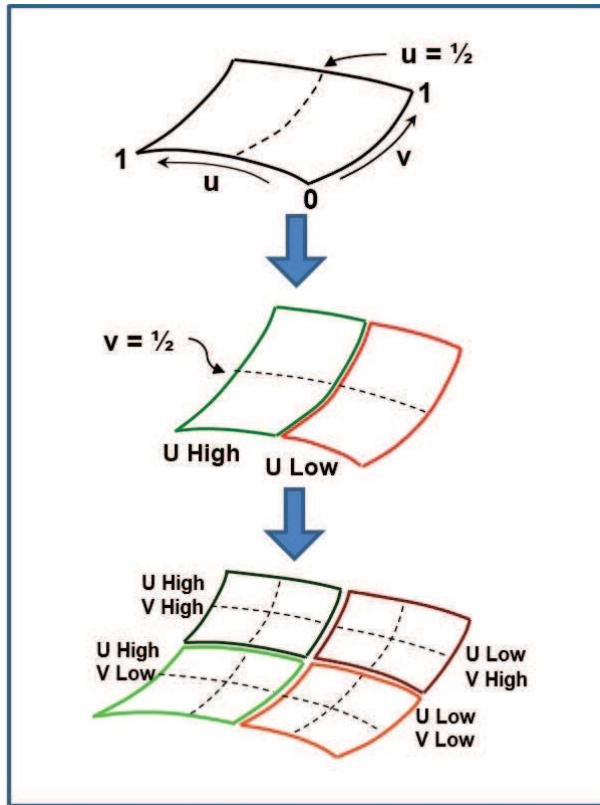


Figure 11.11
Recursive subdivision for Bézier surfaces.

Similar methods exist for rendering Bézier surfaces as we saw previously for Bézier curves, such as by using recursive subdivision [AS14]. When rendering Bézier curves, we stopped subdividing when the *curve* was “straight enough.” For Bézier surfaces, we stop recursing when the *surface* is “flat enough.” One way of doing this is to ensure that all of the recursively generated points in a sub-quadrant control mesh are within some small allowable distance from a plane defined by three of the four corner points of that mesh. The distance d between a point (x,y,z) and a plane (A,B,C,D) is:

$$d = \text{abs} \left(\frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}} \right)$$

If d is less than some sufficiently small tolerance, then we stop subdividing, and simply use the four corner control points of the sub-quadrant mesh to draw two triangles.

The *tessellation* stage of the OpenGL pipeline offers an attractive alternative approach for rendering Bézier surfaces based on the iterative algorithm in Figure 11.5. The strategy is to have the tessellator generate a large grid of vertices, then use the blending functions to reposition those vertices onto the Bézier surface as specified by the cubic Bézier control points. We implement this in Chapter 12.

SUPPLEMENTAL NOTES

This chapter focused on the mathematical fundamentals of parametric Bézier curves and surfaces. We have deferred presenting an implementation of any of them in OpenGL, because an appropriate vehicle for this is the tessellation stage, covered in the next chapter. We also skipped some of the derivations, such as for the recursive subdivision algorithm.

In 3D graphics, there are many advantages to using Bézier curves for modeling objects. First, those objects can, in theory, be scaled arbitrarily and still retain smooth surfaces without “pixelating.” Second, many objects made up of complex curves can be stored much more efficiently as sets of Bézier control points, rather than using thousands of vertices.

Bézier curves have many real-world applications besides computer graphics and automobiles. They can also be found in the design of bridges, such as in the Chords Bridge in Jerusalem [CB16]. Similar techniques are used for building TrueType fonts, which as a result can be scaled to any arbitrary size, or zoomed in to any degree of closeness, while always retaining smooth edges.

Exercises

- 11.1** Quadratic Bézier curves are limited to defining curves that are wholly “concave” or “convex.” Describe (or draw) an example of a curve that bends in a manner that is neither wholly concave nor convex, and thus could not possibly be approximated by a *quadratic* Bézier curve.

- 11.2** Using a pen or pencil, draw an arbitrary set of four points on a piece of paper, number them from 1 to 4 in any order, and then try to draw an approximation of the cubic Bézier curve defined by those four ordered control points. Then rearrange the numbering of the control points (i.e., their order, but without changing their positions) and redraw the new resulting cubic Bézier curve. There are numerous online tools for drawing Bézier curves you can use to check your approximation.

References

- [AS14] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with WebGL*, 7th ed. (Pearson, 2014).
- [CB16] Chords Bridge, Wikipedia, accessed July 2016, https://en.wikipedia.org/wiki/Chords_Bridge.
- [BE16] S. Bernstein, Wikipedia, accessed July 2016, https://en.wikipedia.org/wiki/Sergei_Natanovich_Bernstein.
- [BE72] P. Bézier, *Numerical Control: Mathematics and Applications* (London: John Wiley & Sons, 1972).
- [DC63] P. de Casteljau, *Courbes et surfaces à pôles*, technical report (Paris: A. Citroën, 1963).

TESSELLATION

12.1	<i>Tessellation in OpenGL</i>	261
12.2	<i>Tessellation for Bézier Surfaces</i>	267
12.3	<i>Tessellation for Terrain / Height Maps</i>	273
12.4	<i>Controlling Level of Detail (LOD)</i>	281
	<i>Supplemental Notes</i>	284



The English language term “tessellation” refers to a large class of design activities in which tiles of various geometric shapes are arranged adjacently to form patterns, generally on a flat surface. The purpose can be artistic or practical, with examples dating back thousands of years [TS16].

In 3D graphics, *tessellation* refers to something a little bit different, but no doubt inspired by its classical counterpart. Here, tessellation refers to the generation and manipulation of large numbers of triangles for rendering complex shapes and surfaces, preferably in hardware. Tessellation is a rather recent addition to the OpenGL core, not appearing until 2010 with version 4.0.¹

12.1 TESSELLATION IN OPENGL

OpenGL support for hardware tessellation is made available through three pipeline stages:

1. the *tessellation control shader*
2. the *tessellator*
3. the *tessellation evaluation shader*

¹ Although the GLU toolset previously included a utility for tessellation much earlier called *gluTess*. In 2001, Radeon released the first commercial graphics card with tessellation support, but there were few tools able to take advantage of it.

The first and third stages are programmable; the intervening second stage is not. In order to use tessellation, the programmer generally provides both a control shader and an evaluation shader.

The tessellator (its full name is *tessellation primitive generator*, or *TPG*) is a hardware-supported engine that produces fixed grids of triangles.² The control shader allows us to configure what sort of triangle mesh the tessellator is to build. The evaluation shader then lets us manipulate the grid in various ways. The manipulated triangle mesh is then the source of vertices that proceed through the pipeline. Recall from Figure 2.2 that tessellation sits in the pipeline between the vertex and geometry shader stages.

Let's start with an application that simply uses the tessellator to create a triangle mesh of vertices, and then displays it without any manipulation. For this, we will need the following modules:

1. ***Java/JOGL application:***

Creates a camera and associated mvp matrix. The view (v) and projection (p) matrices orient the camera; the model (m) matrix can be used to modify the location and orientation of the grid.

2. ***Vertex shader:***

Essentially does nothing in this example; the vertices will be generated in the tessellator.

3. ***Tessellation Control Shader (TCS):***

Specifies the grid for the tessellator to build.

4. ***Tessellation Evaluation Shader (TES):***

Applies the mvp matrix to the vertices in the grid.

5. ***Fragment Shader:***

Simply outputs a fixed color for every pixel.

We first list the entire application code, and then discuss its elements in detail. Even a simple example such as this one is fairly complex, so many of the code elements will require explanation.

² or lines, but we will focus on triangles.

Program 12.1 Basic Tessellator Mesh

Java / JOGL application

```
private int createShaderProgram()
{
    // same as before, but now includes TCS and TES

    ...
    int[] tescCompiled = new int[1];
    int[] teseCompiled = new int[1];

    tcShaderSource = readShaderSource("tessC.shader");
    teShaderSource = readShaderSource("tessE.shader");

    int tessCShader = gl.glCreateShader(GL_TESS_CONTROL_SHADER);
    int tessEShader = gl.glCreateShader(GL_TESS_EVALUATION_SHADER);

    gl.glShaderSource(tessCShader, tcShaderSource.length, tcShaderSource, null, 0);
    gl.glShaderSource(tessEShader, teShaderSource.length, teShaderSource, null, 0);

    gl.glCompileShader(tessCShader);
    gl.glCompileShader(tessEShader);

    ...
    gl.glAttachShader(rendering_program, tessCShader);
    gl.glAttachShader(rendering_program, tessEShader);
    ...

}

public void display(GLAutoDrawable drawable)
{
    ...
    gl.glUseProgram(rendering_program);

    ...
    // set up mvp matrix for camera

    ...
    // parameters for drawing like before; draw as follows:
    gl.glPatchParameteri(GL_PATCH_VERTICES, 1);
    gl.glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    gl.glDrawArrays(GL_PATCHES, 0, 1);
}
```

Vertex Shader

```
#version 430
uniform mat4 mvp;
void main(void) { }
```

Tessellation Control Shader

```
#version 430
uniform mat4 mvp;
```

```

layout (vertices = 1) out;

void main(void)
{
    gl_TessLevelOuter[0] = 6;
    gl_TessLevelOuter[1] = 6;
    gl_TessLevelOuter[2] = 6;
    gl_TessLevelOuter[3] = 6;
    gl_TessLevelInner[0] = 12;
    gl_TessLevelInner[1] = 12;
}

```

Tessellation Evaluation Shader

```

#version 430
uniform mat4 mvp;

layout (quads, equal_spacing, ccw) in;

void main (void)
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    gl_Position = mvp * vec4(u,0,v,1);
}

```

Fragment Shader

```

#version 430
out vec4 color;
uniform mat4 mvp;

void main(void)
{
    color = vec4(1.0, 1.0, 0.0, 1.0); // yellow
}

```

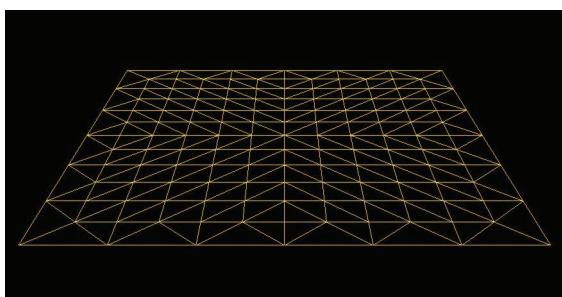


Figure 12.1
Tessellator triangle mesh output.

The resulting output mesh is shown in Figure 12.1.

As can be seen in Figure 12.1, the tessellator produces a mesh of vertices defined by two parameters: *inner level* and *outer level*. In this case, the inner level is 12 and the outer level is 6—the outer edges of the grid are divided into 6 segments, while the lines spanning the interior are divided into 12 segments.

The specific relevant new constructs in Program 12.1 are highlighted. Let's start by discussing the first portion—the Java/JOGL code.

Compiling the two new shaders is done exactly the same as for the vertex and fragment shaders. They are then attached to the same rendering program, and the linking call is unchanged. The only new items are the constants for specifying the type of shader being instantiated—the new constants are named:

```
GL_TESS_CONTROL_SHADER
GL_TESS_EVALUATION_SHADER
```

Note the new items in the `display()` function. The `glDrawArrays()` call now specifies `GL_PATCHES`. When using tessellation, vertices sent from the Java/JOGL application into the pipeline (i.e., in a VBO) *aren't rendered*, but are usually *control points*, such as those we saw for Bézier curves. A set of control points is called a "patch," and in those sections of the code using tessellation, `GL_PATCHES` is the only allowable primitive. The number of vertices in a patch is specified in the call to `glPatchParameteri()`. In this particular example, there aren't any control points being sent, but we are still required to specify at least 1. Similarly, in the `glDrawArrays()` call we indicate a start value of 0 and a vertex count of 1, even though we aren't actually sending any vertices from the JOGL program.

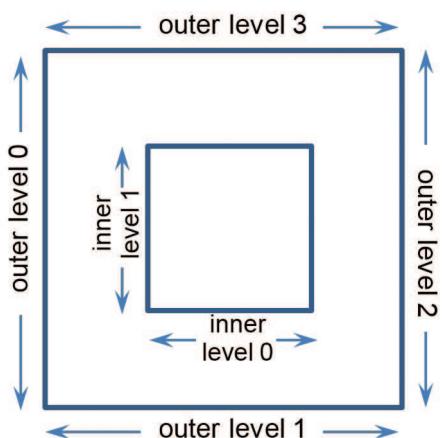
The call to `glPolygonMode()` specifies how the mesh should be rasterized. The default is `GL_FILL`. Shown in the code is `GL_LINE`, which as we saw in Figure 12.1 caused only connecting lines to be rasterized (so we could see the grid itself that was produced by the tessellator). If we change that line of code to `GL_FILL` (or comment it out, resulting in the default behavior `GL_FILL`), we get the version shown in Figure 12.2.

Now let's work our way through the four shaders. As indicated earlier, the vertex shader has little to do, since the Java/JOGL application isn't providing any vertices. All it contains is a uniform declaration, to match the other shaders. In any case, it is a requirement that all shader programs include a vertex shader.

The Tessellation Control Shader specifies the topology of the triangle



Figure 12.2
Tessellated mesh rendered with `GL_FILL`.

**Figure 12.3**

Tessellation levels.

mesh that the tessellator is to produce. Six “level” parameters are set—two “inner” and four “outer” levels—by assigning values to the reserved words named `gl_TessLevelxxx`. This is for tessellating a large *rectangular* grid of triangles, called a *quad*.³ The levels tell the tessellator how to subdivide the grid when forming triangles, and are arranged as shown in Figure 12.3.

Note the line in the control shader that says:

```
layout (vertices=1) out;
```

This is related to the prior `GL_PATCHES` discussion and specifies the number of vertices per “patch” being passed from the vertex shader to the control shader (and “out” to the evalua-

tion shader). In this particular program there are none, but we still must specify at least one, because it also affects how many times the control shader executes. Later this value will reflect the number of control points, and must match the value in the `glPatchParameteri()` call in the Java/JOGL application.

Next let’s look at the Tessellation Evaluation Shader. It starts with a line of code that says:

```
layout (quads, equal_spacing, ccw) in;
```

This may at first appear to be related to the “out” layout statement in the control shader, but actually they are unrelated. Rather, this line is where we instruct the tessellator to generate vertices so they are arranged in a large rectangle (a “quad”). It also specifies the subdivisions (inner and outer) to be of equal length (later we will see a use for subdivisions of unequal length). The “ccw” parameter specifies the winding order in which the tessellated grid vertices are generated (in this case, counter-clockwise).

The vertices generated by the tessellator are then sent to the evaluation shader. Thus, the evaluation shader *may* receive vertices *both* from the control

³ The tessellator is also capable of building a *triangular* grid of triangles, but that isn’t covered in this textbook.

shader (typically as control points), and from the tessellator (the tessellated grid). In Program 12.1, vertices are only received from the tessellator.

The evaluation shader executes once for each vertex produced by the tessellator. The vertex location is accessible using the built-in variable `gl_TessCoord`. The tessellated grid is oriented such that it lies in the X-Z plane, and therefore `gl_TessCoord`'s X and Y components are applied at the grid's X and Z coordinates. The grid coordinates, and thus the values of `gl_TessCoord`, range from 0.0 to 1.0 (this will be handy later when computing texture coordinates). The evaluation shader then uses the mvp matrix to orient each vertex (this was done in the vertex shader in examples from earlier chapters).

Finally, the fragment shader simply outputs a constant color yellow for each pixel. We can, of course, also use it to apply a texture or lighting to our scene as we saw in previous chapters.

12.2 TESSELLATION FOR BÉZIER SURFACES

Let's now extend our program so that it turns our simple rectangular grid into a Bézier surface. The tessellated grid should give us plenty of vertices for sampling the surface (and we can increase the inner/outer subdivision levels if we want more). What we now need is to send *control points* through the pipeline, and then use those control points to perform the computations to convert the tessellated grid into the desired Bézier surface.

Assuming that we wish to build a *cubic* Bézier surface, we will need 16 control points. We could send them from the Java side in a VBO, or we could hardcode them in the vertex shader. Figure 12.4 shows an overview of the process, with the control points coming from the Java side.

Now is a good time to explain a bit more precisely how the tessellation control shader (TCS) works. Similar to the vertex shader, the TCS executes *once per incoming vertex*. Also, recall from Chapter 4 that OpenGL provides a built-in variable called `gl_VertexID` which holds a counter that indicates which invocation of the vertex shader is

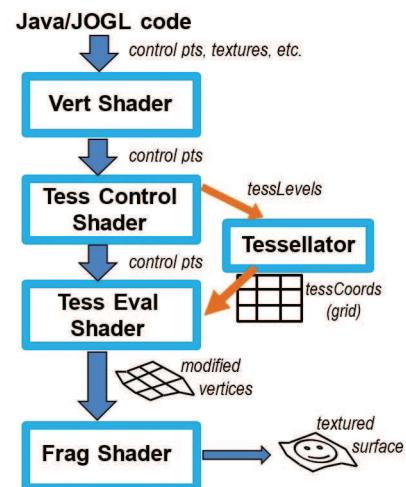


Figure 12.4
Overview of tessellation for Bézier surfaces.

currently executing. A similar built-in variable called `gl_InvocationID` exists for the tessellation control shader.

A powerful feature of tessellation is that the TCS (and also the TES) shader has access to *all* of the control point vertices simultaneously, in arrays. At first, it may seem confusing that the TCS executes once per vertex, when each invocation has access to all of the vertices. It is also counterintuitive that the tessellation levels are specified in assignment statements which are redundantly set at each TCS invocation. Although all of this may seem odd, it is done this way because the tessellation architecture is designed so that TCS invocations can run in parallel.

OpenGL provides several built-in variables for use in the TCS and TES shaders. Ones that we have already mentioned are `gl_InvocationID`, and of course `gl_TessLevelInner` and `gl_TessLevelOuter`. Here are some more details and descriptions of some of the most useful built-in variables:

Tessellation Control Shader (TCS) built-in variables:

- `gl_in[]` – an array containing each of the incoming control point vertices—one array element per incoming vertex. Particular vertex attributes can be accessed as fields using the “.” notation. One built-in attribute is `gl_Position`—thus, the position of incoming vertex “*i*” is accessed as `gl_in[i].gl_Position`.
- `gl_out[]` – an array for sending outgoing control point vertices to the TES—one array element per outgoing vertex. Particular vertex attributes can be accessed as fields using the “.” notation. One built-in attribute is `gl_Position`—thus, the position of outgoing vertex “*i*” is accessed as `gl_out[i].gl_Position`.
- `gl_InvocationID` – an integer ID counter indicating which invocation of the TCS is currently executing. One common use is for passing through vertex attributes; for example, passing the current invocation’s vertex position from the TCS to the TES would be done as follows:
`gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;`

Tessellation Evaluation Shader (TES) built-in variables:

- `gl_in[]` – an array containing each of the incoming control point vertices—one element per incoming vertex. Particular vertex attributes can be accessed as fields using the “.” notation. One built-in attribute is `gl_Position`—thus, incoming vertex positions are accessed as `gl_in[xxx].gl_Position`.

- `gl_Position` – output position of a tessellated grid vertex, possibly modified in the TES. It is important to note that `gl_Position` and `gl_in[xxx].gl_Position` are different—`gl_Position` is the position of an output vertex that originated in the tessellator, while `gl_in[xxx].gl_Position` is a control point vertex position coming into the TES from the TCS.

It is important to note that input and output control point vertex attributes in the TCS are arrays. By contrast, input control point vertices and vertex attributes in the TES are arrays, but output vertices are scalars. Also, it is easy to become confused as to which vertices are for control points, and which are tessellated and then moved to form the resulting surface. To summarize, all vertex inputs and outputs to the TCS are control points, whereas in the TES, `gl_in[]` holds incoming control points, `gl_TessCoord` holds incoming tessellated grid points, and `gl_Position` holds output surface vertices for rendering.

Our tessellation control shader now has two tasks: specifying the tessellation levels, *and* passing the control points through from the vertex shader to the evaluation shader. The evaluation shader can then modify the locations of the grid points (the `gl_TessCoords`) based on the Bézier control points.

Program 12.2 shows all four shaders—vertex, TCS, TES, and fragment—for specifying a control point patch, generating a flat tessellated grid of vertices, re-positioning those vertices on the curved surface specified by the control points, and painting the resulting surface with a texture image. It also shows the relevant portion of the Java/JOGL application, specifically in the `display()` function. In this example, the control points originate in the vertex shader (they are hardcoded there), rather than entering the OpenGL pipeline from the Java/JOGL application. Additional code details follow after the code listing.

Program 12.2 Tessellation for Bézier Surface

Vertex Shader

```
#version 430
out vec2 texCoord;
uniform mat4 mvp;
layout (binding = 0) uniform sampler2D tex_color;

void main(void)
{ // this time the vertex shader defines and sends out control points:
```

```

const vec4 vertices[ ] =
vec4[ ] (vec4(-1.0, 0.5, -1.0, 1.0),vec4(-0.5, 0.5, -1.0, 1.0),
          vec4( 0.5, 0.5, -1.0, 1.0), vec4( 1.0, 0.5, -1.0, 1.0),
          vec4(-1.0, 0.0, -0.5, 1.0), vec4(-0.5, 0.0, -0.5, 1.0),
          vec4( 0.5, 0.0, -0.5, 1.0), vec4( 1.0, 0.0, -0.5, 1.0),
          vec4(-1.0, 0.0, 0.5, 1.0), vec4(-0.5, 0.0, 0.5, 1.0),
          vec4( 0.5, 0.0, 0.5, 1.0), vec4( 1.0, 0.0, 0.5, 1.0),
          vec4(-1.0, -0.5, 1.0, 1.0), vec4(-0.5, 0.3, 1.0, 1.0),
          vec4( 0.5, 0.3, 1.0, 1.0), vec4( 1.0, 0.3, 1.0, 1.0));

// compute an appropriate texture coordinate for the current vertex, shifted from (-1...+1) to (0...1)
texCoord = vec2((vertices[gl_VertexID].x + 1.0) / 2.0, (vertices[gl_VertexID].z + 1.0) / 2.0);

gl_Position = vertices[gl_VertexID];
}

```

Tessellation Control Shader

```
#version 430
```

```

in vec2 texCoord[ ]; // The texture coords output from the vertex shader as scalars arrive in an array
out vec2 texCoord_TCSout[ ]; // and are then passed through to the evaluation shader

uniform mat4 mvp;
layout (binding = 0) uniform sampler2D tex_color;
layout (vertices = 16) out; // there are 16 control points per patch

void main(void)
{
    int TL = 32; // tessellation levels are all set to this value
    if (gl_InvocationID == 0)
    {
        gl_TessLevelOuter[0] = TL; gl_TessLevelOuter[2] = TL;
        gl_TessLevelOuter[1] = TL; gl_TessLevelOuter[3] = TL;
        gl_TessLevelInner[0] = TL; gl_TessLevelInner[1] = TL;
    }
    // forward the texture and control points to the TES
    texCoord_TCSout[gl_InvocationID] = texCoord[gl_InvocationID];
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}

```

Tessellation Evaluation Shader

```

#version 430
layout (quads, equal_spacing, ccw) in;
uniform mat4 mvp;
layout (binding = 0) uniform sampler2D tex_color;
```

```

in vec2 texCoord_TCSout[];           // texture coordinate array coming in
out vec2 texCoord_TESout;           // scalars going out one at a time

void main (void)
{
    vec3 p00 = (gl_in[0].gl_Position).xyz;
    vec3 p10 = (gl_in[1].gl_Position).xyz;
    vec3 p20 = (gl_in[2].gl_Position).xyz;
    vec3 p30 = (gl_in[3].gl_Position).xyz;
    vec3 p01 = (gl_in[4].gl_Position).xyz;
    vec3 p11 = (gl_in[5].gl_Position).xyz;
    vec3 p21 = (gl_in[6].gl_Position).xyz;
    vec3 p31 = (gl_in[7].gl_Position).xyz;
    vec3 p02 = (gl_in[8].gl_Position).xyz;
    vec3 p12 = (gl_in[9].gl_Position).xyz;
    vec3 p22 = (gl_in[10].gl_Position).xyz;
    vec3 p32 = (gl_in[11].gl_Position).xyz;
    vec3 p03 = (gl_in[12].gl_Position).xyz;
    vec3 p13 = (gl_in[13].gl_Position).xyz;
    vec3 p23 = (gl_in[14].gl_Position).xyz;
    vec3 p33 = (gl_in[15].gl_Position).xyz;

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    // cubic Bezier basis functions
    float bu0 = (1.0-u) * (1.0-u) * (1.0-u);           // (1-u)^3
    float bu1 = 3.0 * u * (1.0-u) * (1.0-u);           // 3u(1-u)^2
    float bu2 = 3.0 * u * u * (1.0-u);                  // 3u^2(1-u)
    float bu3 = u * u * u;                                // u^3
    float bv0 = (1.0-v) * (1.0-v) * (1.0-v);           // (1-v)^3
    float bv1 = 3.0 * v * (1.0-v) * (1.0-v);           // 3v(1-v)^2
    float bv2 = 3.0 * v * v * (1.0-v);                  // 3v^2(1-v)
    float bv3 = v * v * v;                                // v^3

    // output the position of this vertex in the tessellated patch
    vec3 outputPosition =
        bu0 * ( bv0*p00 + bv1*p01 + bv2*p02 + bv3*p03 )
        + bu1 * ( bv0*p10 + bv1*p11 + bv2*p12 + bv3*p13 )
        + bu2 * ( bv0*p20 + bv1*p21 + bv2*p22 + bv3*p23 )
        + bu3 * ( bv0*p30 + bv1*p31 + bv2*p32 + bv3*p33 );
    gl_Position = mvp * vec4(outputPosition,1.0f);

    // output the interpolated texture coordinates
    vec2 tc1 = mix(texCoord_TCSout[0], texCoord_TCSout[3], gl_TessCoord.x);
    vec2 tc2 = mix(texCoord_TCSout[12], texCoord_TCSout[15], gl_TessCoord.x);
    vec2 tc = mix(tc2, tc1, gl_TessCoord.y);
}

```

```

texCoord_TESout = tc;
}

```

Fragment Shader

```

#version 430
in vec2 texCoord_TESout;
out vec4 color;
uniform mat4 mvp;
layout (binding = 0) uniform sampler2D tex_color;

void main(void)
{
    color = texture(tex_color, texCoord_TESout);
}

```

Java / JOGL application

```

// This time we also pass a texture to paint the surface.
// Load the texture in init() as usual, then enable it in display()

public void display(GLAutoDrawable drawable)
{
    ...
    gl.glActiveTexture(GL_TEXTURE0);
    gl glBindTexture(GL_TEXTURE_2D, textureID);

    gl.glFrontFace(GL_CCW);

    gl.glPatchParameteri(GL_PATCH_VERTICES, 16); // number of vertices per patch = 16
    gl.glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    gl.glDrawArrays(GL_PATCHES, 0, 16); // total number of patch vertices: 16 x 1 patch = 16
}

```

The vertex shader now specifies 16 control points (the “patch” vertices) representing a particular Bézier surface. In this example they are all normalized to the range [-1...+1]. The vertex shader also uses the control points to determine texture coordinates appropriate for the tessellated grid, with values in the range [0...1]. It is important to reiterate that the vertices output from the vertex shader are **not** vertices that will be rasterized, but instead are Bézier control points. When using tessellation, patch vertices are **never** rasterized—only tessellated vertices proceed to rasterization.

The control shader still specifies the inner and outer tessellation levels. It now has the additional responsibility of forwarding the control points and texture

coordinates to the evaluation shader. Note that the tessellation levels only need to be specified once, and therefore that step is done only during the 0th invocation (recall that the TCS runs once per vertex—thus there are 16 invocations in this example). For convenience, we have specified 32 subdivisions for each tessellation level.

Next, the evaluation shader performs all of the Bézier surface computations. The large block of assignment statements at the beginning of `main()` extract the control points from the incoming `gl_Position`'s of each incoming `gl_in` (note that these correspond to the control shader's `gl_out` variable). The weights for the blending functions are then computed using the grid points coming in from the tessellator, resulting in a new `outputPosition`, to which the model-view-projection matrix is then applied, producing an output `gl_Position` for each grid point, forming the Bézier surface.

It is also necessary to create texture coordinates. The vertex shader only provided one for each control point location. But it isn't the control points that are being rendered—we ultimately need texture coordinates for the much larger number of tessellated grid points. There are various ways of doing this—here we linearly interpolate them using GLSL's handy `mix()` function. The `mix()` function expects three parameters: (a) starting point, (b) ending point, and (c) interpolation value, which ranges from 0 to 1. It returns the value between the starting and ending point corresponding to the interpolation value. Since the tessellated grid coordinates also range from 0 to 1, they can be used directly for this purpose.

This time in the fragment shader, rather than outputting a single color, standard texturing is applied. The texture coordinates, in the attribute `texCoord_TESout`, are those that were produced in the evaluation shader. The changes to the JOGL program are similarly straightforward—note that a patch size of 16 is now specified. The resulting output is shown in Figure 12.5 (a tile texture is applied [LU16]).

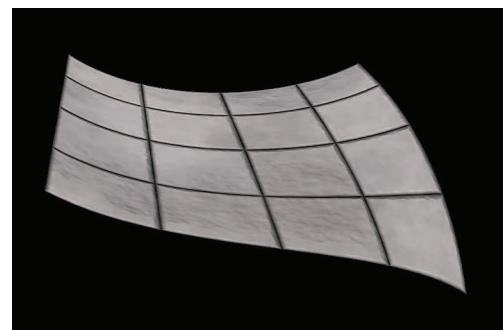


Figure 12.5
Tessellated Bézier surface.

12.3 TESSELLATION FOR TERRAIN / HEIGHT MAPS

Recall that performing height mapping in the vertex shader can suffer from an insufficient number of vertices to render the desired detail. Now that we have a

way to generate lots of vertices, let's go back to Hastings-Trew's moon surface texture map (from [HT16]), and use it as a *height map* by raising *tessellated* vertices to produce moon surface detail. As we will see, this has the advantages of achieving vertex geometry that better matches the moon image, along with improved silhouette (edge) detail.

Our strategy is to modify Program 12.1, placing a tessellated grid in the X-Z plane, and using height mapping to set the Y coordinate of each tessellated grid point. To do this, a patch isn't needed, because we can hardcode the location of the tessellated grid, so we will specify the required minimum of 1 vertex per patch in `glDrawArrays()` and `glPatchParameteri()` as was done in Program 12.1. Hastings-Trew's moon texture image is used both for color and as the height map.

We generate vertex and texture coordinates in the evaluation shader by mapping the tessellated grid's `gl_TessCoord` values to appropriate ranges for vertices and textures.⁴ The evaluation shader also is where the height-mapping is performed, by adding a fraction of the color component of the moon texture to the Y component of output vertex. The changes to the shaders are shown in Program 12.3.

Program 12.3 Simple Tessellated Terrain

Vertex Shader

```
#version 430
uniform mat4 mvp;
layout (binding = 0) uniform sampler2D tex_color;
void main(void) {}
```

Tessellation Control Shader

```
...
layout (vertices = 1) out;      // no control points are necessary for this application
void main(void)
{  int TL=32;
  if (gl_InvocationID == 0)
```

⁴ In some applications the texture coordinates are produced externally, such as when tessellation is being used to provide additional vertices for an imported model. In such cases, the provided texture coordinates would need to be interpolated.

```

    { gl_TessLevelOuter[0] = TL; gl_TessLevelOuter[2] = TL;
      gl_TessLevelOuter[1] = TL; gl_TessLevelOuter[3] = TL;
      gl_TessLevelInner[0] = TL; gl_TessLevelInner[1] = TL;
    }
}

```

Tessellation Evaluation Shader

```

...
out vec2 tes_out;
uniform mat4 mvp;
layout (binding = 0) uniform sampler2D tex_color;

void main (void)
{ // map the tessellated grid vertices from (0..1) onto the desired vertices (-0.5...+0.5)
  vec4 tessellatedPoint = vec4(gl_TessCoord.x - 0.5, 0.0, gl_TessCoord.y - 0.5, 1.0);

  // map the tessellated grid vertices as texture coordinates by "flipping" the Y values vertically.
  // Vertex coords have (0,0) at upper left, texture coords have (0,0) at lower left.
  vec2 tc = vec2(gl_TessCoord.x, 1.0 - gl_TessCoord.y);

  // The image is grayscale, so either component (R,G, or B) can serve as height offset.
  tessellatedPoint.y += (texture(tex_color, tc).r) / 40.0; // scaled down color values.

  // convert the height-map raised point to eye space
  gl_Position = mvp * tessellatedPoint;
  tes_out = tc;
}

```

Fragment Shader

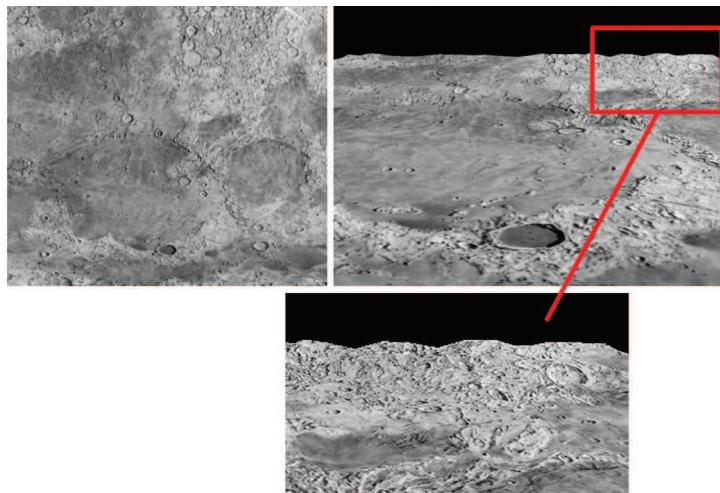
```

...
in vec2 tes_out;
out vec4 color;
layout (binding = 0) uniform sampler2D tex_color;

void main(void)
{ color = texture(tex_color, tes_out);
}

```

The fragment shader is the same as the one for Program 12.2, and simply outputs the color based on the texture image. The Java/JOGL application is essentially unchanged—it loads the texture (serving as both the texture and height

**Figure 12.6**

Tessellated terrain – failed first attempt, with insufficient number of vertices.

map) and enables a sampler for it. Figure 12.6 shows the texture image (on the left) and the final output of this first attempt, which unfortunately does not yet achieve proper height mapping.

The first results are severely flawed. Although we can now see silhouette detail on the far horizon, the bumps there don't correspond to the actual detail in the texture map. Recall that in a height map, white is supposed to mean "high," and black is supposed to mean "low." The area at the upper right, in particular, shows large hills that bear no relation to the light and dark colors in the image.

The cause of this problem is the resolution of the tessellated grid. The maximum number of vertices that can be generated by the tessellator is hardware dependent, and a maximum value of at least 64 for each tessellation level is all that is required for compliance with the OpenGL standard. Our program specified a single tessellated grid with inner and outer tessellation levels of 32, so we generated about 32×32 , or just over 1000 vertices, which is insufficient to reflect the detail in the image accurately. This is especially apparent along the upper right (enlarged in the figure)—the edge detail is only sampled at 32 points along the horizon, producing large, random-looking hills. Even if we increased the tessellation values to 64, the total of 64×64 or just over 4000 vertices would still be woefully inadequate to do height-mapping using the moon image.

One way to increase the number of vertices is by using *instancing*, which we saw in Chapter 4. Our strategy will be to have the tessellator generate grids, and use instancing to repeat this many times. In the vertex shader we build a patch defined by four vertices, one for each corner of a tessellated grid. In our Java/JOGL application we change the `glDrawArrays()` call to `glDrawArraysInstanced()`. There, we specify a grid of 64 by 64 patches, each of which contains a tessellated mesh with levels of size 32. This will give us a total of $64 \times 64 \times 32 \times 32$, or over 4 million vertices.

The vertex shader starts by specifying four texture coordinates $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. When using instancing, recall that the vertex shader has access to an integer variable `gl_InstanceID`, which holds a counter corresponding to the `glDrawArraysInstanced()` call that is currently being processed. We use this ID value to distribute the locations of the individual patches within the larger grid. The patches are positioned in rows and columns, the first patch at location $(0,0)$, the next at $(1,0)$, the next at $(2,0)$, and so on, and the final patch in the first column at $(63,0)$. The next column has patches at $(0,1)$, $(1,1)$, and so forth up to $(63,1)$. The final column has patches at $(0,63)$, $(1,63)$, and so on up to $(63,63)$. The X coordinate for a given patch is the instance ID modulo 64, and the Y coordinate is the instance ID divided by 64 (with integer division). The shader then scales the coordinates back down to the range $(0..1)$.

The control shader is unchanged, except that it passes through the vertices and texture coordinates.

Next, the evaluation shader takes the incoming tessellated grid vertices (specified by `gl_TessCoord`) and moves them into the coordinate range specified by the incoming patch. It does the same for the texture coordinates. It also applies height mapping in the same way as was done in Program 12.3. The fragment shader is unchanged.

The changes to each of the components are shown in Program 12.4. The result is shown in Figure 12.7. Note that the highs and lows now correspond much more closely to light and dark sections of the image.

Program 12.4 Instanced Tessellated Terrain

Java / JOGL application

```
// same as for Bezier surface example, with these changes:  
gl.glPatchParameteri(GL_PATCH_VERTICES, 4);  
gl.glDrawArraysInstanced(GL_PATCHES, 0, 4, 64*64);
```

Vertex Shader

```
...
out vec2 tc;
...
void main(void)
{
    vec2 patchTexCoords[] = vec2[ ] (vec2(0,0), vec2(1,0), vec2(0,1), vec2(1,1));
    // compute an offset for coordinates based on which instance this is
    int x = gl_InstanceID % 64;
    int y = gl_InstanceID / 64;
    // tex coords are distributed across 64 patches, normalized to (0..1). Flip Y coords.
    tc = vec2( (x+patchTexCoords[gl_VertexID].x) / 64.0, (64 - y+patchTexCoords[gl_VertexID].y) / 64.0);
    // vertex locations are the same as texture coordinates, except they range from -0.5 to +0.5.
    gl_Position = vec4(tc.x - 0.5, 0.0, (1.0 - tc.y) - 0.5, 1.0); // Also un-flip the Y's.
}
```

Tessellation Control Shader

```
...
layout (vertices = 4) out;
in vec2 tc[];
out vec2 tcs_out[];
...
void main(void)
{
    // tessellation level specification the same as the previous example
    ...
    tcs_out[gl_InvocationID] = tc[gl_InvocationID];
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

Tessellation Evaluation Shader

```
...
in vec2 tcs_out[];
out vec2 tes_out;
void main (void)
{
    // map the texture coordinates onto the sub-grid specified by the incoming control points
    vec2 tc = vec2(tcs_out[0].x + (gl_TessCoord.x) / 64.0, tcs_out[0].y + (1.0 - gl_TessCoord.y) / 64.0);
```

```

// map the tessellated grid onto the sub-grid specified by the incoming control points
vec4 tessellatedPoint = vec4(gl_in[0].gl_Position.x + gl_TessCoord.x / 64.0, 0.0,
                            gl_in[0].gl_Position.z + gl_TessCoord.y / 64.0, 1.0);

// add the height from the height map to the vertex:
tessellatedPoint.y += (texture(tex_height, tc).r) / 40.0;

gl_Position = mvp * tessellatedPoint;
tes_out = tc;
}

```

Now that we have achieved height mapping, we can work on improving it and incorporating lighting. One challenge is that our vertices do not yet have normal vectors associated with them. Another challenge is that simply using the texture image as a height map has produced an overly “jagged” result—in this case because not all grayscale variation in the texture image is due to height. For this particular texture map, it so happens that Hastings-Trew has already produced an improved height map that we can use [HT16]. It is shown in Figure 12.8 (on the left).

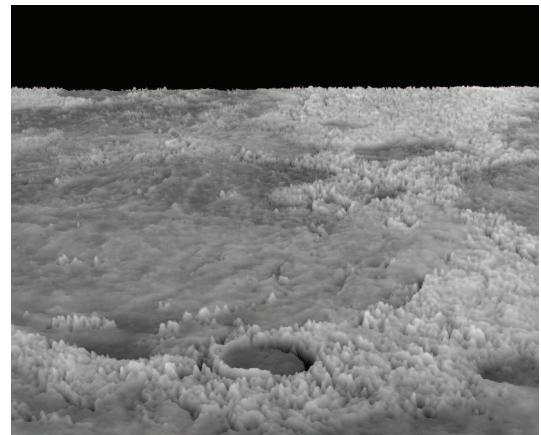


Figure 12.7
Tessellated terrain – second attempt, with instancing.

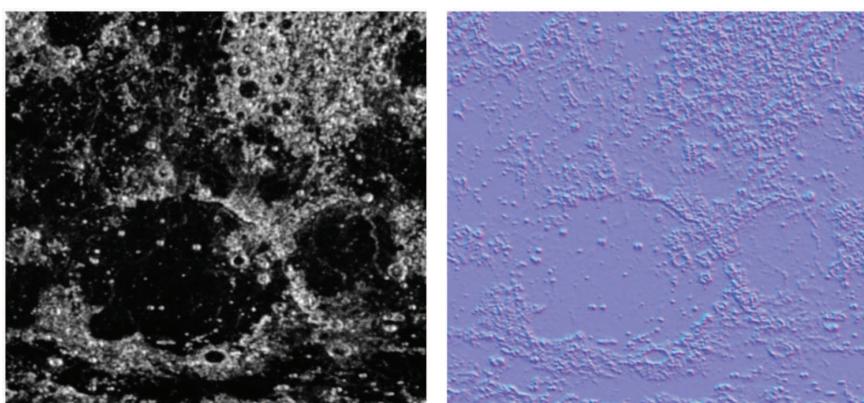


Figure 12.8
Moon surface: height map [HT16] and normal map.

To create normals, we could compute them on the fly, by generating the heights of neighboring vertices (or neighboring texels in the height map), building vectors connecting them, and using a cross product to find the normal. This requires some tuning, depending on the precision of the scene (and/or the height map image). Here we have instead used the GIMP “normalmap” plugin [GP16] to generate a normal map based on Hastings-Trew’s height map, shown in Figure 12.8 (on the right).

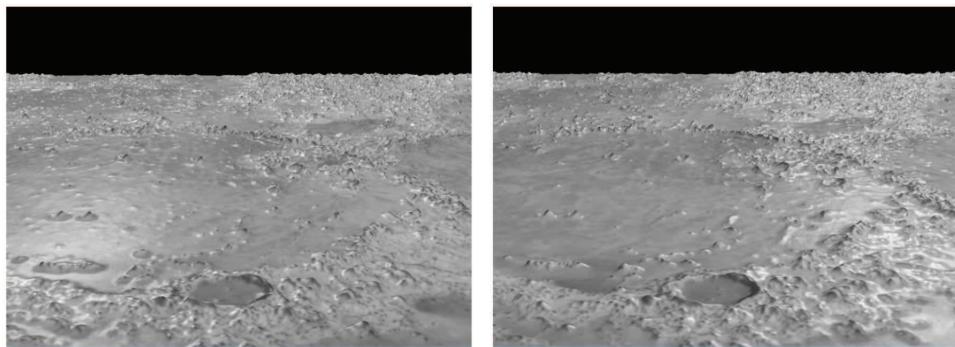
Most of the changes to our code are now simply to implement the standard methods for Phong shading:

- *Java /JOGL application*
We load and activate an additional texture to hold the normal map. We also add code to specify the lighting and materials as we have done in previous applications.
- *Vertex shader*
The only additions are declarations for lighting uniforms and the sampler for the normal map. Lighting code customarily done in the vertex shader is moved to the tessellation evaluation shader, because the vertices aren’t generated until the tessellation stage.
- *Tessellation Control shader*
The only additions are declarations for lighting uniforms and the sampler for the normal map.
- *Tessellation Evaluation shader*
The preparatory code for Phong lighting is now placed in the evaluation shader:

```
varyingVertPos = (mv_matrix * position).xyz;
varyingLightDir = light.position - varyingVertPos;
```

- *Fragment shader*
The typical code sections, described previously, for computing Phong (or Blinn-Phong) lighting are done here, as well as the code to extract normals from the normal map. The lighting result is then combined with the texture image with a weighted sum.

The final result, with height and normal mapping, and Phong lighting, is shown in Figure 12.9. The terrain now responds to lighting. In this example, a positional light has been placed to the left of center in the image on the left, and to the right of center in the image on the right.

**Figure 12.9**

Tessellated terrain with normal map and lighting (light source positioned at left, and at right, respectively).

Although the response to the movement of the light is difficult to tell from a still picture, the reader should be able to discern the diffuse lighting changes, and that specular highlights on the peaks are very different in the two images. This is of course more obvious when the camera or the light source is moving. The results are still imperfect, because the original texture that is incorporated in the output includes shadows that will appear on the rendered result, regardless of lighting.

12.4 CONTROLLING LEVEL OF DETAIL (LOD)

Using instancing to generate millions of vertices in real-time, as in Program 12.4, is likely to place a load on even a well-equipped modern computer. Fortunately, the strategy of dividing the terrain into separate patches, as we have done to increase the number of generated grid vertices, also affords us a nice mechanism for reducing that load.

Of the millions of vertices being generated, many of them aren't necessary. Vertices in patches that are close to the camera *are* important because we expect to discern detail in nearby objects. However, the further the patches are from the camera, the less likely there will even be enough pixels in the rasterization to warrant the number of vertices we are generating!

Changing the number of vertices in a patch based on the distance from the camera is a technique called *level of detail*, or *LOD*. Sellers et al. describe a way of controlling LOD in instanced tessellation [SW15], by modifying the control shader. Program 12.5 shows a simplified version of Sellers' approach. The strategy is to use the patch's perceived size to determine the values of its tessellation levels.

Since the tessellated grid for a patch will eventually be placed within the square defined by the four control points entering the control shader, we can use the locations of the control points relative to the camera to determine how many vertices should be generated for the patch. The steps are as follows:

1. Calculate the screen locations of the four control points by applying the MVP matrix to them.
 2. Calculate the lengths of the sides of the square (i.e., the width and height) defined by the control points (in screen space). Note that even though the four control points form a square, these side lengths can differ because the perspective matrix has been applied.
 3. Scale the lengths' values by a tunable constant, depending on the precision needed for the tessellation levels (based on the amount of detail in the height map).
 4. Add 1 to the scaled length values, to avoid the possibility of specifying a tessellation level of 0 (which would result in no vertices being generated).
 5. Set the tessellation levels to the corresponding calculated width and height values.

Recall that in our instanced example we are not creating just one grid, but 64×64 of them. So the five steps listed above are performed for each patch. Thus, the level of detail varies from patch to patch.

All of the changes are in the control shader and are shown in Program 12.5, with the generated output following in Figure 12.10. Note that the variable `gl_InvocationID` refers to which vertex in the patch is being processed (not which patch is being processed). Therefore, the LOD computation which tells the tessellator how many vertices to generate occurs during the processing of the 0th vertex in each patch.

Program 12.5 Tessellation Level of Detail (LOD)

Tessellation Control Shader

```

vec4 p1 = normalize(mvp * gl_in[1].gl_Position);
vec4 p2 = normalize(mvp * gl_in[2].gl_Position);
float width = length(p2.xy - p0.xy) * subdivisions + 1.0; // perceived "width" of tessellated grid
float height = length(p1.xy - p0.xy) * subdivisions + 1.0; // perceived "height" of tessellated grid
gl_TessLevelOuter[0] = height; // set tessellation levels based on perceived side lengths
gl_TessLevelOuter[1] = width;
gl_TessLevelOuter[2] = height;
gl_TessLevelOuter[3] = width;
gl_TessLevelInner[0] = width;
gl_TessLevelInner[1] = height;
}
// forward texture coordinates and control points to TES as before
tcs_out[gl_InvocationID] = tc[gl_InvocationID];
gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}

```

Applying these control shader changes to the instanced (but not lighted) version of our scene from Figure 12.7, and replacing the height map with Hastings-Trew's improved version shown in Figure 12.8 produces the improved scene, with more realistic horizon detail, shown in Figure 12.10.

In this example it is also useful to change the layout specifier in the evaluation shader from:

```
layout (quads, equal_spacing) in;
```

to:

```
layout (quads, fractional_even_spacing) in;
```

The reason for this modification is difficult to illustrate in still images. In an animated scene, as a tessellated object moves through 3D space, if LOD is used it is sometimes possible to *see* the changes in tessellation levels on the surface of the object as wiggling artifacts called “popping.” Changing from *equal spacing* to *fractional spacing* reduces this effect by making the grid geometry of adjacent patch instances more similar, even if they differ in level of detail. (See Exercises 12.2 and 12.3.)

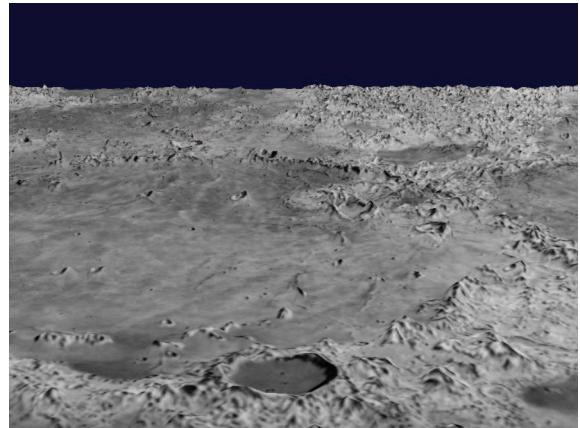


Figure 12.10
Tessellated moon with controlled level of detail (LOD).

Employing LOD can dramatically reduce the load on the system. For example, when animated, the scene might be less likely to appear jerky or lag than could be the case without controlling LOD.

Applying this simple LOD technique to the version that includes Phong shading (i.e., Program 12.4) is a bit trickier. This is because the changes in LOD between adjacent patch instances can in turn cause sudden changes to the associated normal vectors, causing popping artifacts in the lighting! As always, there are tradeoffs and compromises to consider when constructing a complex 3D scene.

SUPPLEMENTAL NOTES

Combining tessellation with LOD is particularly useful in real-time virtual reality applications that require both complex detail for realism and frequent object movement and/or changes in camera position, such as in computer games. In this chapter we have illustrated the use of tessellation and LOD for real-time terrain generation, although it can also be applied in other areas such as in displacement mapping for 3D models (where tessellated vertices are added to the surface of a model and then moved so as to add detail). It is also useful in Computer Aided Design applications.

Sellers et al. extends the LOD technique (shown in Program 12.5) further than we have presented, by also eliminating vertices in patches that are *behind the camera* (they do this by setting their inner and outer levels to zero) [SW15]. This is an example of a *culling* technique, and it is a very useful one because of the load that instanced tessellation can still place on the system.

Exercises

- 12.1** Modify Program 12.1 to experiment with various values for inner and outer tessellation levels, and observing the resulting rendered mesh.
- 12.2** Modify Program 12.1 by changing the layout specifier in the evaluation shader from `equal_spacing` to `fractional_even_spacing`, as shown in Section 12.4. Observe the effect on the generated mesh.
- 12.3** Test Program 12.5 with the layout specifier in the evaluation shader set to `equal_spacing`, and then to `fractional_even_spacing`, as described in Section

12.4. Observe the effects on the rendered surface as the camera moves. You should be able to observe popping artifacts in the first case, which are mostly alleviated in the second case.

- 12.4 (PROJECT)** Modify Program 12.3 to utilize a height map of your own design (you could use the one you built previously in Exercise 10.2). Then add lighting and shadow-mapping so that your tessellated terrain casts shadows. This is a complex exercise, because some of the code in the first and second shadow-mapping passes will need to be moved to the evaluation shader.

References

- [GP16] GIMP Plugin Registry, *normalmap plugin*, accessed July 2016, <http://registry.gimp.org/node/69>.
- [LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).
- [HT16] J. Hastings-Trew, *JHT's Planetary Pixel Emporium*, accessed July 2016, <http://planetpixelemporium.com/>.
- [SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).
- [TS16] Tessellation, Wikipedia, accessed July 2016, <https://en.wikipedia.org/wiki/Tessellation>.

GEOMETRY SHADERS

13.1	<i>Per-Primitive Processing in OpenGL</i>	287
13.2	<i>Altering Primitives</i>	288
13.3	<i>Deleting Primitives</i>	293
13.4	<i>Adding Primitives</i>	294
	<i>Supplemental Notes</i>	297



Immediately following tessellation in the OpenGL pipeline is the *geometry* stage. Here, the programmer has the option of including a *geometry shader*. This stage actually pre-dates tessellation; it became part of the OpenGL core at version 3.2 (in 2009).

Like tessellation, geometry shaders enable the programmer to manipulate *groups* of vertices, in ways that are impossible to do in a vertex shader. In some cases, a task might be accomplished using either a tessellation shader or a geometry shader, as their capabilities overlap in some ways.

13.1 PER-PRIMITIVE PROCESSING IN OPENGL

The geometry shader stage is situated between tessellation and rasterization, within the segment of the pipeline devoted to *primitive processing* (refer back to Figure 2.2). Whereas vertex shaders enable the manipulation of one vertex at a time, and fragment shaders enable the manipulation of one fragment (essentially one pixel) at a time, *geometry shaders enable manipulation of one primitive at a time*.

Recall that primitives are the basic building blocks in OpenGL for drawing objects. Only a few types of primitives are available; we will focus primarily on geometry shaders that manipulate triangles. Thus, when we say that a geometry shader can

manipulate one primitive at a time, we usually mean that the shader has access to *all three vertices of a triangle* at a time. Geometry shaders allow you to:

- Access all vertices in a primitive at once, and
- Output the same primitive unchanged, or
- Output the same primitive changed, or
- Output additional primitives, or
- Delete the primitive (not output it at all).

Similar to the tessellation evaluation shader, incoming vertex attributes are accessible in a geometry shader as *arrays*. However, in a geometry shader, incoming attribute arrays are indexed only up to the primitive size. For example, if the primitives are triangles, then the available indices are 0, 1, and 2. Accessing the vertices themselves is done using the predefined array `gl_in`, as follows:

```
gl_in[2].gl_Position           // position of the 3d vertex
```

Also similar to the tessellation evaluation shader, the geometry shader's output vertex attributes are all *scalets*. That is, the output is a stream of individual vertices (their positions, and other attribute variables, if any) that form primitives.

There is a layout qualifier used to set the primitive input/output types and the output size.

The special GLSL command `EmitVertex()` specifies that a vertex is to be output. The special GLSL command `EndPrimitive()` indicates the completion of building a particular primitive.

The built-in variable `gl_PrimitiveIDIn` is available and holds the ID of the current primitive. The ID numbers start at 0 and count up to the number of primitives minus 1.

Three common categories of operations that we will explore are:

- altering primitives
- deleting primitives
- adding primitives

13.2 ALTERING PRIMITIVES

Geometry shaders are convenient for changing the shape of an object, when that change can be affected through isolated changes to the primitives (typically triangles).

Consider, for example, the torus we rendered previously in Figure 7.12. Suppose that torus represented an inner tube (such as for a tire), and we want to “inflate” it. Simply applying a scale factor in the Java/JOGL code won’t accomplish this, because its fundamental shape wouldn’t change. Giving it the appearance of being “inflated” requires also making the inner hole smaller as the torus stretches into the empty center space.

One way of doing this would be to add the surface normal vector to each vertex. While it is true that this could be done in the vertex shader, let’s do it in the geometry shader, for practice. Program 13.1 shows the GLSL geometry shader code. The other modules are the same as for Program 7.3, with a few minor changes: the fragment shader input names now need to reflect the geometry shader outputs (for example, `varyingNormal` becomes `varyingNormalG`), and the Java/JOGL application needs to compile the geometry shader and attach it to the shader program prior to linking. The new shader is specified as being a geometry shader as follows:

```
int gShader = gl.glCreateShader(GL_GEOMETRY_SHADER);
```

Program 13.1 Geometry Shader: Altering Vertices

```
#version 430

layout (triangles) in;

in vec3 varyingNormal[ ];           // inputs from the vertex shader
in vec3 varyingLightDir[ ];
in vec3 varyingHalfVector[ ];

out vec3 varyingNormalG;           // outputs through the rasterizer to the fragment shader
out vec3 varyingLightDirG;
out vec3 varyingHalfVectorG;

layout (triangle_strip, max_vertices=3) out;

// matrices and lighting uniforms same as before
...
void main (void)
{ // move vertices along the normal, and pass through the other vertex attributes unchanged
    for (i=0; i<3; i++)
    { gl_Position = proj_matrix *
        gl_in[i].gl_Position + normalize(vec4(varyingNormal[i], 1.0)) * 0.4;
        varyingNormalG = varyingNormal[i];
        varyingLightDirG = varyingLightDir[i];
```

```

    varyingHalfVectorG = varyingHalfVector[i];
    EmitVertex();
}
EndPrimitive();
}

```

Note in Program 13.1 that the input variables corresponding to the output variables from the vertex shader are declared as *arrays*. This provides the programmer a mechanism for accessing each of the vertices in the triangle primitive and their attributes using the indices 0, 1, and 2. We wish to move those vertices outward along their surface normals. Both the vertices and the normals have already been transformed to view space with the mv matrix, in the vertex shader. We add the normal to each of the incoming vertex positions (`gl_in[...].gl_Position`), and then apply the projection matrix to the result, producing each output `gl_Position`.

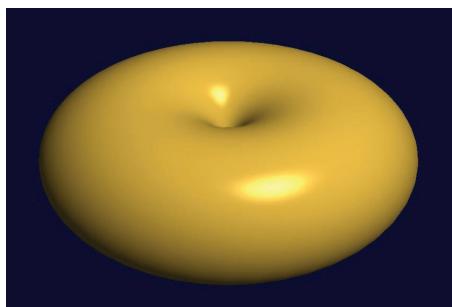


Figure 13.1

"Inflated" torus with vertices altered by geometry shader.

Note the use of the GLSL call `EmitVertex()` that specifies when we have finished computing the output `gl_Position` and its associated vertex attributes and are ready to output a vertex. The `EndPrimitive()` call specifies that we have completed the definition of a set of vertices comprising a primitive (in this case, a triangle). The result is shown in Figure 13.1.

The geometry shader includes two layout qualifiers. The first specifies the input primitive type and must be compatible with the primitive type in the Java-side `glDrawArrays()` call. The options are:

<i>geometry shader input primitive</i>	<i>compatible OpenGL primitives sent from glDrawArrays()</i>	<i>#vertices per invocation</i>
points	GL_POINTS	1
lines	GL_LINES, GL_LINE_STRIP	2
lines_adjacency	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY	4
triangles	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN	3
triangles_adjacency	GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY	6

The various OpenGL primitive types (including “strip” and “fan” types) were described in Chapter 4. “Adjacency” types were introduced in OpenGL for use with geometry shaders, and allow access to vertices adjacent to the primitive. We don’t use them in this book, but they are listed for completeness.

The output primitive type must be *points*, *line_strip*, or *triangle_strip*. Note that the output layout qualifier also specifies the maximum number of vertices the shader outputs in each invocation.

This particular alteration to the torus could have been done more easily in the vertex shader. However, suppose that instead of moving each vertex outward along its own surface normal, we wished instead to move each *triangle* outward along its surface normal, in effect “exploding” the torus triangles outward. The vertex shader cannot do that, because computing a normal for the triangle requires averaging the vertex normals of all three triangle vertices, and the vertex shader only has access to the vertex attributes of one vertex in the triangle at a time. We can, however, do this in the geometry shader, because the geometry shader *does* have access to all three vertices in each triangle. We average them to compute a surface normal for the triangle, then add that averaged normal to each of the vertices in the triangle primitive. Figure 13.2, Figure 13.3, and Figure 13.4 show the averaging of the surface normals, the modified geometry shader main() code, and the resulting output, respectively.

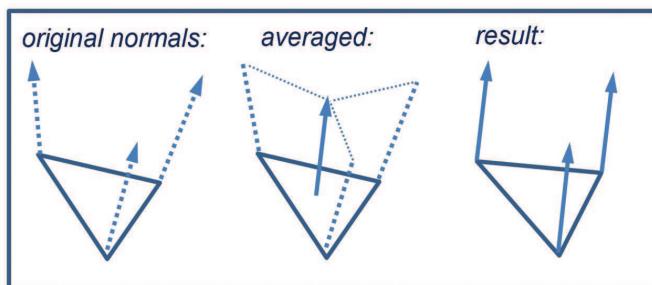


Figure 13.2

Applying averaged triangle surface normal to triangle vertices.

The appearance of the “exploded” torus can be improved by ensuring that the inside of the torus is also visible (normally those triangles are culled by OpenGL because they are “back-facing”). One way of doing this is to render the torus twice, once in the normal manner, and once with winding order reversed (reversing the

```

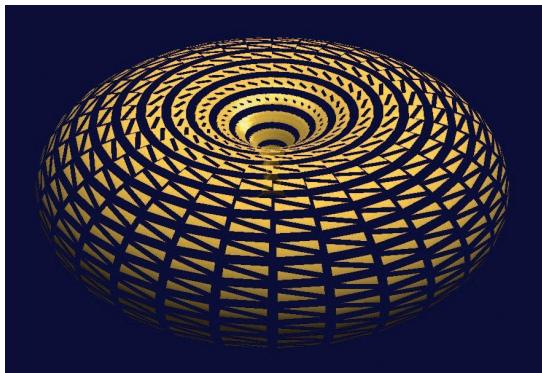
void main (void)
{
    // average the three triangle vertex normals, creating a single triangle surface normal
    vec4 triangleNormal =
        vec4((varyingNormal[0] + varyingNormal[1] + varyingNormal[2]) / 3.0), 1.0);

    // move all three vertices outward along the same normal
    for (i=0; i<3; i++)
    {
        gl_Position = proj_matrix * (gl_in[i].gl_Position + normalize(triangleNormal) * 0.4);
        varyingNormalG = varyingNormal[i];
        varyingLightDirG = varyingLightDir[i];
        varyingHalfVectorG = varyingHalfVector[i];
        EmitVertex();
    }
    EndPrimitive();
}

```

Figure 13.3

Modified geometry shader for "exploding" the torus.

**Figure 13.4**

"Exploded" torus.

changes to display() function:

```

...
// draw front-facing triangles – enable lighting
gl.glUniform1i(enable_lighting_location, 1);
gl.glFrontFace(GL_CCW);
gl.glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);

// draw back-facing triangles – disable lighting
gl.glUniform1i(enable_lighting_location, 0);
gl.glFrontFace(GL_CW);
gl.glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);

```

winding order effectively switches which faces are front-facing and which are back-facing). We also send a flag to the shaders (in a uniform) to disable diffuse and specular lighting on the back-facing triangles, to make them less prominent. The changes to the code are as follows:

modification to fragment shader:

```
...
if (enableLighting == 1)
{ fragColor = ... // when rendering front faces, use normal lighting computations
}
else // when rendering back faces, enable only the ambient lighting component
{ fragColor = globalAmbient * material.ambient + light.ambient * material.ambient;
}
```

The resulting “exploded” torus, including back faces, is shown in Figure 13.5.

13.3 DELETING PRIMITIVES

A common use for geometry shaders is to build richly ornamental objects out of simple ones, by judiciously deleting some of the primitives. For example, removing some of the triangles from our torus can turn it into a sort of complex latticed structure that would be more difficult to model from scratch. A geometry shader that does this is shown in Program 13.2, and the output is shown in Figure 13.6.

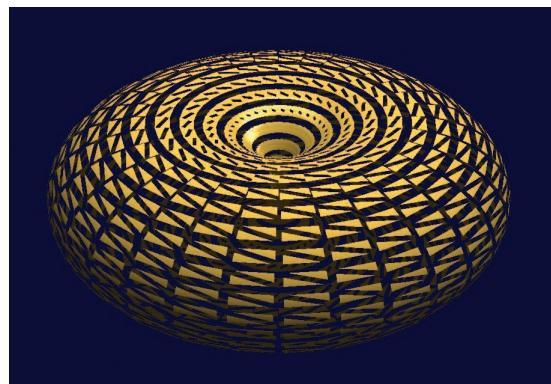


Figure 13.5
“Exploded” torus including back faces.

Program 13.2 Geometry: Delete Primitives

```
// inputs, outputs, and uniforms as before
...
void main (void)
{ if ( mod(gl_PrimitiveIDIn,3) != 0 )
    { for (int i=0; i<3; i++)
        { gl_Position = proj_matrix * gl_in[i].gl_Position;
          varyingNormalG = varyingNormal[i];
          varyingLightDirG = varyingLightDir[i];
          varyingHalfVectorG = varyingHalfVector[i];
          EmitVertex();
        }
    }
    EndPrimitive();
}
```

No other changes to the code are necessary. Note the use of the mod function—all vertices are passed through, except the first of every three primitives, which

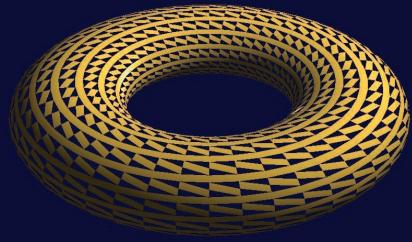


Figure 13.6
Geometry shader: primitive deletion.

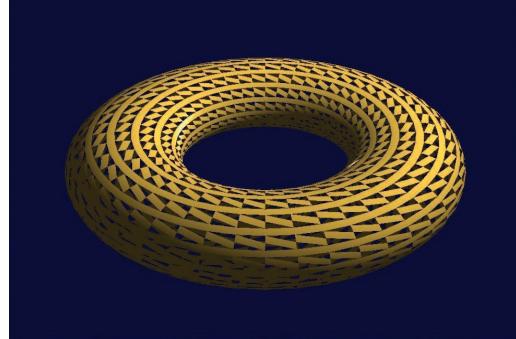


Figure 13.7
Primitive deletion showing back faces.

are ignored. Here too, rendering the back-facing triangles can improve realism, as shown in Figure 13.7.

13.4 ADDING PRIMITIVES

Perhaps the most interesting and powerful use of geometry shaders is for adding additional vertices and/or primitives to a model being rendered. This makes it possible to do such things as increase the detail in an object to improve height mapping, or to change the shape of an object completely.

Consider the following example, where we change each triangle in the torus to a tiny pyramid.

Our strategy, similar to our previous “exploded” torus example, is illustrated in Figure 13.8. The vertices of an incoming triangle primitive are used to define the base of a pyramid. The walls of the pyramid are constructed of those vertices, and of a new point (called the “spike point”) computed by averaging the normals of the original vertices:

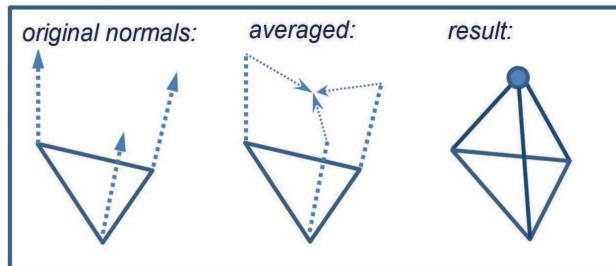


Figure 13.8
Converting triangles to pyramids.

The geometry shader in Program 13.3 does this for each triangle primitive in the torus. For each incoming triangle, it outputs three triangle primitives, for a total of nine vertices.

Note the `setAttribValues()` function in Program 13.3. Its purpose is to assign appropriate values to the outgoing vertex attributes for each vertex in the primitive. In this example, it simply copies the incoming vertex attributes for the *n*th incoming vertex, into each of the three vertices in the corresponding outgoing *n*th face of the constructed tiny pyramid. The particular attributes are for facilitating lighting, and the fact that the outgoing normal for each pyramid face has been set to an original torus normal represents a simplification—actual normals for the pyramid sides would be considerably different.

Program 13.3 Geometry: Add Primitives

```

...
// variables and uniforms as before, except:
layout (triangle_strip, max_vertices = 9) out;
...

void setAttribValues(int n)
{
    varyingNormalG = varyingNormal[n];
    varyingLightDirG = varyingLightDir[n];
    varyingVertPosG = varyingVertPos[n];
}

void main (void)
{
    // finds the endpoints of the surface normals:
    vec4 sp0 = gl_in[0].gl_Position + vec4(varyingNormal[0],1.0) * 0.1;           // the 0.1 is a scale factor
    vec4 sp1 = gl_in[1].gl_Position + vec4(varyingNormal[1],1.0) * 0.1;
    vec4 sp2 = gl_in[2].gl_Position + vec4(varyingNormal[2],1.0) * 0.1;

    // averages the endpoints of the surface normals:
    float x = (sp0.x + sp1.x + sp2.x) / 3.0;
    float y = (sp0.y + sp1.y + sp2.y) / 3.0;
    float z = (sp0.z + sp1.z + sp2.z) / 3.0;

    // the endpoints form the additional pyramid vertex:
    vec4 spikePoint = vec4(x,y,z,1);

    // triangle #1
    gl_Position = proj_matrix * gl_in[0].gl_Position;
    setAttribValues(0); EmitVertex();
}

```

```

gl_Position = proj_matrix * gl_in[1].gl_Position;
setAttribValues(1); EmitVertex();
gl_Position = proj_matrix * spikePoint;
setAttribValues(2); EmitVertex(); EndPrimitive();

// triangle #2
gl_Position = proj_matrix * gl_in[2].gl_Position;
setAttribValues(2); EmitVertex();
gl_Position = proj_matrix * gl_in[0].gl_Position;
setAttribValues(0); EmitVertex();
gl_Position = proj_matrix * spikePoint;
setAttribValues(1); EmitVertex(); EndPrimitive();

// triangle #3
gl_Position = proj_matrix * gl_in[1].gl_Position;
setAttribValues(1); EmitVertex();
gl_Position = proj_matrix * gl_in[2].gl_Position;
setAttribValues(2); EmitVertex();
gl_Position = proj_matrix * spikePoint;
setAttribValues(0); EmitVertex(); EndPrimitive();
}

```

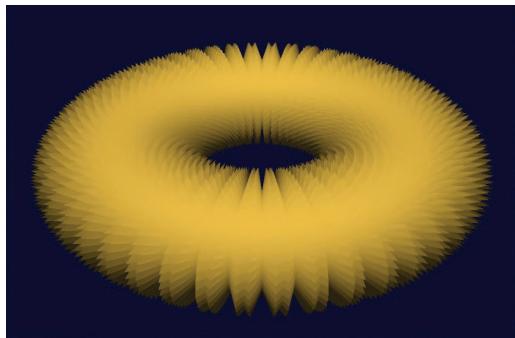


Figure 13.9
Geometry shader: primitive addition.

The resulting output is shown in Figure 13.9. This rendering uses Phong shading with specular highlighting disabled—because we have not computed surface normals for the newly added primitives (specular highlights would appear noticeably unrealistic in the absence of sufficiently detailed normals). Of course, the result could be improved by calculating the proper normals for each of the newly created triangle primitives, and by incorporating shadow mapping (also not done here).

Careful application of this technique can enable the simulation of spikes, thorns, hair, and other fine surface protrusions, as well as the reverse, such as indentations and craters ([DV14], [TR13], and [KS16]).

SUPPLEMENTAL NOTES

One of the appeals of geometry shaders is that they are relatively easy to use. Although many applications for which geometry shaders are used *could* be achieved using tessellation, the mechanism of geometry shaders often makes them easier to implement and debug. Of course, the relative fit of geometry versus tessellation depends on the particular application.

Exercises

- 13.1** Modify Program 13.1 so that it moves each vertex slightly towards the center of its primitive triangle. The result should look similar to the exploded torus in Figure 13.5, but without the overall change in torus size.
- 13.2** Modify Program 13.2 so that it deletes one of the other vertices in each triangle primitive, and observe the effect on the resulting rendered torus.
- 13.3 (RESEARCH & PROJECT)** Modify Program 13.3 so that it adds outward-facing *line segments* rather than triangles.

References

- [DV14] J. deVries, *LearnOpenGL*, 2014, Accessed July 2016, <http://www.learnopengl.com/>.
- [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).
- [TR13] P. Trettner, *Prototype Grass* (blog), 2013, Accessed July 2016, <https://upvoid.com/devblog/2013/02/prototype-grass/>.

OTHER TECHNIQUES

14.1	<i>Fog</i>	299
14.2	<i>Compositing / Blending / Transparency</i>	302
14.3	<i>User-Defined Clipping Planes</i>	308
14.4	<i>3D Textures</i>	310
14.5	<i>Noise</i>	316
14.6	<i>Noise Application – Marble</i>	321
14.7	<i>Noise Application – Wood</i>	323
14.8	<i>Noise Application – Clouds</i>	327
14.9	<i>Noise Application – Special Effects</i>	332
	<i>Supplemental Notes</i>	334



In this chapter we explore a variety of techniques utilizing the tools we have learned throughout the book. Some we will develop fully, while for others we will offer a more cursory description. Graphics programming is a huge field, and this chapter is by no means comprehensive, but rather an introduction to just a few of the creative effects that have been developed over the years.

14.1 FOG

Usually when people think of fog, they think of early misty mornings with low visibility. In truth, atmospheric haze (such as fog) is more common than most of us think. The majority of the time, there is some degree of haze in the air, and although we have become accustomed to seeing it, we don't usually realize it is there. So we can enhance the realism in our outdoor scenes by introducing fog—even if only a small amount.

Fog also can enhance the sense of depth. When close objects have better *clarity* than distant objects, it is one more visual cue that our brains can use to decipher the topography of a 3D scene.

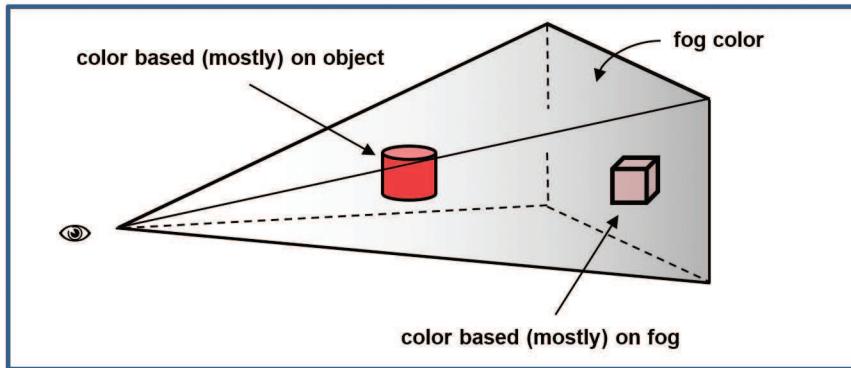


Figure 14.1
Fog: blending based on distance.

There are a variety of methods for simulating fog, from very simple ones to sophisticated models that include light scattering effects. However, even very simple approaches can be effective. One such method is to blend the actual pixel color with another color (the “fog” color, typically gray or bluish-gray—also used for the background color), based on the distance the object is from the eye.

Figure 14.1 illustrates the concept. The eye (camera) is shown at the left, and two red objects are placed in the view frustum. The cylinder is closer to the eye, so it is mostly its original color (red); the cube is further from the eye, so it is mostly fog color. For this simple implementation, virtually all of the computations can be performed in the fragment shader.

Program 14.1 shows the relevant code for a very simple fog algorithm that uses a linear blend from object color to fog color based on the distance from the camera to the pixel. Specifically, this example adds fog to the height mapping example from Program 10.4.

Program 14.1 Simple Fog Generation

Vertex (or Tessellation Evaluation) shader

```
...
out vec3 vertEyeSpacePos;
...
// compute vertex position in eye space, without perspective, and send it to the fragment shader.
// (the variable "p" is the height-mapped vertex, as described earlier in Program 10-4)
vertEyeSpacePos = (mv * p).xyz;
```

Fragment shader

```
...
in vec3 vertEyeSpacePos;
out vec3 fragColor;
...
void main(void)
{
    vec4 fogColor = vec4(0.7, 0.8, 0.9, 1.0); // bluish gray
    float fogStart = 0.0;
    float fogEnd = 1.5;

    // the distance from the camera to the vertex in eye space is simply the length of a
    // vector to that vertex, because the camera is at (0,0,0) in eye space.
    float dist = length(vertEyeSpace.xyz);
    float fogFactor = clamp(((fogEnd - dist) / (fogEnd - fogStart)), 0.0, 1.0);
    fragColor = mix(fogColor, (texture(t,tc), fogFactor);
}
```

The variable `fogColor` specifies a color for the fog. The variables `fogStart` and `fogEnd` specify the range (in eye space) over which the output color transitions from object color to fog color, and can be tuned to meet the needs of the scene. The percentage of fog mixed with the object color is calculated in the variable `fogFactor`, which is the ratio of how close the vertex is to `fogEnd` to the total length of the transition region. The GLSL `clamp()` function is used to restrict this ratio to being between the values 0.0 and 1.0. The GLSL `mix()` function then returns a weighted average of fog color and object color, based on the value of `fogFactor`. Figure 14.2 shows the addition of fog to a scene with height mapped terrain. (A rocky texture from [LU16] has also been applied.)

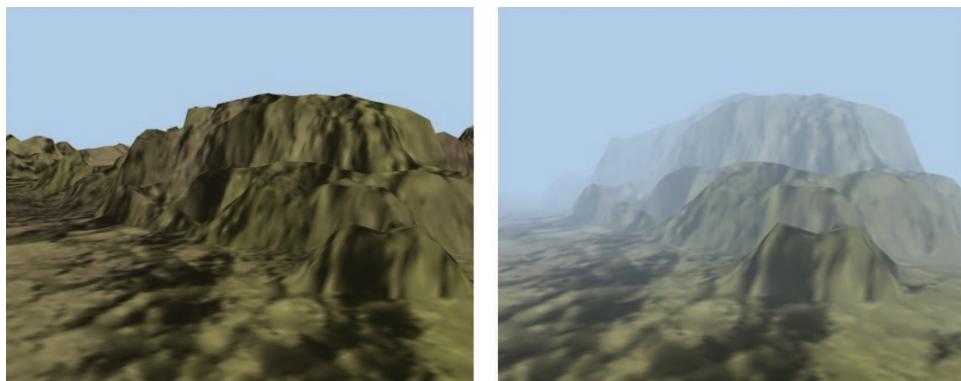


Figure 14.2

Fog example.

14.2 COMPOSITING / BLENDING / TRANSPARENCY

We have already seen a few examples of blending—in the supplementary notes for Chapter 7, and just above in our implementation of fog. However, we haven't yet seen how to utilize the blending (or *compositing*) capabilities that follow after the fragment shader, during pixel operations (recall the pipeline sequence shown in Figure 2.2). It is there that *transparency* is handled, which we look at now.

Throughout this book we have made frequent use of the `vec4` data type, to represent 3D points and vectors in a homogenous coordinate system. You may have noticed that we also frequently use a `vec4` to store *color* information, where the first three values consist of red, green, and blue, and the fourth element is—what?

The fourth element in a color is called the *alpha channel*, and specifies the *opacity* of the color. Opacity is a measure of how *non-transparent* the pixel color is. An alpha value of 0 means “no opacity,” or completely transparent. An alpha value of 1 means “fully opaque,” not at all transparent. In a sense, the “transparency” of a color is $1-\alpha$, where α is the value of the alpha channel.

Recall from Chapter 2 that pixel operations utilize the *Z-buffer*, which achieves hidden surface removal by replacing an existing pixel color when another object's location at that pixel is found to be closer. We actually have more control over this process—we may choose to *blend* the two pixels.

When a pixel is being rendered, it is called the “source” pixel. The pixel already in the frame buffer (presumably rendered from a previous object) is called the

“destination” pixel. OpenGL provides many options for deciding which of the two pixels, or what sort of combination of them, ultimately is placed in the frame buffer. Note that the pixel operations step is not a programmable stage—so the OpenGL tools for configuring the desired compositing are found in the Java/JOGL application, rather than in a shader.

The two OpenGL functions for controlling

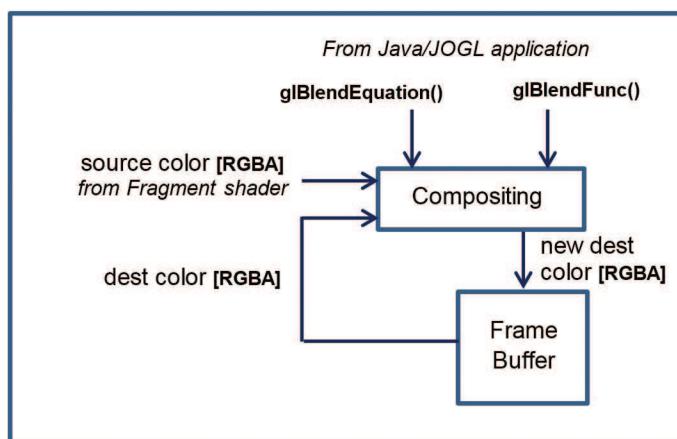


Figure 14.3

OpenGL compositing overview.

compositing are `glBlendEquation(mode)` and `glBlendFunc(srcFactor, destFactor)`. Figure 14.3 shows an overview of the compositing process.

The compositing process works as follows:

1. The *source* and *destination* pixels are multiplied by *source factor* and *destination factor*, respectively. The source and destination factors are specified in the `blendFunc()` function call.
2. The specified *blendEquation* is then used to combine the modified source and destination pixels to produce a new destination color. The blend equation is specified in the `glBlendEquation()` call.

The most common options for `glBlendFunc()` parameters (i.e., `srcFactor` and `destFactor`) are:

<i>glBlendFunc()</i> parameter	<i>resulting srcFactor or destFactor</i>
<code>GL_ZERO</code>	(0,0,0,0)
<code>GL_ONE</code>	(1,1,1,1)
<code>GL_SRC_COLOR</code>	($R_{src}, G_{src}, B_{src}, A_{src}$)
<code>GL_ONE_MINUS_SRC_COLOR</code>	(1,1,1,1) - ($R_{src}, G_{src}, B_{src}, A_{src}$)
<code>GL_DST_COLOR</code>	($R_{dest}, G_{dest}, B_{dest}, A_{dest}$)
<code>GL_ONE_MINUS_DST_COLOR</code>	(1,1,1,1) - ($R_{dest}, G_{dest}, B_{dest}, A_{dest}$)
<code>GL_SRC_ALPHA</code>	($A_{src}, A_{src}, A_{src}, A_{src}$)
<code>GL_ONE_MINUS_SRC_ALPHA</code>	(1,1,1,1) - ($A_{src}, A_{src}, A_{src}, A_{src}$)
<code>GL_DST_ALPHA</code>	($A_{dest}, A_{dest}, A_{dest}, A_{dest}$)
<code>GL_ONE_MINUS_DST_ALPHA</code>	(1,1,1,1) - ($A_{dest}, A_{dest}, A_{dest}, A_{dest}$)
<code>GL_CONSTANT_COLOR</code>	($R_{blendColor}, G_{blendColor}, B_{blendColor}, A_{blendColor}$)
<code>GL_ONE_MINUS_CONSTANT_COLOR</code>	(1,1,1,1) - ($R_{blendColor}, G_{blendColor}, B_{blendColor}, A_{blendColor}$)
<code>GL_CONSTANT_ALPHA</code>	($A_{blendColor}, A_{blendColor}, A_{blendColor}, A_{blendColor}$)
<code>GL_ONE_MINUS_CONSTANT_ALPHA</code>	(1,1,1,1) - ($A_{blendColor}, A_{blendColor}, A_{blendColor}, A_{blendColor}$)
<code>GL_ALPHA_SATURATE</code>	($f, f, f, 1$) where $f = \min(A_{src}, 1)$

Those options that indicate a “blendColor” (GL_CONSTANT_COLOR, etc.) require an additional call to `glBlendColor()` to specify a constant color that will be used to compute the blend function result. There are a few additional blend functions that aren’t listed above, not described here.

The possible options for the `glBlendEquation()` parameter (i.e., mode) are:

<code>GL_FUNC_ADD</code>	$\text{result} = \text{source}_{\text{RGBA}} + \text{destination}_{\text{RGBA}}$
<code>GL_FUNC_SUBTRACT</code>	$\text{result} = \text{source}_{\text{RGBA}} - \text{destination}_{\text{RGBA}}$
<code>GL_FUNC_REVERSE_SUBTRACT</code>	$\text{result} = \text{destination}_{\text{RGBA}} - \text{source}_{\text{RGBA}}$
<code>GL_MIN</code>	$\text{result} = \min(\text{source}_{\text{RGBA}}, \text{destination}_{\text{RGBA}})$
<code>GL_MAX</code>	$\text{result} = \max(\text{source}_{\text{RGBA}}, \text{destination}_{\text{RGBA}})$

The `glBlendFunc()` defaults are `GL_ONE` (1.0) for `srcFactor`, and `GL_ZERO` (0.0) for `destFactor`. The default for `glBlendEquation()` is `GL_FUNC_ADD`. Thus, by default, the source pixel is untouched (multiplied by one), the destination pixel is scaled to zero, and the two are added—meaning that the source pixel becomes the frame buffer color.

There are also the commands `glEnable(GL_BLEND)` and `glDisable(GL_BLEND)`, which can be used to tell OpenGL to apply the specified blending, or to ignore it.

We won’t illustrate the effects of all of the options here, but will walk through some illustrative examples.

Suppose we specify the following settings in the Java/JOGL application:

- `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`
- `glBlendEquation(GL_FUNC_ADD)`

Compositing would proceed as follows:

1. The source pixel is scaled by its alpha value.
2. The destination pixel is scaled by $1-\text{srcAlpha}$ (the *source transparency*).
3. The pixel values are added together.

For example, if the source pixel is red, with 75% opacity: [1, 0, 0, 0.75], and the destination pixel contains completely opaque green: [0, 1, 0, 1], then the result placed in the frame buffer would be:

```

srcPixel * srcAlpha = [0.75, 0, 0, 0.5625]
destPixel * (1-srcAlpha) = [0, 0.25, 0, 0.25]
resulting pixel = [0.75, 0.25, 0, 0.8125]

```

That is, mostly red, with some green, and mostly solid. The overall effect of the settings is to let the destination show through by an amount corresponding to the source pixel's transparency. In this example, the pixel in the frame buffer is green, and the incoming pixel is red with 25% transparency (75% opacity). So some green is allowed to show through the red.

It turns out that these settings for blend function and blend equation work well in many cases. Let's apply them to a practical example in a scene containing two 3D models: a torus, and a pyramid in front of the torus. Figure 14.4 shows such a scene, on the left with an opaque pyramid, and on the right with the pyramid's alpha value set to 0.8. Lighting has been added.

For many applications—such as creating a flat “window” as part of a model of a house—this simple implementation of transparency may be sufficient. However, in the example shown in Figure 14.4, there is a fairly obvious inadequacy. Although the pyramid model is now effectively transparent, an actual transparent pyramid should reveal not only the objects behind it, but also *its own back surfaces*.

Actually, the reason that the back faces of the pyramid did not appear is because we enabled back-face culling. A reasonable idea might be to disable

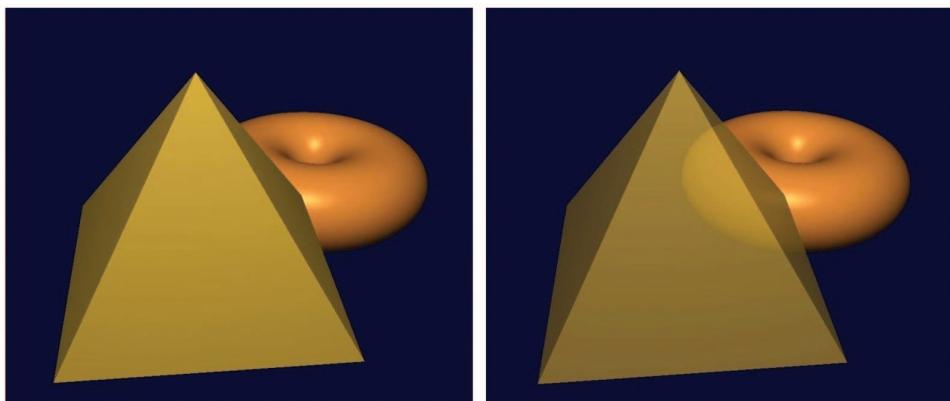


Figure 14.4

Pyramid with alpha=1.0 (left), and alpha=0.8 (right).

back-face culling while drawing the pyramid. However, this often produces other artifacts, as shown in Figure 14.5 (on the left). The problem with simply disabling back-face culling is that the effects of blending depend on the order that surfaces are rendered (because that determines the source and destination pixels), and we don't always have control over the rendering order. It is generally advantageous to render opaque objects first, as well as objects that are in the back (such as the torus) before any transparent objects. This also holds true for the surfaces of the pyramid, and in this case the reason that the two triangles comprising the base of the pyramid appear different is that one of them was rendered before the front of the pyramid, and one was rendered after. Artifacts such as this are sometimes called “ordering” artifacts, and can manifest in transparent models because we cannot always predict the order that its triangles will be rendered.

We can solve the problem in our pyramid example by rendering the front and back faces separately, ourselves, starting with the back faces. Program 14.2 shows the code for doing this. We specify the alpha value for the pyramid by passing it to the shader program in a uniform variable, then apply it in the fragment shader by substituting the specified alpha into the computed output color.

Note also that for lighting to work properly, we must flip the normal vector when rendering the back faces. We accomplish this by sending a flag to the vertex shader, where we then flip the normal vector.

Program 14.2 Two-Pass Blending for Transparency

JOGL/Java application - in display() for rendering pyramid:

```
...
gl.glEnable(GL_CULL_FACE);
...
gl.glEnable(GL_BLEND);           // configure blend settings
gl.glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
gl.glBlendEquation(GL_FUNC_ADD);

gl.glCullFace(GL_FRONT); // render pyramid back faces first
gl.glProgramUniform1f(rendering_program2, alpha_location, 0.3f); // back faces set to very transparent
gl.glProgramUniform1f(rendering_program2, flip_location, -1.0f); // flip normals on back faces
gl.glDrawArrays(GL_TRIANGLES, 0, pyramid.getNumVertices());
```

```

gl.glCullFace(GL_BACK);      // then render pyramid front faces
gl.glProgramUniform1f(rendering_program2, alpha_location, 0.8f); // front faces are slightly transparent
gl.glProgramUniform1f(rendering_program2, flip_location, 1.0f); // don't flip normals on front faces
gl.glDrawArrays(GL_TRIANGLES, 0, pyramid.getNumVertices());

gl.glDisable(GL_BLEND);

```

Vertex shader:

```

...
if (flipNormal < 0) varyingNormal = -varyingNormal;
...

```

Fragment shader:

```

...
fragColor = globalAmbient * material.ambient + ... etc.      // same as for Blinn-Phong lighting.
fragColor = vec4(fragColor.xyz, alpha); // replace alpha value with one sent in uniform variable

```

The result of this “two-pass” solution is shown in Figure 14.5, on the right.

Although it works well here, the two-pass solution shown in Program 14.2 is not always adequate. For example, some more complex models may have hidden surfaces that are *front-facing*, and if such an object were made transparent, our algorithm would fail to render those hidden front-facing portions of the model. Alec Jacobson describes a 5-pass sequence that works in a large number of cases [JA12].

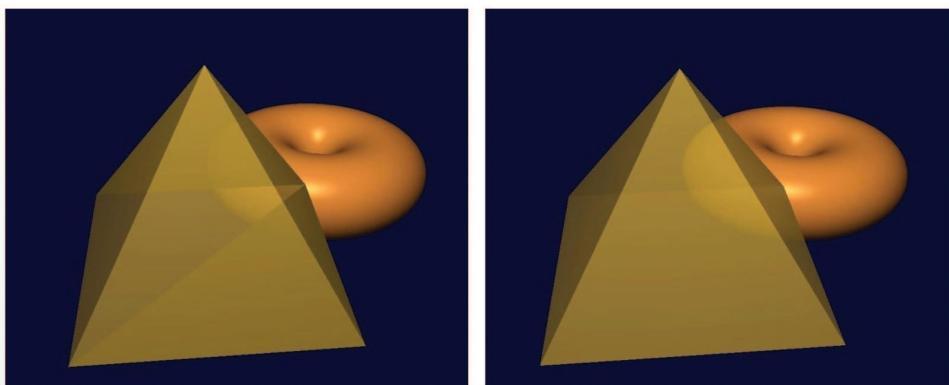


Figure 14.5

Transparency & back faces: ordering artifacts (left) and two-pass correction (right).

14.3 USER-DEFINED CLIPPING PLANES

OpenGL includes the capability to specify clipping planes beyond those defined by the view frustum. One use for a user-defined clipping plane is to slice a model. This makes it possible to create complex shapes by starting with a simple model, and slicing sections off of it.

A clipping plane is defined according to the standard mathematical definition of a plane:

$$ax + by + cz + d = 0$$

where a , b , c , and d are parameters defining a particular plane in 3D space with X, Y, and Z axes. The parameters represent a vector (a,b,c) normal to the plane, and a distance d from the origin to the plane. Such a plane can be specified in the vertex shader using a `vec4`, as follows:

```
vec4 clip_plane = vec4(0.0, 0.0, -1.0, 0.2);
```

This would correspond to the plane:

$$(0.0)x + (0.0)y + (-1.0)z + 0.2 = 0$$

The clipping can then be achieved, also in the vertex shader, by using the built-in GLSL variable `gl_ClipDistance[]`, as in the following example:

```
gl_ClipDistance[0] = dot(clip_plane.xyz, vertPos) + clip_plane.w;
```

In this example, `vertPos` refers to the vertex position coming into the vertex shader in a vertex attribute (such as in a VBO) and `clip_plane` was defined above. We then compute the signed distance from the clipping plane to the incoming vertex (shown in Chapter 3), which is either zero if the vertex is on the plane, or is negative or positive depending on which side of the plane the vertex lies. The subscript on the `gl_ClipDistance` array enables multiple clipping distances (i.e., multiple planes) to be defined. The maximum number of user clipping planes that can be defined depends on the graphics card's OpenGL implementation.

User-defined clipping must then be *enabled* in the Java/JOGL application. There are built-in OpenGL identifiers `GL_CLIP_DISTANCE0`, `GL_CLIP_DISTANCE1`, and so on, corresponding to each `gl_ClipDistance[]` array element. The 0th user-defined clipping plane can be enabled, for example, as follows:

```
gl.glEnable(GL_CLIP_DISTANCE0);
```

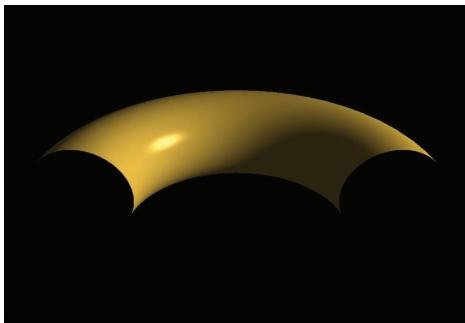


Figure 14.6
Clipping a torus.

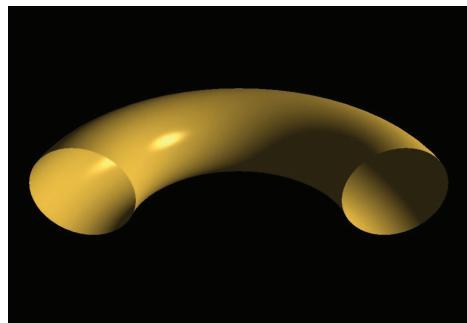


Figure 14.7
Clipping with back faces.

Applying the above steps to our lighted torus results in the output shown in Figure 14.6, in which the front half of the torus has been clipped. (A rotation has also been applied to provide a clearer view.)

It may appear that the bottom portion of the torus has also been clipped, but that is because the inside faces of the torus were not rendered. When clipping reveals the inside surfaces of a shape, it is necessary to render them as well, or the model will appear incomplete (as it does in Figure 14.6).

Rendering the inner surfaces requires making a second call to `gl_DrawArrays()`, with the winding order reversed. Additionally, it is necessary to reverse the surface normal vector when rendering the back-facing triangles (as was done in the previous section). The relevant modifications to the Java application and the vertex shader are shown in Program 14.3, with the output shown in Figure 14.7.

Program 14.3 Clipping with Back Faces

Java/JOGL application:

```
public void display(GLAutoDrawable drawable)
{
    ...
    int flipLocation = gl.glGetUniformLocation(rendering_program, "flipNormal");
    ...
    gl glEnable(GL_CLIP_DISTANCE0);
    // normal drawing of external faces
    gl glUniform1i(flipLocation, 0);
    gl.glFrontFace(GL_CCW);
    gl.glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);
```

```
// rendering of back faces with normals reversed
gl.glUniform1i(flipLocation, 1);
gl.glFrontFace(GL_CW);
gl.glDrawArrays(GL_TRIANGLES, 0, numTorusVertices);
}
```

Vertex shader:

```
...
vec4 clip_plane = vec4(0.0, 0.0, -1.0, 0.1);
uniform int flipNormal; // flag for inverting normal
...
void main(void)
{
    ...
    if (flipNormal==1) varyingNormal = -varyingNormal;
    ...
    gl_ClipDistance[0] = dot(clip_plane.xyz, vertPos) - clip_plane.w;
    ...
}
```

14.4 3D TEXTURES

Whereas 2D textures contain image data indexed by two variables, 3D textures contain the same type of image data, but in a 3D structure that is indexed by three variables. The first two dimensions still represent *width* and *height* in the texture map; the third dimension represents *depth*.

Because the data in a 3D texture is stored in a similar manner as for 2D textures, it is tempting to think of a 3D texture as a sort of 3D “image.” However, we generally don’t refer to 3D texture source data as a 3D image, because there are no commonly used image file formats for this sort of structure (i.e., there is nothing akin to a 3D JPEG, at least not one that is truly 3-dimensional). Instead, we suggest thinking of a 3D texture as a sort of substance into which we will submerge (or “dip”) the object being textured, resulting in the object’s surface points obtaining their colors from the corresponding locations in the texture. Alternatively, it can be useful to imagine that the object is being “carved” out of the 3D texture “cube,” much like a sculptor carves a figure out of a single solid block of marble.

OpenGL has support for 3D texture objects. In order to use them, we need to learn how to build the 3D texture, and how to use it to texture an object.

Unlike 2D textures, which can be built from standard image files, 3D textures are usually generated *procedurally*. As was done previously for 2D textures, we decide on a resolution—that is, the number of texels in each dimension. Depending on the colors in the texture, we may build a 3-dimensional array containing those colors. Alternatively, if the texture holds a “pattern” that could be utilized with various colors, we might instead build an array that holds the pattern, such as with 0s and 1s.

For example, we can build a 3D texture that represents horizontal stripes by filling an array with 0s and 1s corresponding to the desired stripe pattern. Suppose that the desired resolution of the texture is 200x200x200 texels, and the texture is comprised of alternating stripes that are each 10 texels high. A simple function that builds such a structure by filling an array with appropriate 0s and 1s in a nested loop (assuming in this case that width, height, and depth variables are each set to 200), would be:

```
void generate3Dpattern()
{   for (int x=0; x<texWidth; x++)
    {   for (int y=0; y<texHeight; y++)
        {   for (int z=0; z<texDepth; z++)
            {   if ((y/10) % 2 == 0)
                tex3Dpattern[x][y][z] = 0.0;
            else
                tex3Dpattern[x][y][z] = 1.0;
        }   }   }
```

The pattern stored in the `tex3Dpattern` array is illustrated in Figure 14.8 with the 0s rendered in blue, and the 1s rendered in yellow.

Texturing an object with the above striped pattern requires the following steps:

1. Generating the pattern as already shown
2. Using the pattern to fill a byte array of desired colors
3. Loading the byte array into a texture object
4. Deciding on appropriate 3D texture coordinates for the object vertices
5. Texturing the object in the fragment shader using an appropriate sampler

Texture coordinates for 3D textures range from 0 to 1, in the same manner as for 2D textures.

Interestingly, step #4 (determining 3D texture coordinates) is usually a lot simpler than one might initially suspect. In fact, it is usually simpler than for 2D textures!

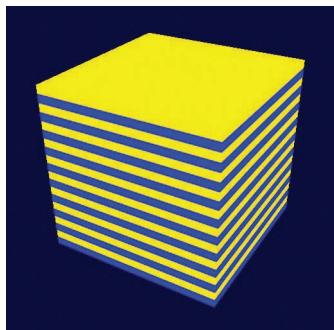


Figure 14.8
Striped 3D texture pattern.

This is because (in the case of 2D textures) since a 3D object was being textured with a 2D image, we needed to decide how to “flatten” the 3D object’s vertices (such as by UV-mapping) to create texture coordinates. But when 3D texturing, both the object and the texture are of the same dimensionality (three). In most cases, we want the object to reflect the texture pattern, as if it were “carved” out of it (or dipped into it). So the vertex locations themselves serve as the texture coordinates! Usually all that is necessary is to apply some simple scaling to ensure that the object’s vertices’ location coordinates map to the 3D texture coordinates’ range (0...1).

Since we are generating the 3D texture procedurally, we need a way of constructing an OpenGL texture map out of generated data. The process for loading data into a texture is similar to what we saw earlier in Section 5.12. In this case, we fill a 3D array with color values, then copy them into a texture object.

Program 14.4 shows the various components for achieving all of the steps listed above, in order to texture an object with blue and yellow horizontal stripes from a procedurally built 3D texture. The desired pattern is built in the generate3Dpattern() function, which stores the pattern in an array named “tex3Dpattern”. The “image” data is then built in the function fillDataArray(), which fills a 3D array with byte data corresponding to the RGB colors R, G, B, and A, each in the range (0...255), according to the pattern. Those values are then copied into a texture object in the load3DTexture() function.

Program 14.4 3D Texturing, Striped Pattern

Java/JOGL application:

```

...
private int texHeight= 200;
private int texWidth = 200;
private int texDepth = 200;
private double[ ][ ][ ] tex3Dpattern = new double[texWidth][texHeight][texDepth];
...

// fill a byte array with RGB blue/yellow values corresponding to the pattern built by generate3Dpattern()
private void fillDataArray(byte data[ ])
{
    for (int i=0; i<texWidth; i++)
    {
        for (int j=0; j<texHeight; j++)
        {
            ...
        }
    }
}

```

```

    { for (int k=0; k<texDepth; k++)
    { if (tex3Dpattern[i][j][k] == 1.0)
        { // yellow color
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+0] = (byte) 255; // red
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+1] = (byte) 255; // green
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+2] = (byte) 0; // blue
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+3] = (byte) 255; // alpha
        }
        else
        { // blue color
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+0] = (byte) 0; // red
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+1] = (byte) 0; // green
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+2] = (byte) 255; // blue
            data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+3] = (byte) 255; // alpha
        }
    } } } } }

// build 3D pattern of stripes
void generate3Dpattern()
{ for (int x=0; x<texWidth; x++)
    { for (int y=0; y<texHeight; y++)
        { for (int z=0; z<texDepth; z++)
            { if ((y/10)%2 == 0)
                tex3Dpattern[x][y][z] = 0.0;
            else
                tex3Dpattern[x][y][z] = 1.0;
        } } } }

// load the sequential byte data array into a texture object
private int load3DTexture()
{ GL4 gl = (GL4) GLContext.getCurrentGL();

    byte[ ] data = new byte[texWidth*texHeight*texDepth*4];
    fillDataArray(data);
    ByteBuffer bb = Buffers.newDirectByteBuffer(data);

    int[ ] textureIDs = new int[1];
    gl glGenTextures(1, textureIDs, 0);
    int textureID = textureIDs[0];
    gl glBindTexture(GL_TEXTURE_3D, textureID);
    gl glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA8, texWidth, texHeight, texDepth);
    gl glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0,
        texWidth, texHeight, texDepth, GL_RGBA, GL_UNSIGNED_INT_8_8_8_8_REV, bb);
    gl.glTexParameter(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    return textureID;
}

```

```

public void init(GLAutoDrawable drawable)
{
    ...
    generate3Dpattern(); // the 3D pattern and texture is only loaded once, so is done from init()
    textureID = load3DTexture();
}

public void display(GLAutoDrawable drawable)
{
    ...
    gl.glActiveTexture(GL_TEXTURE0);
    gl glBindTexture(GL_TEXTURE_3D, textureID);
    gl.glDrawArrays(GL_TRIANGLES, 0, numObjVertices);
}

```

Vertex Shader:

```

...
out vec3 originalPosition; // the original model vertices will be used for texture coordinates
...
layout (binding=0) uniform sampler3D s;
void main(void)
{
    originalPosition = position; // pass original model coordinates for use as 3D texture coordinates
    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}

```

Fragment Shader:

```

...
in vec3 originalPosition; // receive original model coordinates for use as 3D texture coordinates
out vec4 fragColor;
...
layout (binding=0) uniform sampler3D s;
void main(void)
{
    fragColor = texture(s, originalPosition/2.0 + 0.5); // vertices are (-1...+1), tex coords are (0...1)
}

```

In the Java/JOGL application, the `load3Dtexture()` function is similar to the Java AWT `loadTexture()` function shown earlier in Program 5.2. As before, it expects the image data to be formatted as a sequence of bytes corresponding to RGBA color components. The function `fillDataArray()` does this, applying the RGB values for yellow and blue corresponding to the striped pattern built by the `generate3Dpattern()` function and held in the `tex3Dpattern` array. Note also the specification of

texture type `GL_TEXTURE_3D` in the `display()` function.

Since we wish to use the object's vertex locations as texture coordinates, we pass them through from the vertex shader to the fragment shader. The fragment shader then scales them so that they are mapped into the range $(0,1)$ as is standard for texture coordinates. Finally, 3D textures are accessed via a `sampler3D` uniform, which takes three parameters instead of two. We use the vertex's original X, Y, and Z coordinates, scaled to the correct range, to access the texture. The result is shown in Figure 14.9.

More complex patterns can be generated by modifying `generate3Dpattern()`. Figure 14.10 shows a simple change that converts the striped pattern to a 3D checkerboard. The resulting effect is then shown in Figure 14.11. It is worth noting that the effect is very different than would be the case if the dragon's surface had been textured with a 2D checkerboard texture pattern. (See Exercise 14.2.)

```
void generate3Dpattern()
{ int xStep, yStep, zStep, sumSteps;
for (int x=0; x<texWidth; x++)
{ for (int y=0; y<texHeight; y++)
{ for (int z=0; z<texDepth; z++)
{ xStep = (x / 10) % 2;
yStep = (y / 10) % 2;
zStep = (z / 10) % 2;
sumSteps = xStep + yStep + zStep;
if ((sumSteps % 2) == 0)
    tex3Dpattern[x][y][z] = 0.0;
else
    tex3Dpattern[x][y][z] = 1.0;
} } } }
```



Figure 14.9
Dragon object with 3D striped texture.

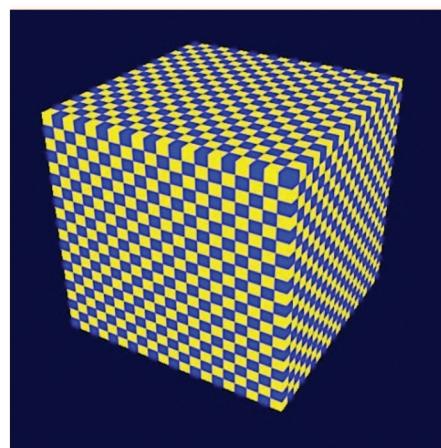


Figure 14.10
Generating a checkerboard 3D texture pattern.

**Figure 14.11**

Dragon with 3D checkerboard texture.

14.5 NOISE

Many natural phenomena can be simulated using randomness, or *noise*. One common technique, *Perlin Noise*, is named after Ken Perlin, who in 1997 received an Academy Award¹ for developing a practical way to generate and use 2D and 3D noise. The method described here is based on Perlin's.

There are many applications of noise in graphics scenes. A few common examples are clouds, terrain, wood grain, minerals (such as veins in marble),

smoke, fire, flames, planetary surfaces, and random movements. In this section, we focus on generating 3D textures containing noise, and then subsequent sections illustrate using the noise data to generate complex materials such as marble and wood, and to simulate animated cloud textures for use with a cube map or skydome. Spatial data (e.g., 2D or 3D) that contains noise is sometimes called a *noise map*.

We start by constructing a 3D texture map out of random data. This can be done using the functions shown in the previous section, with a few modifications. First, we replace the `generate3Dpattern()` function from Program 14.4 with the following simpler `generateNoise()` function:

```
double[ ][ ][ ] noise = new double[n noiseWidth][n noiseHeight][n noiseDepth];
Random random = new Random();
...
void generateNoise()
{
    for (int x=0; x<n noiseWidth; x++)
        { for (int y=0; y<n noiseHeight; y++)
            { for (int z=0; z<n noiseDepth; z++)
                { noise[x][y][z] = random.nextDouble(); } // returns a double in the range (0...1)
        } } }
```

¹ The Technical Achievement Award, given by the Academy of Motion Picture Arts and Sciences.

Next, the `filldataArray()` function from Program 14.4 is modified so that it copies the noise data into the byte array in preparation for loading into a texture object, as follows:

```
private void filldataArray(byte data[])
{
    for (int i=0; i<noiseWidth; i++)
    {
        for (int j=0; j<noiseHeight; j++)
        {
            for (int k=0; k<noiseDepth; k++)
            {
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (byte) (noise[i][j][k] * 255);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (byte) (noise[i][j][k] * 255);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (byte) (noise[i][j][k] * 255);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (byte) 255;
            }
        }
    }
}
```

The rest of Program 14.4 for loading data into a texture object and applying it to a model is unchanged. We can view this 3D noise map by applying it to our simple cube model, as shown in Figure 14.12. In this example, `noiseHeight = noiseWidth = noiseDepth = 256`.

This is a 3D noise map, although it isn't a very useful one. As is, it is just *too* noisy to have very many practical applications. To make more practical, tunable noise patterns, we will replace the `filldataArray()` function with different noise-producing procedures.

Suppose that we fill the data array by “zooming in” to a small subsection of the noise map illustrated above in Figure 14.12, using indexes made smaller by integer division. The modification to the `filldataArray()` function is shown below. The resulting 3D texture can be made more or less “blocky” depending on the “zooming” factor used to divide the index. In Figure 14.13, the textures show the result of zooming in by dividing the indices by zoom factors 8, 16, and 32 (left to right, respectively).

```
private void filldataArray(byte data[])
{
    int zm = 8; // zoom factor
    for (int i=0; i<noiseWidth; i++)
    {
        for (int j=0; j<noiseHeight; j++)
        {
            for (int k=0; k<noiseDepth; k++)
            {
```

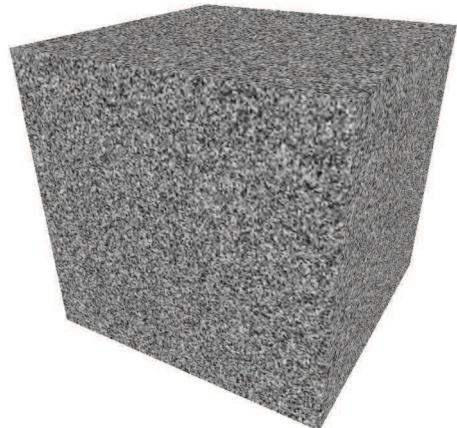
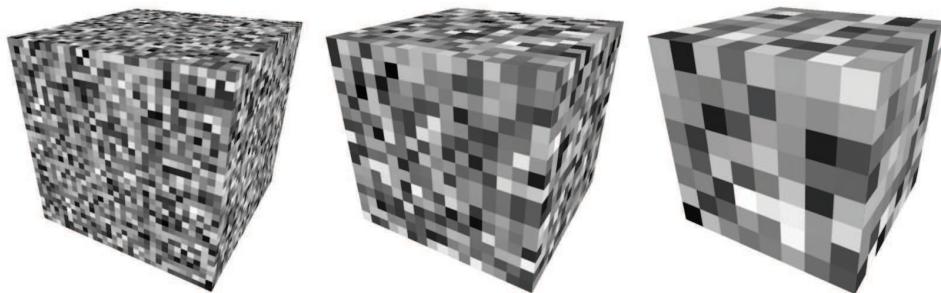


Figure 14.12
Cube textured with 3D noise data.

**Figure 14.13**

"Blocky" 3D noise maps with various "zooming in" factors.

```
{
    data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] =
        (byte) (noise [i/zm] [j/zm] [k/zm] * 255);
    data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] =
        (byte) (noise [i/zm] [j/zm] [k/zm] * 255);
    data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] =
        (byte) (noise [i/zm] [j/zm] [k/zm] * 255);
    data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] =
        (byte) 255;
} } } }
```

The "blockiness" within a given noise map can be *smoothed* by interpolating from each discrete grayscale color value to the next one. That is, for each small "block" within a given 3D texture, we set each texel color within the block by interpolating from its color to its neighboring blocks' colors. The interpolation code is shown below in the function `smoothNoise()`, along with the modified `fillDataArray()` function. The resulting "smoothed" textures (for zooming factors 2, 4, 8, 16, 32, and 64—left to right, top to bottom) then follow in Figure 14.14. Note that the zoom factor is now a `double`, because we need the fractional component to determine the interpolated grayscale values for each texel.

```
private void fillDataArray(byte data[ ])
{
    double zm = 32.0; // zoom factor
    for (int i=0; i<noiseWidth; i++)
    {
        for (int j=0; j<noiseHeight; j++)
        {
            for (int k=0; k<noiseDepth; k++)
            {
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 + 0] =
                    (byte) (smoothNoise (i/zm, j/zm, k/zm) * 255);
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 + 1] =
                    (byte) (smoothNoise (i/zm, j/zm, k/zm) * 255);
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 + 2] =
                    (byte) (smoothNoise (i/zm, j/zm, k/zm) * 255);
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 + 3] =
                    (byte) (smoothNoise (i/zm, j/zm, k/zm) * 255);
            }
        }
    }
}
```

```

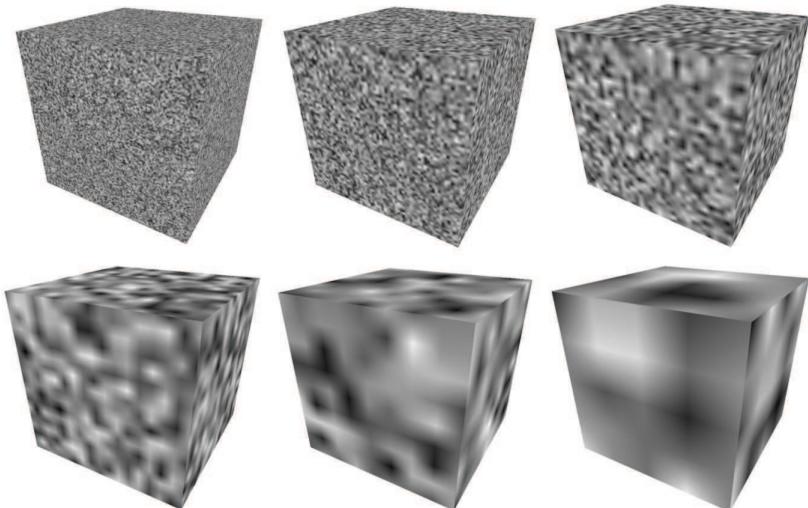
        data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 +3] =
            (byte) 255;
    } } } }

double smoothNoise(double x1, double y1, double z1)
{ // fraction of x1, y1, z1 (% of distance from current block to next block, for this texel)
    double fractX = x1 - (int) x1;
    double fractY = y1 - (int) y1;
    double fractZ = z1 - (int) z1;

    // the indices for neighboring pixels in the X, Y, and Z directions
    int x2 = ((int) x1 + noiseWidth - 1) % noiseWidth;
    int y2 = ((int) y1 + noiseHeight - 1) % noiseHeight;
    int z2 = ((int) z1 + noiseDepth - 1) % noiseDepth;

    // smooth the noise by interpolating the greyscale intensity along all three axes
    double value = 0.0;
    value += fractX      * fractY      * fractZ      * noise [(int) x1] [(int) y1] [(int) z1];
    value += fractX      * (1-fractY)  * fractZ      * noise [(int) x1] [(int) y2] [(int) z1];
    value += (1-fractX)  * fractY      * fractZ      * noise [(int) x2] [(int) y1] [(int) z1];
    value += (1-fractX)  * (1-fractY)  * fractZ      * noise [(int) x2] [(int) y2] [(int) z1];
    value += fractX      * fractY      * (1-fractZ)  * noise [(int) x1] [(int) y1] [(int) z2];
    value += fractX      * (1-fractY)  * (1-fractZ)  * noise [(int) x1] [(int) y2] [(int) z2];
    value += (1-fractX)  * fractY      * (1-fractZ)  * noise [(int) x2] [(int) y1] [(int) z2];
    value += (1-fractX)  * (1-fractY)  * (1-fractZ)  * noise [(int) x2] [(int) y2] [(int) z2];
    return value;
}

```

**Figure 14.14**

Smoothing of 3D textures, at various zooming levels.

The `smoothNoise()` function computes a grayscale value for each texel in the smoothed version of a given noise map by computing a weighted average of the eight grayscale values surrounding the texel in the corresponding original “blocky” noise map. That is, it averages the color values at the eight vertices of the small “block” the texel is in. The weights for each of these “neighbor” colors are based on the texel’s distance to each of its neighbors, normalized to the range (0...1).

Next, smoothed noise maps of various zooming factors are *combined*. A new noise map is created in which each of its texels is formed by another weighted average, this time based on the sum of the texels at the same location in each of the “smoothed” noise maps, with the zoom factor serving as the weight. The effect has been dubbed “turbulence,” although it is really more closely related to the harmonics produced by summing various waveforms. A new `turbulence()` function, and a modified version of `filldataArray()` that specifies a noise map that sums zoom levels 1 through 32 (the ones that are powers of two) is shown below, followed by an image of a cube textured with the resulting noise map.

```

private double turbulence(double x, double y, double z, double maxZoom)
{   double sum = 0.0, zoom = maxZoom;
    while(zoom >= 1.0) // the last pass is when zoom=1.
    {   // compute weighted sum of smoothed noise maps
        sum = sum + smoothNoise(x/zoom, y/zoom, z/zoom) * zoom;
        zoom = zoom / 2.0;           // for each zoom factor that is a power of two.
    }
    sum = 128.0 * sum / maxZoom; // guarantees RGB < 256 for maxZoom values up to 64
    return sum;
}

private void filldataArray(byte data[])
{
    double maxZoom = 32.0;
    for (int i=0; i<noiseWidth; i++)
    {   for (int j=0; j<noiseHeight; j++)
        {   for (int k=0; k<noiseDepth; k++)
            {   data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] =
                (byte) turbulence (i, j, k, maxZoom);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] =
                (byte) turbulence (i, j, k, maxZoom);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] =
                (byte) turbulence (i, j, k, maxZoom);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] =
                (byte) 255;
            }
        }
    }
}

```

3D noise maps, such as the one shown in Figure 14.15, can be used for a wide variety of imaginative applications. In the next sections, we will use it to generate marble, wood, and clouds. The distribution of the noise can be adjusted by various combinations of zoom-in levels.

14.6 NOISE APPLICATION – MARBLE

By modifying the noise map and adding Phong lighting with an appropriate ADS material as described previously in Figure 7.3, we can make the dragon model appear to be made of a marble-like stone.

We start by generating a striped pattern somewhat similar to the “stripes” example from earlier in this chapter—the new stripes differ from the previous ones, first because they are diagonal, and also because they are produced by a sine wave and therefore have blurry edges. We then use the noise map to perturb those lines, storing them as grayscale values. The changes to the `filldataArray()` function are as follows:

```
private void filldataArray(byte data[])
{
    double veinFrequency = 2.0;
    double turbPower = 1.5;
    double maxZoom = 64.0;
    for (int i=0; i<noiseWidth; i++)
    {
        for (int j=0; j<noiseHeight; j++)
        {
            for (int k=0; k<noiseDepth; k++)
            {
                double xyzValue = i / noiseWidth + j / noiseHeight + k / noiseDepth
                               + turbPower * turbulence(i,j,k,maxZoom) / 256.0;
                double sineValue = Math.abs(Math.sin(xyzValue * 3.14159 * veinFrequency));
                Color c = new Color((float)sineValue, (float)sineValue, (float)sineValue);
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4)+ k*4 + 0] = (byte) c.getRed();
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4)+ k*4 + 1] = (byte) c.getGreen();
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4)+ k*4 + 2] = (byte) c.getBlue();
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4)+ k*4 + 3] = (byte) 255;
            }
        }
    }
}
```

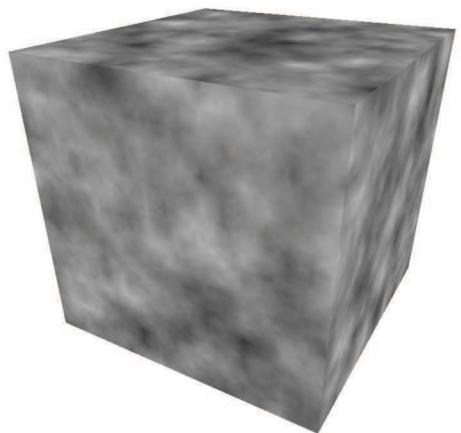
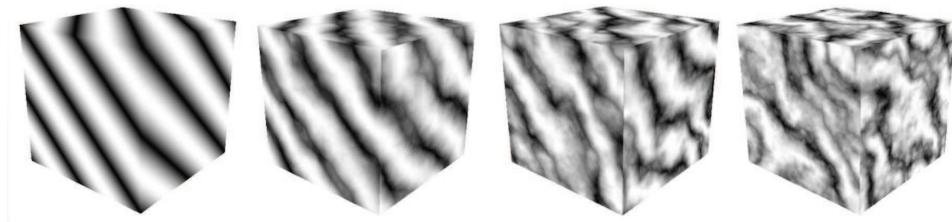


Figure 14.15
3D texture map with combined “turbulence” noise.

**Figure 14.16**

Building 3D “marble” noise maps.

The variable `veinFrequency` is used to adjust the number of stripes, `turbSize` adjusts the zoom factor used when generating the turbulence, and `turbPower` adjusts the amount of perturbation in the stripes (setting it to zero leaves the stripes unperturbed). Since the same sine wave value is used for all three (RGB) color components, the final color stored in the image data array is grayscale. Figure 14.16 shows the resulting texture map for various values of `turbPower` (0.0, 0.5, 1.0, and 1.5, left to right).

Since we expect marble to have a shiny appearance, we incorporate Phong shading to make a “marble” textured object look convincing. Program 14.5 summarizes the code for generating a marble dragon. The vertex and fragment shaders are the same as used for Phong shading, except that we also pass through the original vertex coordinates for use as 3D texture coordinates (as described earlier). The fragment shader builds a weighted sum to combine the noise result with the lighting result.

Program 14.5 Building a Marble Dragon

Java/JOGL application

```
...
public void init(GLAutoDrawable drawable)
{
    ...
    // material ADS settings for use in Phong Shading
    float[] matAmbient = new float[] {0.5f, 0.5f, 0.5f, 1.0f};
    float[] matDiffuse = new float[] {0.5f, 0.5f, 0.5f, 1.0f};
    float[] matSpecular = new float[] {1.0f, 0.5f, 0.5f, 1.0f};
    thisMaterial.setAmbient(matAmbient);
    thisMaterial.setDiffuse(matDiffuse);
```

```

    thisMaterial.setSpecular(matSpecular);
    thisMaterial.setShininess(50.0f);
}

private void filldataArray(byte data[ ])
{
    double veinFrequency = 2.0;
    double turbPower = 3.0;
    double turbSize = 64.0;
    // remainder is as shown above for building the marble noise map
    ...
}

```

Vertex Shader:

```
// unchanged from program 14-4
```

Fragment Shader:

```

...
void main(void)
{
    ...
    fragColor =
        0.5 * (globalAmbient * material.ambient + light.ambient * material.ambient
            + light.diffuse * material.diffuse * max(cosTheta,0.0)
            + light.specular * material.specular * pow(max(cosPhi,0.0), material.shininess))
        + 0.5 * texture(s, originalPosition / 3.0 + 0.5);
    // model vertices are (-1.5...+1.5), texture coordinates are (0...1)
}

```

By modifying the ADS material values specified in `init()`, we can simulate different marble or jade-like stones. Figure 14.17 shows two examples, the first using the ADS material values shown in Program 14.5, and the second utilizing the “jade” ADS colors shown previously in Figure 7.3.

14.7 NOISE APPLICATION – WOOD

Creating a “wood” texture can be done in a similar way as was done in the previous “marble” example. Trees grow in *rings*, and it is these rings that produce the “grain” we see in objects made of wood. As trees grow, environmental stresses create variations in the rings, which we also see in the grain.

**Figure 14.17**

Dragon textured with 3D marble noise maps

We start by building a procedural “rings” 3D texture map, similar to the “checkerboard” from earlier in this chapter. We then use the noise map to perturb those rings, inserting dark and light brown colors into the ring texture map. By adjusting the number of rings, and the degree to which we perturb the rings, we can simulate wood with various types of grain. Shades of brown can be made by combining similar amounts of red and green, with less blue. We then apply Phong shading with a low level of “shininess.”

We can generate rings encircling the Z-axis in our 3D texture map by modifying the `fillDataArray()` function, using trigonometry to specify values for X and Y that are equidistant from the Z axis. We use a sine wave to repeat this process cyclically, raising and lowering the red and green components equally based on this sine wave, to produce the varying

shades of brown. The variable `sineValue` holds the exact shade, which can be adjusted by slightly offsetting one or the other (in this case increasing the red by 80, and the green by 30). We can create more (or fewer) rings by adjusting the value of `xyPeriod`. The resulting texture is shown in Figure 14.18.

```
private void filldataArray(byte data[])
{
    double xyPeriod = 30.0;
    for (int i=0; i<noiseWidth; i++)
    {
        for (int j=0; j<noiseHeight; j++)
        {
            for (int k=0; k<noiseDepth; k++)
            {
                double xValue = (i - (double)noiseWidth/2.0) / (double)noiseWidth;
                double yValue = (j - (double)noiseHeight/2.0) / (double)noiseHeight;
                double distanceFromZ = Math.sqrt(xValue * xValue + yValue * yValue);
                double sineValue = 128.0 * Math.abs(Math.sin(2.0 * xyPeriod * distanceFromZ * 3.14159));
                Color c = new Color((int)(80+(int)sineValue), (int)(30+(int)sineValue), 30);
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4+0] = (byte) c.getRed();
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4+1] = (byte) c.getGreen();
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4+2] = (byte) c.getBlue();
                data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4+3] = (byte) 255;
            }
        }
    }
}
```

The wood rings in Figure 14.18 are a good start, but they don't look very realistic—they are too perfect. To improve this, we use the noise map (more specifically, the turbulence) to perturb the `distanceValue` variable so that the rings have slight variations. The computation is modified as follows:

```
double distanceValue = Math.sqrt(xValue * xValue + yValue * yValue)
+ turbPower * turbulence(i, j, k, maxZoom) / 256.0;
```

Again, the variable `turbPower` adjusts how much turbulence is applied (setting it to 0.0 results in the unperturbed version shown in Figure 14.18), and `maxZoom` specifies the zoom value (typically 32). Figure 14.19 shows the resulting wood textures for `turbPower` values 0.05, 1.0, and 2.0 (left to right).

We can now apply the 3D wood texture map to a model. The realism of the texture can be further enhanced by applying a rotation to the `originalPosition` vertex locations used for texture coordinates; this is because most items carved out of wood don't perfectly align with the orientation of the rings. To accomplish this, we send an additional rotation matrix to the shaders for rotating the texture coordinates. We also add Phong shading, with appropriate wood-color ADS values, and a modest level of shininess. The complete additions and changes for creating a “wood dolphin” are shown in Program 14.6.

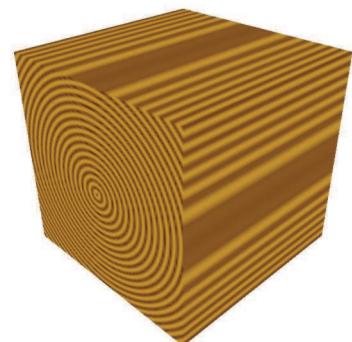


Figure 14.18
Creating rings for 3D wood texture.

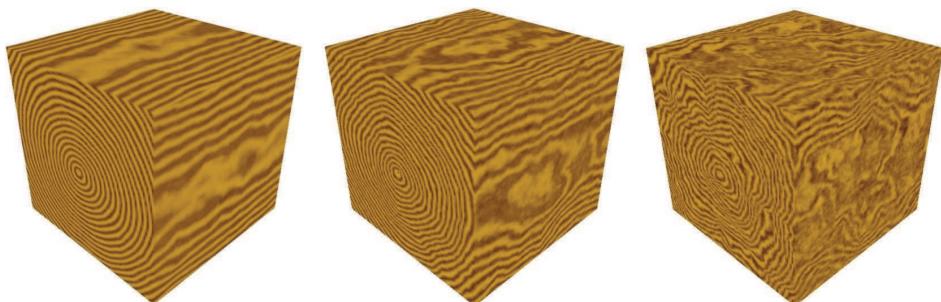


Figure 14.19
“Wood” 3D texture maps with rings perturbed by noise map.

Program 14.6 Creating a Wood Dolphin

Java/JOGL application:

```

public void init(GLAutoDrawable drawable)
{
    ...
    float[ ] brownAmbient = new float[ ] {0.5f, 0.35f, 0.15f, 1.0f};
    float[ ] brownDiffuse = new float[ ] {0.5f, 0.35f, 0.15f, 1.0f};
    float[ ] brownSpecular = new float[ ] {0.5f, 0.35f, 0.15f, 1.0f};
    thisMaterial.setAmbient(brownAmbient);
    thisMaterial.setDiffuse(brownDiffuse);
    thisMaterial.setSpecular(brownSpecular);
    thisMaterial.setShininess(15.0f);
    texRot.setToldentity();

    // rotation to be applied to texture coordinates – adds additional grain variation
    texRot.rotateX(35.0f); texRot.rotateY(35.0f); texRot.rotateZ(35.0f);
}

private void fillDataArray(byte data[ ])
{
    double xyPeriod = 40.0;
    double turbPower = 0.1;
    double maxZoom = 32.0;
    for (int i=0; i<noiseWidth; i++)
    {
        for (int j=0; j<noiseHeight; j++)
        {
            for (int k=0; k<noiseDepth; k++)
            {
                double xValue = (i - (double)noiseWidth/2.0) / (double)noiseWidth;
                double yValue = (j - (double)noiseHeight/2.0) / (double)noiseHeight;
                double distanceValue = Math.sqrt(xValue * xValue + yValue * yValue)
                    + turbPower * turbulence(i, j, k, maxZoom) / 256.0;
                double sineValue = 128.0 * Math.abs(Math.sin(2.0 * xyPeriod * distanceValue * 3.14159));
                Color c = new Color((int)(80+(int)sineValue), (int)(30+(int)sineValue), 30);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (byte) c.getRed();
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (byte) c.getGreen();
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (byte) c.getBlue();
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (byte) 255;
            }
        }
    }
}

private void display(GLAutoDrawable drawable)
{
    ...
    t_location = gl.glGetUniformLocation(rendering_program, "texRot");
    gl.glUniformMatrix4fv(t_location, 1, false, texRot.getFloatValues(), 0);
    ...
}

```

Vertex shader:

```
...
uniform mat4 texRot;
...
void main(void)
{
    ...
    originalPosition = vec3(texRot * vec4(position, 1.0)).xyz;
    ...
}
```

Fragment shader:

```
void main(void)
{
    ...
    uniform mat4 texRot;
    ...
    // combine lighting with 3D texturing
    fragColor =
        0.5 * (...)
        +
        0.5 * texture(s,originalPosition / 2.0 + 0.5);
}
```

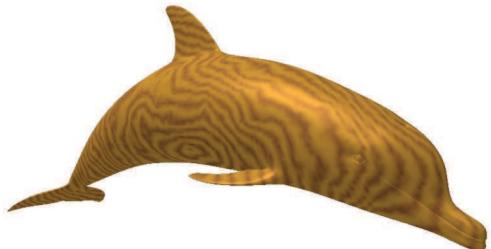


Figure 14.20
Dolphin textured with “wood” 3D noise map.

The resulting 3D textured wood dolphin is shown in Figure 14.20.

There is one additional detail in the fragment shader worth noting. Since we are rotating the model within the 3D texture, it is sometimes possible for this to cause the vertex positions to move beyond the required (0,1) range of texture coordinates as a result of the rotation. If this were to happen, we could adjust for this possibility by dividing the original vertex positions by a larger number (such as 4.0 rather than 2.0), and then adding a slightly larger number (such as 0.6) to center it in the texture space.

14.8 NOISE APPLICATION – CLOUDS

The “turbulence” noise map built earlier in Figure 14.15 already looks a bit like clouds. Of course, it isn’t the right color, so we start by changing it from grayscale to an appropriate mix of light blue and white. A straightforward way of doing

this is to assign a color with a maximum value of 1.0 for the blue component, and varying (but equal) values between 0.0 and 1.0 for the red and green components depending on the values in the noise map. The new `filldataArray()` function follows:

```
private void filldataArray(byte data[ ])
{   for (int i=0; i<noiseWidth; i++)
{   for (int j=0; j<noiseHeight; j++)
{   for (int k=0; k<noiseDepth; k++)
{   float brightness = 1.0f - (float) turbulence(i,j,k,32) / 256.0f;
Color c = new Color(brightness, brightness, 1.0f, 1.0f);
data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (byte) c.getRed();
data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (byte) c.getGreen();
data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (byte) c.getBlue();
data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (byte) 255;
} } } }
```

The resulting blue version of the noise map can now be used to texture a skydome. Recall that a skydome is a sphere or half-sphere that is textured, rendered with depth-testing disabled, and placed so that it surrounds the camera (similar to a skybox).

One way of building the skydome would be to texture it in the same way as we have for other 3D textures, using the vertex coordinates as texture coordinates. However, in this case, it turns out that using the skydome's 2D texture coordinates instead produces patterns that look more like clouds, because the spherical distortion slightly stretches the texture map horizontally. We can grab a 2D slice from the noise map by setting the third dimension in the GLSL `texture()` call to a constant value. Assuming that the skydome's texture coordinates have been sent to the OpenGL pipeline in a vertex attribute, in the standard way, the following fragment shader textures it with a 2D slice of the noise map:

```
#version 430
in vec2 texCoord;
out vec4 fragColor;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler3D s;

void main(void)
{   fragColor = texture(s,vec3(tc.x, tc.y, 0.5));    // constant value in place of tc.z
}
```

The resulting textured skydome is shown in Figure 14.21. Although the camera is usually placed inside the skydome, we have rendered it here with the camera outside, so that the effect on the dome itself can be seen. The current noise map leads to “misty-looking” clouds.

Although our misty clouds look nice, we would like to be able to shape them—that is, make them more or less hazy. One way of doing this is to modify the `turbulence()` function so that it uses an exponential, such as a logistic function,² to make the clouds look more “distinct.” The modified `turbulence()` function is shown in Program 14.7, along with an associated `logistic()` function. The complete Program 14.7 also incorporates the `smooth()`, `filldataArray()`, and `generateNoise()` functions described earlier.

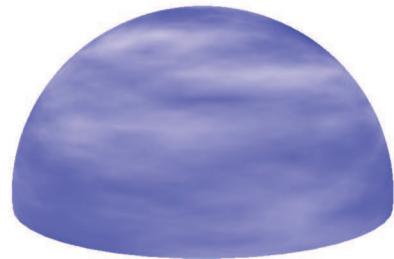


Figure 14.21
Skydome textured with misty clouds.

Program 14.7 Cloud Texture Generation

```
private double turbulence(double x, double y, double z, double size)
{
    double value = 0.0, initialSize = size, cloud_quant;
    while(size >= 0.9)
    {
        value = value + smoothNoise(x/size, y/size, z/size) * size;
        size = size / 2.0;
    }
    cloud_quant = 110.0;      // tunable quantity of clouds
    value = value/initialSize;
    value = 256.0 * logistic(value * 128.0 - cloud_quant);
    return value;
}

private double logistic(double x)
{
    double k = 0.2; // tunable haziness of clouds, produces more or less distinct cloud boundaries
    return (1.0 / (1.0+Math.pow(2.718, -k*x)));
}
```

² A “logistic” (or “sigmoid”) function has an S-shaped curve with asymptotes on both ends. Common examples are hyperbolic tangent and $f(x) = 1/(1+e^{-x})$. They are also sometimes called “squashing” functions.



Figure 14.22
Skydome with exponential cloud texture.

The logistic function causes the colors to tend more towards white or blue, rather than values in-between, producing the visual effect of there being more distinct cloud boundaries. The variable `cloud_quant` adjusts the relative amount of white (versus blue) in the noise map, which in turn leads to more (or fewer) generated white regions (i.e., distinct clouds) when the logistic function is applied. The resulting skydome, now with more distinct cloud formations, is shown in Figure 14.22.

Lastly, real clouds aren't static. To enhance the realism of our clouds, we should animate them by (a) making them move or "drift" over time and (b) gradually changing their form as they drift.

One simple way of making the clouds "drift" is to slowly rotate the skydome. This isn't a perfect solution, as real clouds tend to drift in a straight direction rather than rotating around the observer. However, if the rotation is slow and the clouds are simply for decorating a scene, the effect is likely to be adequate.

Having the clouds gradually change form as they drift may at first seem tricky. However, given the 3D noise map we have used to texture the clouds, there is actually a very simple and clever way of achieving the effect. Recall that although we constructed a 3D texture noise map for clouds, we have so far only used one "slice" of it, in conjunction with the skydome's 2D texture coordinates (we set the "Z" coordinate of the texture lookup to a constant value). The rest of the 3D texture has so far gone unused.

Our trick will be to replace the texture lookup's constant "Z" coordinate with a variable that changes gradually over time. That is, as we rotate the skydome, we gradually increment the depth variable, causing the texture lookup to use a different slice. Recall that when we built the 3D texture map, we applied smoothing to the color changes along all three axes. So, neighboring slices from the texture map are very similar, but slightly different. Thus, by gradually changing the "Z" value in the `texture()` call, the appearance of the clouds will gradually change.

The code changes to cause the clouds to slowly move and change over time are shown in Program 14.8.

Program 14.8 Animating the Cloud Texture

Java/JOGL application

```
private double rotAmt = 0.0; // Y-axis rotation amount to make clouds appear to drift
private float depth = 0.01f; // depth lookup for 3D noise map, to make clouds gradually change
...
public void display(GLAutoDrawable drawable)
{
    ...
    // gradually rotate the skydome
    m_matrix.setT0Identity();
    m_matrix.translate(objLoc.getX(), objLoc.getY(), objLoc.getZ());
    rotAmt = rotAmt + 0.2;
    m_matrix.rotateY(rotAmt);

    // gradually alter the third texture coordinate to make clouds change
    dloc = gl.glGetUniformLocation(program, "d");
    depth = depth + 0.00005f;
    if (depth >= 0.99) depth = 0.01f; // wrap-around when we get to the end of the texture map
    gl.glUniform1f(dloc, depth);
    ...
}
```

Fragment Shader:

```
#version 430

in vec2 tc;
out vec4 fragColor;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler3D s;
uniform float d;

void main(void)
{
    fragColor = texture(s, vec3(tc.x, tc.y, d)); // gradually-changing "d" replaces previous constant
}
```

While we cannot show the effect of gradually changing drifting and animated clouds in a single still image, Figure 14.23 shows such changes in a series of snapshots of the 3D generated clouds as they drift across the skydome from right to left.



Figure 14.23
3D clouds changing while drifting.

14.9 NOISE APPLICATION – *SPECIAL EFFECTS*

Noise textures can be used for a variety of special effects. In fact, there are so many possible uses that its applicability is limited only by one's imagination.

One very simple special effect that we will demonstrate here is a *dissolve effect*. This is where we make an object appear to gradually dissolve into small particles, until it eventually disappears. Given a 3D noise texture, this effect can be achieved with very little additional code. Note that this example only works if animation is incorporated, such as with the use of FPSAnimator.

To facilitate the dissolve effect, we introduce the GLSL `discard` command. This command is only legal in the fragment shader, and when executed, it causes the fragment shader to discard the current fragment (meaning not render it).

Our strategy is a simple one. In the Java/JOGL application, we create a fine-grained noise texture map identical to the one shown back in Figure 14.12, and also a float variable counter that gradually increases over time. This variable is then sent down the shader pipeline in a uniform variable, and the noise map is also placed in a texture map with an associated sampler. The fragment shader then accesses the noise texture using the sampler—in this case we use the returned noise value to determine whether or not to discard the fragment. We do this by

comparing the grayscale noise value against the counter, which serves as a sort of “threshold” value. Because the threshold is gradually changing over time, we can set it up so that gradually more and more fragments are discarded. The result is that the object appears to gradually dissolve. Program 14.9 shows the relevant code sections, which are added to the earth-rendered sphere from Program 6.1. The generated output is shown in Figure 14.24.

Program 14.9 Dissolve Effect Using *discard* Command

Java/JOGL application:

```
private float threshold = 0.0f; // gradually-increasing threshold for retaining/discardng fragment
...
...
```

in Display:

```
...
t_location = gl.glGetUniformLocation(rendering_program, "t");
threshold = threshold + .002f;
gl.glUniform1f(t_location, threshold);
...
gl.glActiveTexture(GL_TEXTURE0);
gl.glBindTexture(GL_TEXTURE_3D, textureID);           // noise texture
gl.glActiveTexture(GL_TEXTURE1);
gl.glBindTexture(GL_TEXTURE_2D, textureIDearth);      // earth texture
...
gl.glDrawArrays(GL_TRIANGLES, 0, numSphereVertices);
```

Fragment Shader:

```
#version 430
in vec2 tc;          // texture coordinates for this fragment
in vec3 origPos;    // original vertex positions in the model, for accessing 3D texture
...
layout (binding=0) uniform sampler3D n;           // sampler for noise texture
layout (binding=1) uniform sampler2D e;           // sampler for earth texture
...
uniform float t; // threshold for retaining or discarding fragment
void main(void)
{
    float noise = texture(n, origPos).x; // retrieve noise value for this fragment.
    if (noise > t)                  // if the noise value is greater than the current threshold value,
```

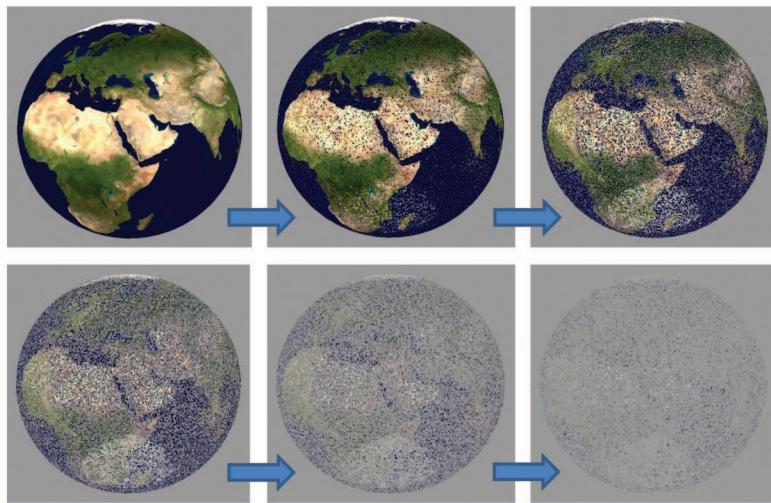


Figure 14.24
Dissolve effect with discard shader.

```
{
    fragColor = texture(e, tc);           // render the fragment using the earth texture.
}
else
{
    discard;   // otherwise, discard the fragment (do not render it)
}
}
```

The `discard` command should, if possible, be used sparingly because it can incur a performance penalty. This is because its presence makes it more difficult for OpenGL to optimize Z-buffer depth testing.

SUPPLEMENTAL NOTES

In this chapter, we used Perlin noise to generate clouds, and to simulate both wood and a marble-like stone from which we rendered a dragon and a dolphin. People have found many other uses for Perlin noise. For example, it can be used to create fire and smoke [KE16, AF14], build realistic bump maps [GR05], and has been used to generate terrain in the video game Minecraft [PE11].

The noise maps generated in this chapter are based on procedures outlined by Lode Vandevenne [VA04]. There remain some deficiencies in our 3D cloud generation. The texture is not seamless, so at the 360° point there is a noticeable vertical line. (This is also why we started the depth variable in Program 14.8 at 0.01 rather

than at 0.0—to avoid encountering the seam in the Z dimension of the noise map). Simple methods exist for removing the seams [AS04], if needed. Another issue is at the northern peak of the skydome where the spherical distortion in the skydome causes a pincushion effect.

The clouds we implemented in this chapter also fail to model some important aspects of real clouds, such as the way that they scatter the sun’s light. Real clouds also tend to be more white on the top and grayer at the bottom. Our clouds also don’t achieve a 3D “fluffy” look that many actual clouds have.

Similarly, more comprehensive models exist for generating fog, such as the one described by Kilgard and Fernando [KF03].

While perusing the OpenGL documentation, the reader might notice that GLSL includes some noise functions named `noise1()`, `noise2()`, `noise3()`, and `noise4()`, that are described as taking an input seed and producing Gaussian-like stochastic output. We didn’t use these functions in this chapter because as of this writing, most vendors have not implemented them. For example, many NVIDIA cards currently return 0 for these functions, regardless of the input seed.

Exercises

- 14.1** Modify Program 14.2 to gradually increase the alpha value of an object, causing it to gradually fade out and eventually disappear.
- 14.2** Modify Program 14.3 to clip the torus along the horizontal, creating a circular “trough.”
- 14.3** Modify Program 14.4 (the version including the modification in Figure 14.10 that produces a 3D cubed texture), so that it instead textures a simpler curved object, such as the Studio 522 dolphin. Then observe the results. Many people when first observing the result—such as that shown on the dragon, but even on simpler objects—believe that there is some error in the program. Unexpected surface patterns can result from “carving” an object out of 3D textures, even simple ones.
- 14.4** The simple sine wave used to define the wood “rings” (shown in Figure 14.18) generate rings in which the light and dark areas are equal width. Experiment with modifications to the associated `fillDataArray()` function with the goal of making the dark rings narrower in width than the light rings. Then observe the effects on the resulting wood-textured object.

- 14.5 (PROJECT)** Incorporate the logistic function (from Program 14.7) into the marble dragon from Program 14.5, and experiment with the settings to create more distinct veins.
- 14.6** Modify Program 14.9 to incorporate the zooming, smoothing, turbulence, and logistic steps described in the prior sections. Observe the changes in the resulting dissolve effect.

References

- [AF14] S. Abraham and D. Fussell, “Smoke Brush,” *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering (NPAR’14)*, 2014, accessed July 2016, <https://www.cs.utexas.edu/~theshark/smokebrush.pdf>.
- [AS04] D. Astle, “Simple Clouds Part 1,” gamedev.net, 2004, accessed July 2016, http://www.gamedev.net/page/resources/_/technical/game-programming/simple-clouds-part-1-r2085.
- [GR05] S. Green, “Implementing Improved Perlin Noise,” *GPU Gems 2*, NVIDIA, 2005, accessed July 2016, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter26.html.
- [JA12] A. Jacobson, “Cheap Tricks for OpenGL Transparency,” 2012, accessed July 2016, <http://www.alecjacobson.com/weblog/?p=2750>.
- [KE16] B. Kaskosz, D. Ensley, and D. Gries, “Billowing Fire with Perlin Noise and Filters – Flash AS3 Effect,” *Flash & Math*, accessed July 2016, <http://www.flashandmath.com/about/>.
- [KF03] M. Kilgard and R. Fernando, “Advanced Topics,” *The Cg Tutorial* (Addison-Wesley, 2003), accessed July 2016, http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html.
- [LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).
- [PE11] M. Persson, “Terrain Generation, Part 1,” *The Word of Notch* (blog), Mar 9, 2011, accessed July 2016, <http://notch.tumblr.com/post/3746989361/terrain-generation-part-1>.
- [VA04] L. Vandevenne, “Texture Generation Using Random Noise,” *Lode’s Computer Graphics Tutorial*, 2004, accessed July 2016, <http://lodev.org/cgtutor/randomnoise.html>.

Index

3D textures

- complex patterns, 315
- Java/JOGL application, 314
- striped pattern, 311, 312*f*, 312–314

A

Adding primitives, 294–297

ADS lighting model, 158, 163
computations

- ambient contribution, 165
- diffuse contribution, 166–167
- specular contribution, 167–168

implementing
faceted shading, 169
smooth shading. *see* Smooth
shading

Aliasing, 115, 116

Alpha channel, 302

Altering primitives, 288–293

Ambient reflection, 158

Animation, 30–32

Anisotropic filtering (AF), 120
normal mapping, 241, 242*f*

Application Programming Interfaces

(APIs), 2

Aspect ratio, 53

Attenuation factors, 160, 161

B

Back-face culling, 97–99

Bernstein, Sergei, 251

Bézier curve, 249, 249*f*

Bézier surfaces, 137, 249

cubic, 256–258

quadratic, 254–256

tessellation

control shader, 270

evaluation shader, 270–272

fragment shader, 272

Java/JOGL application, 272

mix() function, 273

overview of, 267, 267*f*

TCS, 268

TES, 268–269

vertex shader, 272, 273

Bézier, Pierre, 249

- Bilinear filtering, 118
- Bitangent vectors, 235, 236
- Blender, 144, 145, 146
- Blending function, 251
- Blending. *See* Transparency
- Blinn, James, 180
- Blinn-Phong shading, 180
- Bronze, 164
- Buffer, 9, 16
- Bump mapping
- perturbed normal vectors for, 232, 232*f*
 - procedural bump mapping, 232, 233*f*
- C**
- Camera space. *See* Eye space
- Camera, 49, 50
- Clipping plane, 308–310
- Clouds, 327–332
- Color buffer, 21, 22
- Compositing. *See* Transparency
- Computer aided design, 56
- Concatenation, 39
- Cross product, 47
- Crow, Frank, 191
- Cube map
- camera placed inside, 212, 212*f*
 - six-faced skybox texture, 211, 212*f*
 - texture coordinates, 212, 213*f*
 - using OpenGL, 219–223
- Culling. *See* Back-face culling
- D**
- de Casteljau, Paul, 249
- de Casteljau’s algorithm, 253
- Debugging, 26, 27
- Deleting primitives, 293–294
- Depth buffer. *See* Z-buffer
- Depth-fighting, 94
- Diffuse reflection, 158
- Digital Content Creation (DCC), 144
- Direct buffer, 67
- Directional light, 159–160
- Dissolve effect, 332
- Distant light. *See* Directional light
- 3D models
- loading externally produced models
 - Blender, 145, 146
 - DCC-created model, 144
 - ImportedModel classes, 149
 - ModelImporter class, 149
 - OBJ files, 145, 147
- OpenGL indexing
- inner and outer variables, 139
 - torus, 137–138
 - VBO, 139
- procedural models
- objects, types of, 130
 - sphere vertex, 132, 132*f*, 136
 - vertices, 133
- Dot product, 45–47
- E**
- Emissiveness, 164
- Enhancing surface detail, 231–247
- Environment mapping
- Blinn-Phong lighting, 224–225
 - overview, 224, 224*f*
 - reflection vector, 224
 - shaders, sets of, 227
 - shininess, 223
 - texture cube maps, 223
- Erroneous self-shadowing.
- See* Shadow acne
- Euler angles, 42
- Euler’s Theorem, 42
- Eye space, 49

F

- Faceted shading, 169
- Far clipping plane, 53
- Field of view, 53
- Flat shading. *See* Faceted shading
- FOG, 299–301
- FPSAnimator, 30, 31
- Fragment shader, 20–21
- Frame rate, 30, 76
- Frame, 30
- Frustum, 53
- Full-scene anti-aliasing (FSAA), 126

G

- Geometry shaders
 - adding primitives, 294–297
 - altering primitives, 288–293
 - deleting primitives, 293–294
 - OpenGL, per-primitive processing in, 287–288
- GLCanvas, 9, 30
- Global ambient light, 159
- GLSL, 2
 - building matrix transforms, 57–59
 - error-detecting modules
 - checkGLError(), 23, 24
 - printProgramLog, 24
 - printShaderLog, 24
 - GLSL shader code, files, 27–28
 - shader languages, 8
 - texture mapping, 104
- GNU Image Manipulation Program (GIMP), 234, 237

Gold, 164

Gouraud shading, 169–177

Gouraud, Henri, 169

Graphics card (GPU), 1

Graphics pipeline. *See* OpenGL pipeline

Graphics programming

four languages and libraries

graphicslib3D, 3

Java, 2

JOGL, 3

OpenGL / GLSL, 2

installation and configuration

graphicslib3D, 4–5

Java, 3

JOGL, 4

OpenGL / GLSL, 4

Graphicslib3D

building scale matrices, 41

languages and libraries, 3

rotation matrices, 42

torus, 140

installation and configuration, 4–5

H

Hard shadows, 209

Height mapping

defining, 242

interpretation, 242, 243*f*

terrain, 244, 244*f*

vertex manipulation, 243

vertex-shader-based, 244, 245,

245*f*, 246

Hidden surface removal algorithm

(HSR), 21–22, 192

Hierarchical model

88

Homogeneous notation

36

HSR. *See* hidden surface removal

I

Identity matrix, 37

Immediate mode, 2

Indexing, 137

Instancing, 81–84

J

- Jade, 163, 164
- Java Development Kit (JDK), 3
- Java OpenGL (JOGL)
 - animator classes, 30
 - overview of, 8, 8*f*
 - vertices, 29
- Java
 - installation and configuration, 3
 - languages and libraries, 2
- JogAmp, 3
- JOGL
 - installation and configuration, 4
 - languages and libraries, 3
 - OpenGL texture object, 105
 - texture mapping, 104

L

- Layout qualifier, 66
- Level of detail (LOD), 281–284
- Lighting
 - ADS lighting computations
 - ambient contribution, 165
 - diffuse contribution, 166–167
 - specular contribution, 167–168
 - ADS model, 158
 - combining lighting and textures, 182–184
 - directional/distant light, 159–160
 - global ambient, 159
 - implementing ADS lighting
 - faceted shading, 169
 - smooth shading. *see* Smooth shading
 - materials, 163–165
 - positional, 160–161
 - reflection model, 158
 - shading models, 158

- shadows, 190*f*
- spotlights, 161–163
- types of, 159
- Lightweight Java Game Library (LWJGL), xiv
- Linear filtering, 118
- Local space. *See* Model space
- LOD. *See* level of detail
- Logistic function, 330
- Look-at matrix, 56–57
- Luxo Jr., 163

M

- Managing 3D graphics data
 - back-face culling, 97–99
 - buffers, types of, 67
 - combating Z-fighting artifacts, 94–95
- 3D cube
 - 3D cube program, 72–74
 - display() function, 76
 - fragment shader, 77, 78
 - frame rate, 76
 - translate() and rotate()
 - functions, 80
 - varyingColor, 79
 - vertex_positions, 75
- matrix stacks
 - display() function, 90
 - hierarchical models, 88
 - planet, 92
 - simple solar system, 92–93
 - sun's rotation, 90–91
 - view matrices, 89
- model-view and perspective matrices
 - defining, 69
 - model matrix, 70
 - perspective matrix, 70
 - view matrix, 70

primitives
line, 96
patch, 97
point, 96
triangle, 95–96
rendering multiple different models, 85–87
rendering multiple objects, 80–84
uniform variables, 67–68
vertex attributes
interpolation of, 68–69
vertex shader, 66
Marble, 322
Marble noise maps, 321–323
Mathematical foundations
3D coordinate systems, 36
building matrix transforms, GLSL functions for, 57–59
eye space and synthetic camera, 49–52
local and world space, 48
look-at matrix, 56–57
matrices
addition, 38
identity matrix, 37
inverse, 39
multiplication, 38–39
transformation, 40–44
transpose, 37–38
points, 36
projection matrices
orthographic projection, 54–56
perspective projection, 52–54
vectors
cross product, 47
dot product, 45–47
Materials, 163–165
Matrices
addition, 38
identity matrix, 37
inverse, 39
look-at matrix, 56–57
multiplication, 38–39
orthographic projection, 54–56
perspective projection, 52–54
projection matrices
orthographic projection, 54–56
perspective projection, 52–54
transformation
rotation, 42–44
scale, 41–42
translation, 40–41
transpose, 37–38
Matrix multiplication, 38–39
Matrix stacks
display() function, 90
hierarchical models, 88
planet, 92
simple solar system, 92–93
sun’s rotation, 90–91
view matrices, 89
Maya, 144
Mipmapping
aliasing artifacts, 115, 116
brick texture, 119
minification, 118
minification technique, 118
trilinear filtering, 118, 118f, 119
Modeling, 129–155
Model matrix, 48
Model space, 48
Model-view matrix, 67, 85

N

NASA, 149
Near clipping plane *See* Projection plane
Newell, Martin, 137
Noise. *See* Perlin Noise
Normal mapping
AF, 241, 242f

Blinn-Phong lighting, 235
fragment shader, 236
image file, 234
mipmapping, 241
moon surface rendered, 239
object normal, 235
tangent and bitangent vectors, 235, 236
TBN matrix, 237
texture units, application of, 234
Normal vector, 132, 224
NVIDIA, 335

O

OBJ. *See* Wavefront OBJ
Opacity, 302
OpenGL Architecture Review Board (ARB), 2
OpenGL camera, 50, 195
OpenGL extension, 120
OpenGL pipeline
 error-detecting modules, 23, 25–26
 enabling debugging causes, 26
 enabling tracing causes, 26
geometry shader, 18–19
hardware side, 7
Java/Jogl application
 C pointer, 12
 color buffer, 11–12
 Direct3D acceleration, 10
 GL4, 11
 GLCanvas, 9, 30
 overview of, 8, 9f
pixel operations, 21–22
rasterization, 19–20
shader stages, 8–9
software side, 7
tessellation, 17–18
vertex and fragment shaders, 20–21

glShaderSource(), 15
glUseProgram, 15
pixel processing, 17
primitives, 12
RGB color, 16
VAOs, 16
vshaderSource and
 fshaderSource, 15

OpenGL Shading Language. *See* GLSL

OpenGL

 glGenVertexArrays() and
 glGenBuffers(), 65
 per-primitive processing in, 287–288
 primitive types, 95
 VAOs, 65

Orthographic projection, 54–56, 209

P

Parametric surfaces
 cubic Bézier curves
 analytic definition for, 251
 de Casteljau's algorithm, 253
 recursive subdivision algorithm, 253, 254f, 254
 cubic Bézier surfaces, 256–258
 quadratic Bézier curve, 249–251
 quadratic Bézier surfaces, 254–256
Patch, 265
Pearl, 163
Perlin Noise
 3D noise data, cube textured with, 317, 317f
 noise application
 clouds texture, 327–332
 special effects, 332–334
 wood texture, 323–327
 marble noise maps, 321–323
smoothed noise maps, 320
zooming factors, 318

- Perlin, Ken, 316
Perspective correction, 123
Perspective distortion, 123–124
Perspective matrix, 53
Perspective transform, 53
Peter panning, 206, 207*f*
Pewter, 163
Phong shading
 Blinn-Phong shading, 180–181
 external models with, 181, 182*f*
 implementing Phong shading,
 177, 177*f*
 Stanford Dragon, 181–182
Phong, Bui Tuong, 177
Photoshop, 214, 237
Pipeline. *See* OpenGL pipeline
Pixar, 163
Point, 36
Popping, 283
Positional light, 160–161
Primitive assembly, 18
Primitives, 12
Procedural bump mapping,
 232, 233*f*
Projection matrices
 orthographic projection, 54–56
 perspective projection, 52–54
Projection plane, 49, 53, 55
Projective shadows, 190–191
- Q**
- Quadratic Bézier curve, 249–251
Quaternion, 44, 60
- R**
- Rasterization, 19–20
Reflection mapping. *See* Environment
 mapping
- Reflection model, 158
Right-hand rule, 47
- S**
- Sampler variable, 110
Sampler2D, 110
Sampler3D, 314, 315
SamplerCube, 220
Scale matrix, 41–42
Shader programs, 1, 2
Shading models, 158
Shadow acne, 205
Shadow buffering, 193
Shadow mapping artifacts
 jagged shadow edges,
 207, 208*f*
 peter panning, 206, 207*f*
 shadow acne, 205
 shadow bar, 207
Shadow texture, 193
Shadow volumes, 191–192, 191*f*
Shadows
 importance of, 189–190
 mapping
 artifacts, 205–208
 HSR algorithm, 192
 JOGL code, 200
 light position, draw objects from,
 193–194
 lighted scene, 199, 199*f*
 rendering scene, 195–199
 shadow buffering, 193
 shadow textures, 193
 strategy, 193
 Z-buffer, texture, 194–195
 projective shadows, 190–191
 volumes, 191–192, 191*f*
Shininess, 163, 168
Silver, 164

- Skyboxes
concept of, 211
implementing
 texture coordinate, 216
 texture images, 218
 using OpenGL cube maps, 219–223
texture cube map, 212
- Skydomes
advantages, 214
disadvantage, 214
using sphere, 215, 215*f*
- Smooth shading
 Gouraud shading, 169–177
 Phong shading, 177–182
- Soft shadows, 209
- Specular highlight, 167, 177, 239, 296
- Specular reflection, 158
- Spotlights, 161–163
- Stanford dragon, 181
- Starfield, 215, 215
- Stencil buffer, 192
- Studio 522 dolphin, 184
- T**
- Tangent vectors, 235, 236
- Teapot, Utah, 137
- Terragen, 214
- Terrain, 244
- Tessellation control shader (TCS), 267, 268
- Tessellation evaluation shader (TES), 268–269
- Tessellation levels, 266
- Tessellation primitive generator (TPG), 262
- Tessellation, 258
 controlling LOD, 281–284
 Bézier surfaces
 control shader, 270
- evaluation shader, 270–272
fragment shader, 272
Java/Jogl application, 272
mix() function, 273
overview of, 267, 267*f*
TCS, 268
TES, 268–269
vertex shader, 272, 273
- OpenGL
fragment shader, 267
inner level and outer level, 264, 266
modules, 262
patch, 265, 266
pipeline stages, 261
triangle mesh output, 264, 264*f*
- terrain/height maps
control shader, 278
evaluation shader, 278–279
fragment shader, 275
Java/Jogl application, 277
Phong shading, 280
vertex shader, 274–275, 277, 278
- Texels, 106
- Texture coordinates
3D model, 106
constructing, 108–109
cube model, 107
curved geometric shapes, 108
values, 110
vertex shader, 106
- Texture cube map. *See* Cube map
- Texture image, 104, 124
- Texture mapping, 193
anisotropic filtering (AF), 120
creating texture object, 108
JAVA AWT classes, 124–126
JOGL/GLSL, 104
mipmapping
 aliasing artifacts, 115, 116

- brick texture, 119
minification technique, 118
trilinear filtering, 118, 118*f*, 119
- OpenGL texture object, 105
perspective distortion, 123–124
pyramid’s model, 111
sampler variables and texture units, 110–111
- texture coordinates
3D model, 106
constructing, 108–109
cube model, 107
curved geometric shapes, 108
values, 110
vertex shader, 106
- type, 105
wrapping and tiling, 121–122
- Texture object, 108
Texture units, 103, 110–111
- Tiling. *See* Wrapping
- Tracing, 26
- Translation matrix, 40–41
- Transparency
alpha channel, 302
compositing process, 303
`glBlendEquation()` parameter, 304
`glBlendFunc()` parameters, 303
Java/JOGL application, 304
opacity, 302
Z-buffer, 302
- Trilinear filtering, 118, 118*f*, 119
- Turberville, Jay, 181
- Turbulence, 320, 327
- U**
- Uniform sampler variable, 110
Uniform variable, 31, 67–68
- User-defined clipping planes, 308–310
- UV-mapping, 108, 150
- V**
- VAO. *See* Vertex Array Object, 65
VBO. *See* Vertex Buffer Object, 64
Vecmath, 3
- Vectors
cross product, 47
dot product, 45–47
- Vertex Array Objects (VAOs), 16
- Vertex attributes
interpolation of, 68–69
vertex shader, 12, 66
- Vertex Buffer Object (VBO), 64, 139
- Vertex, 16, 17
- View matrix, 70
- View space. *See* Eye space
- Viewing transform matrix, 50
- W**
- Wavefront OBJ, 145
- Winding order, 97
- Wireframe rendering, 20*f*
- World space, 48
- Wood, 323
- Wrapping, 121–122
- Z**
- Z-buffer algorithm, 22, 22*f*, 94
Z-buffer, 22, 94, 95, 302
Z-fighting, 94

